Heterogeneous Multithreaded Computing

by

Howard J. Lu

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 1995

Copyright 1995 Howard J. Lu. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce
and to distribute copies of this thesis document in whole or in part,
and to grant others the right to do so.

Author_____

Department of Electrical Engineering and Computer Science

May 17, 1995

Certified by_____

Charles E. Leiserson

Thesis Supervisor

Accepted

by_____

F.R. Morgenthaler

Chairman, Department Committee on Graduate Theses

Heterogeneous Multithreaded Computing
by
Howard J. Lu

Submitted to the
Department of Electrical Engineering and Computer Science

May 1995

In Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

This thesis studies issues introduced by supporting multithreaded computing in environments composed of workers heterogeneous with respect to data representation, speed, and scale. We implement a heterogeneous multithreaded runtime system, and prove its efficiency.

We present an implementation of Cilk that supports execution in heterogeneous environments. To support communication between workers with different data representations, we propose two mechanisms. The first mechanism translates messages to a canonical form on transmission and to the native host format on reception. The second mechanism generates signature tables of structures that are communicated between the workers to guide translation. We propose but do not implement an extension to generate signature tables for all C functions so that functions can be passed as messages.

We prove the following: for a fully strict Cilk program with T_1 total work and a critical path of length T_∞, running on P workers with a maximum communication time σ_{max}, average performance π_{ave}, and maximum performance π_{max}, we obtain the execution time $T_P = O(T_1/\pi_{ave}P + \sigma_{max}T_\infty\pi_{max}/\pi_{ave})$. Furthermore, the "effective parallelism" of such a computation is only a factor of $\sigma_{max}\pi_{max}$ worse than the average parallelism T_1/T_∞.

Thesis Supervisor: Charles E. Leiserson
Title: Professor, Massachusetts Institute of Technology

1 Introduction

With the growing availability of computer resources, the idea of distributing computation across a set of processors has become very popular. Many runtime systems have been developed to implement the distribution of the work of a multithreaded parallel program across a homogeneous set of workers, such as a group of identical workstations or the nodes of a multiprocessor. Many issues and problems arise when trying to implement such a system in a heterogeneous environment, however. In this thesis, we examine the problems introduced by differences in data representation, speed, and scale. We also show both practically and analytically that a runtime system can run efficiently in a heterogeneous environment.

We have chosen to investigate these problems using the Cilk runtime system [1]. Cilk is a C-based runtime system that supports the execution of multithreaded programs in a continuation passing style across a homogeneous set of workers. We have chosen Cilk because it is simple, portable, and provably efficient in a homogeneous environment. In this thesis, we show that Cilk is provably efficient in heterogeneous environments as well.

To demonstrate the practicality and feasibility of a runtime system operating across a heterogeneous environment, we have produced two heterogeneous implementations of Cilk. The first runs on the MIT Phish network of workstations [3], an environment in which the workers are of the same scale but are heterogeneous with respect to their performance speeds. The second implementation runs on the PVM message-passing layer [6]. This implementation allows execution of a multithreaded program on any virtual computer composed of workers of different performances and scale.

In generating these two implementations of Cilk, we discovered the problem that heterogeneous workers could store data differently, requiring some form of translation to allow communication between workers. After examining different solutions to this problem, we have chosen to translate all messages to some canonical format when sending them, and translate them back into the native host format upon reception. This mechanism of translation requires that the structure of data is known beforehand, both at the source and destination of the message. Therefore, messages whose structure depends on the user's own code represent a problem. Our solution is to have the Cilk

preprocessor construct signature tables of all structures introduced by the user's code, and use these signature tables to guide the translation mechanism.

Having heterogeneous workers also poses the problem that the slowest workers might always be hoarding the computation, slowing down the total execution time of the program. Blumofe and Leiserson [2] showed that in a P-worker homogeneous environment, the execution time of a fully strict Cilk program with T_1 total work and a critical path of length T_∞ was $T_P = O(T_1/P + T_\infty)$. We show that in an environment in which each worker steals at the same rate, e.g., limited by the speed of a network, a fully strict Cilk program has a time of execution $T_P = O(T_1/(\pi_{ave}P) + T_\infty\pi_{max}/\pi_{ave})$, where π_{max} is the performance of the fastest worker and π_{ave} is the average performance of all P workers.

We also extend our analysis to prove that computing with workers of different scales does not affect the time of execution by more than a constant factor. We prove that for a fully strict Cilk program, the execution time is $T_P = O(T_1/(\pi_{ave}P) + T_\infty\sigma_{max}\pi_{max}/\pi_{ave})$, where σ_{max} is the longest amount of time it takes for a single worker to perform a steal.

We are generally interested in the environments in which the speedup T_1/T_P increases linearly with P. In homogeneous environments, linear speedup occurs when the number of workers P is at most the order of the average parallelism T_1/T_∞. We define the "effective parallelism" to be the maximum number of workers that still guarantees linear speedup. For homogeneous environments, the effective parallelism is equal to the average parallelism T_1/T_∞. A fully strict Cilk program running on P homogeneous workers has an execution time $T_P = O(T_1/P + T_\infty)$. When the number of workers P is at most the order of the average parallelism T_1/T_∞, T_1/P dominates the sum $T_1/P + T_\infty$. We derive $T_P = O(T_1/P)$ which yields the linear speedup $T_1/T_P = O(P)$ for the homogeneous case.

In a completely heterogeneous environment, we conclude that the "effective parallelism" is $O((T_1/T_\infty)(1/(\sigma_{max}\pi_{max})))$ because when the number of workers $P = O((T_1/T_\infty)(1/(\sigma_{max}\pi_{max})))$, we obtain linear speedup. From our theoretical bound on the execution time of a Cilk program in a heterogeneous environment, we note that all of the computational performance is applied to performing the total work. The only penalty incurred is a slight decrease in the effective parallelism by $\sigma_{max}\pi_{max}$, a constant factor.

We present some background information on the Cilk runtime system in section 2. In section 3, we investigate the issues brought about when computing in an environment that is heterogeneous with respect to data representation. In section 4, we explore the theoretical issues of computing in an environment of workers running in

different speeds. We analyze the implications of running in a heterogeneous environment of different scales in section 5. And in section 6, we present some implementation details for constructing Cilk to run heterogeneously with respect to data representation, speed, and scale.

2 Cilk

We have chosen the Cilk multithreaded runtime system [1] to study the issues of heterogeneous multithreaded computing, because Cilk is portable, simple, and provably efficient. In this section, we justify our choice of Cilk as our vehicle to study the issues of heterogeneous multithreaded computing. We also present background information about the Cilk runtime system.

Cilk was an ideal choice for studying heterogeneous multithreaded computing, because it is a simple runtime system that supports the execution of multithreaded programs in a homogeneous environment. Because of its simplicity (Cilk is only a couple thousand lines of code long), we were able to easily extend Cilk to operate in a heterogeneous environment.

Cilk's portability allows the implementation of its runtime system on a diverse collection of computers. Cilk is written entirely in C, and the runtime system runs on a vanilla C stack. Because of this, Cilk can be easily ported to different computers. Cilk can therefore be run across a diverse set of workers, such as the Connection Machine CM5 MPP, the Intel Paragon MPP, the Silicon Graphics Power Challenge SMP, the MIT Phish network of workstations [3], and on the PVM message-passing layer [6].

We also chose Cilk because it is efficient, both empirically and theoretically. The Cilk scheduling algorithm allows us to view a multithreaded computation as a bounded-degree directed acyclic graph of tasks. With this view, a fully strict Cilk program, running in a P processor homogeneous environment with total work T_1 and a critical path of length T_∞, has an expected time of execution $T_P = O(T_1/P + T_\infty)$. The execution uses space $S_P \leq PS_1$, where S_1 is the maximum amount of space used executing the same program on a single worker. And, the expected communication incurred by a P-worker execution is $O(S_{max}PT_\infty)$, where S_{max} is the size of the largest communicated data structure.

Cilk achieves its empirical and analytical efficiency through the use of its work-stealing scheduling paradigm. The Cilk runtime system schedules the execution of threads through a work-stealing algorithm: whenever there are idle workers, they attempt to "steal" tasks from other workers. This algorithm reduces the amount of

communication between workers, and still ensures that workers are rarely idle if work remains to be performed.

In the Cilk system, communication between workers is achieved through message-passing. All the threads executing on the workers can communicate with each other by sending a message through an underlying message-passing layer. Completed threads send their results as a message to a continuing thread. Data structures called closures are transmitted between workers when one worker manages to steal a task from another.

Because of Cilk's simplicity, portability, and efficiency, it was an ideal choice to study the issues of heterogeneity in multithreaded computing. Cilk's use of message-passing allowed us to construct a heterogeneous implementation of Cilk relatively easily.

3 Heterogeneity in Data Representation

In this section, we explore the problems and issues introduced when extending the Cilk runtime system to execute across a set of workers that can possess different representations of data. Allowing workers to represent data differently introduces the problem of a "language barrier": workers with different data representations cannot communicate without some form of translation. After considering different solutions, we chose to implement a system in which all messages are translated into a canonical message format on transmission and translated into the native host format upon reception. Although this protocol allows redundant translations, it is simple to implement and only requires that workers only know two data formats: the canonical one and their native one. To enable the communication of user-defined structures, we present a mechanism that uses signature tables to guide the translation.

In running Cilk programs in an environment in which workers may have different representations of data, the problem arises of establishing a coherent protocol for communication among workers. We therefore decided to solve this problem practically, by implementing a version of the Cilk runtime system that runs on the MIT Phish network of workstations. After a careful analysis of the "language barrier" problem, we were able to produce a practical solution and implement it in this version of Cilk.

The main problem with a system in which workers represent data differently is that more support is necessary for these workers to communicate. For example, one instance of this "language barrier" is the famous Endian byte ordering problem [4]. There is one school of thought, Big Endian, which believes that bytes should be ordered by placing the most significant byte as the leftmost byte. There is another school, Little Endian, which believes in the opposite: that the least significant byte should be leftmost.

This difference is shown in figure 1. When two machines of different Endian communicate, some translation is needed so that the workers interpret and understand each other's messages correctly.

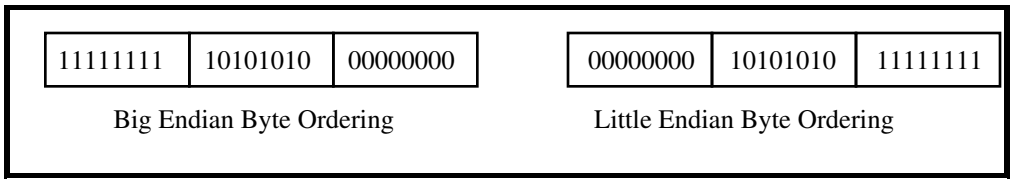

Figure 1: The different representations of the binary number 111111111010101000000000 in both the Big and Little Endian Byte Orderings.

The first issue we encountered is deciding how to implement this mechanism of translation. Two solutions immediately suggested themselves:

Mechanism 1: Upon sending a message, the sender must include an extra field in the message, stored in some canonical format, denoting in which data format the rest of the message is stored. Upon reception, the recipient then examines the extra field of the message. If the recipient understands the data format, it simply uses the message. Otherwise, it translates it.

The benefit of this scheme is that no unnecessary translations need be performed. The main disadvantage is that it must send a longer message, since an extra field must be maintained that denotes what data format the message is in. Moreover, each worker must know how to translate from any possible data format into the native data format.

Mechanism 2: Upon sending a message, the sender translates the entire message into some canonical data format. Upon reception, the recipient translates the message from the canonical format into its native data format.

The benefits of this mechanism are that it ensures that messages are as small as possible, since no extra data fields need be maintained, and that it provides a level of abstraction between the workers and the message-passing layer. Workers only need to know how to handle two data formats, the canonical message format and their native format. Unfortunately, Mechanism 2 always performs translation of the message, even when the sender and recipient possess the same data format.

Choosing between Mechanism 1 and Mechanism 2 yields a tradeoff: fewer translations versus smaller message sizes and fewer number of data formats each worker

must know. In the Phish network of workstations, the number of translations does not make too much difference. The overhead of translating a message is negligible in comparison to the communication cost of sending it. Since unnecessary translations do not noticeably affect performance, we chose to implement Mechanism 2 so that workers only need to know how to translate between their own native format and the canonical message format.

After examining the benefits and disadvantages of both solutions, we implemented Mechanism 2 in the Cilk runtime system running on the Phish network, because it is simple, transmitted messages are as short as possible, and the actual overhead of translation is negligible. If the cost of translation becomes significant however, because either the messages are larger or the communication costs decrease, we would choose to implement Mechanism 1 because of its efficiency, particularly if there are only a few possible data formats.

The second issue of heterogeneity of data representation we encountered was deciding how to actually implement the translation. We conceived of three methods: memoized translation on demand, no-memo translation on demand, and translation on reception. We describe these three methods:

Memoized Translation on Demand:

Upon receiving the message, the recipient does not actually translate any of the fields of the message until the values contained in those fields are needed by the worker's execution. When the value is requested, the worker first checks to see if the field has already been translated. If it has, it gets the value; otherwise, it translates the value and then uses it.

The benefit of this method is that unnecessary translations of data fields are never performed. Unfortunately, this method incurs the overhead of keeping track of which fields of the message are still in the canonical message format and which fields are in the native format. Furthermore, this mechanism does not easily work if data formats have different sizes, because it translates the data fields of a message in place.

No-memo Translation on Demand:

Upon receiving the message, the recipient does not translate any of the fields of the message until the values contained in those fields are needed. When requested, the field is translated and provided.

The benefit of this method is that a field that is never used by the worker is never translated. This method can unnecessarily translate the same field multiple times, however.

Translation on Reception:

Upon receiving the message, the entire message is translated from the canonical format into the native host data format.

This method is simple, because no information needs to be kept track of. The contents of the message are always in the same data format. Another benefit with the translation on reception method is that, because of its simplicity, it is straightforward to implement. The only code that needs to be added to the runtime system is that upon receiving a message, a translation procedure is invoked on the message. The disadvantage of this method is that it performs unnecessary translations.

Choosing between the three methods for translation again yields a trade-off: the overhead of maintaining information and the complexity of translation-on-demand versus the simplicity and unnecessary translations of the translation on reception method. As stated earlier, since the translation costs are negligible with respect to the actual cost of communication in the Phish system, unnecessary translations do not significantly degrade performance. Consequently, we chose to implement a translation on reception method. But once again, if communication costs are sufficiently small such that the overhead of translation becomes significant, one would switch to the no-memo translation on demand method. And, if maintaining information about which fields are in which format can be done cheaply, one would switch to the memoized translation on demand method.

The third issue of heterogeneity that we encountered is the problem that our translation mechanism requires that the sender and recipient of the message know the structure of the message. But in Cilk, it is possible to send data whose structure depends on the execution of the user's code. To support the passing of custom data structures as messages, we extended our translation mechanism to use signature tables to look up the structure of all possible messages.

The ability to send custom data structures is vital to the Cilk runtime system. Cilk contains two types of communication: continuations and work-stealing. Synchronization and the completion of work is communicated between threads by sending a simple and known type of data from one thread to the other through a function called a continuation. This continuation mechanism is simple to implement, because the type of the data is known by both threads. The other type of communication, work-stealing, is achieved by

having the thief receive a data structure called a closure from the victim. The actual structure of the closure depends on the program that is being executed. Therefore, without the ability to send custom data structures, Cilk would not be able to support its work-stealing algorithm for scheduling threads.

We solve the problem of passing custom data structures as data messages by constructing signature tables. Signature tables contain all the information detailing the structure of all closures that can be generated in the execution of the user's code. Because the user's code must be processed by a Cilk preprocessor before runtime anyway, we require that the preprocessor emits these signature tables into the user's code. When a closure is passed, its structure can be looked up in the signature tables and translated appropriately, both on the sending and receiving side.

Another possible solution to this problem would have been to burden the user to provide procedures to handle the packing and unpacking of the data structures. We did not select this method, because we do not want to expose the need for translation or the different data formats to the user.

The final issue of heterogeneity we encountered is the problem that heterogeneous workers possess different memory maps. Consequently, a memory reference to a function in one worker's memory will be nonsense in the other worker's memory. This instance of a "language barrier" again requires some mechanism for translation to allow coherent communication between workers. Since the memory references to functions are meaningless across different workers, we construct a signature table of all functions. Memory references to functions can be translated into indices into this signature table. When a memory reference is sent in a message, it is translated into the appropriate index into the table. Upon reception, the index is translated into a memory reference in the recipient's memory.

We implemented this solution in the Cilk runtime system to support active messages [5]. An active message is essentially a data message and a memory reference to a piece of code, called a handler procedure. Upon reception of the message, the recipient invokes the handler procedure on the rest of the message. This handler procedure handles the unpacking of the message into appropriate data structures. Cilk uses these active messages to handle all of the communication between the workers. To support active messages in our implementation of Cilk, we generated a signature table of all handler procedures used by the Cilk runtime system. When an active message was sent with a reference to one of these procedures, the reference was translated into an index into the table. Upon reception, the index was translated into a memory reference to a handler procedure, which is then invoked on the rest of the message.

In the current implementation of Cilk, our solution does not support the passing of references of user-defined functions in a message. The Cilk preprocessor cannot generate a signature table for all user-defined functions, because Cilk code can be linked with standard C code. The Cilk preprocessor only processes Cilk code, so it only generates a table for the functions in the Cilk code and not the linked C code. Our solution of using signature tables does not work because of this limitation of the current Cilk preprocessor. If we later produce a Cilk linker or require the Cilk preprocessor to process all of the codes, our signature table solution would support the passing of user-defined functions.

To further demonstrate the fact that heterogeneity is not a significant problem to multithreaded computing, we constructed an implementation of Cilk built on top of the PVM message-passing layer. The PVM layer allows us to run Cilk on a variety of workers that can differ with data format, in speed, and in scale. By employing our translation mechanism and using signature tables, this version of Cilk is able to run heterogeneous multithreaded computing without a significant degradation in performance.

4 Heterogeneity in Speed

In this section, we explore the issues introduced by extending Cilk to run on a set of workers that are heterogeneous with respect to the speed at which they work. If all the workers communicate at the same constant rate (as in the case when they all communicate through a network), we demonstrate that the Cilk runtime system is able to handle the differences in worker speed in a fair manner. We derive theoretical bounds on the space, time, and communication costs of a fully strict Cilk program running in such a heterogeneous environment. In an environment of P workers, a fully strict Cilk program with total work T_1 and a critical path of length T_∞ uses space $S_P \le S_1 P$ and incurs $O(S_{max} P T_\infty)$ communication cost, where S_1 is the maximum space used in a single worker execution and S_{max} is the size of the largest data structure communicated. The execution time of such a program is

$$T_P = O(T_1/(\pi_{ave} P) + \sigma T_\infty \pi_{max}/\pi_{ave}),$$

where π_{max} is the performance of the fastest worker and π_{ave} is the average performance over the worker pool. We conclude that heterogeneity of speed only affects the time of execution: the space and communication costs are within a constant factor of the homogeneous case. Furthermore, the computation is still efficient: all of the computation performance $\pi_{ave} P$ is being applied to the total work T_1. We define the "average parallelism" of a computation to be T_1/T_∞, the "speedup" of the computation to be T_1/T_P, and the "effective parallelism" of the computation to be the maximum number of workers

allowable to still guarantee linear speedup (i.e. the speedup T_1/T_P grows linearly with P). We show that the effective parallelism of a Cilk program running in a heterogeneous environment with workers of differing performances is

$$\frac{T_1}{T_\infty}\frac{1}{\sigma\pi_{\max}}.$$

Recalling that the effective parallelism in a homogeneous environment is T_1/T_∞, heterogeneity in speed only affects the effective parallelism by a constant factor, $1/(\sigma\pi_{max})$.

Because the Cilk scheduler provides a view of multithreaded computations as directed acyclic graphs (dags) of tasks, Blumofe and Leiserson [2] were able to quantify the space requirements, execution time, and communication cost of a fully strict Cilk program running across P homogeneous processors in terms of measurements of this dag. We use these same measurements to quantify space, time, and communication costs in a heterogeneous environment, where workers possess different work performances.

Following the same argument of Blumofe and Leiserson, we derive that the space S_P used by the execution of a fully strict Cilk program running over P heterogeneous workers is bounded by S_1P, where S_1 is the maximum of the activation depths of any of the workers, running by itself. Running strict Cilk programs with workers of differing performances produces the same space bounds $S_P \le S_1P$ as the program running in the homogeneous environment.

We also show that for fully strict programs running on P workers with different performances, the expected total number of bytes communicated is $O(PT_\infty S_{max})$, where T_∞ is the length of the critical path measured in instructions and S_{max} is the size of the largest task in the dag. This proof follows directly from the proof in the homogeneous case [2]. Therefore, running Cilk programs with workers of differing performances does not affect the communication costs.

To summarize, allowing workers to possess different performances does not affect the theoretical bounds on the space and communication of the execution of a multithreaded Cilk program. The proofs follow directly from the homogeneous case. The bound on the execution time of the program is somewhat different, however.

We introduce terminology in order to allow us to quantify the execution time of a Cilk program. The quantity we wish to analyze is the time used in executing the program in an environment of P workers with differing performances. We let this time be T_P. We measure the length T_∞ of the critical path of the computation in terms of instructions. Likewise, we also measure the total work T_1 of the computation, in instructions. The total work T_1 reflects the minimum number of instructions that need to be executed in

order to perform the computation on a single worker. We measure the performance π_i of the ith worker in terms of the rate in which it can execute instructions (e.g., instructions per second). We also assume that the time σ to perform a steal operation is uniform across all of the workers.

Similar to the work of Blumofe and Leiserson [2], we use an accounting argument to analyze the running time of the Cilk work-stealing scheduling algorithm. We assume that a fully strict multithreaded computation with work T_1 and critical path length T_∞ is executing in a heterogeneous environment of P workers. In their argument, Blumofe and Leiserson posit that at each time step, one dollar is collected from each processor. These P total dollars are then placed in one of three buckets: *Work*, *Steal*, or *Wait*. Blumofe and Leiserson were able to bound the expected number of dollars that are in each of the three bins when the execution of a fully strict Cilk program terminates: the *Work* bin contains T_1 dollars, the *Steal* bin contains $O(PT_\infty)$ dollars, and the *Wait* bin contains $O(PT_\infty)$ dollars. Therefore, the expectation of the total dollars collected in the execution of a fully strict Cilk program is $O(T_1 + PT_\infty)$. The execution time T_P of the program is the total number of dollars collected divided by the number of dollars collected per time step. Blumofe and Leiserson therefore concluded that: $T_P = O(T_1/P + T_\infty)$.

Unfortunately, this argument does not directly extend to the heterogeneous case, because it does not account for the fact that workers can do unequal amounts of work in the same time due to performance differences.

We therefore construct an accounting argument with P+2 buckets: *Work*, *Wait*, and $Steal_1$, $Steal_2$,...,$Steal_P$. For each second of execution, for all i=1,...,P, we collect π_i dollars from the ith worker, where π_i is the performance of the ith worker. We let π_{ave} be the average performance of the set of workers

$$\pi_{ave} = \frac{1}{P}\sum_{i=1}^{P}\pi_i .$$

The total number of dollars collected each second is therefore $\pi_{ave}P$. Depending on its actions for that second, for all i=1,...,P, the ith worker must throw its dollars into one of three buckets: *Work*, $Steal_i$, or *Wait*. If the worker executes instructions during that second, it places its π_i dollars in *Work*. If the worker initiates a steal attempt, it places its π_i dollars in $Steal_i$. And if the worker waits for a queued steal request, it places its π_i dollars in the *Wait* bucket. We can therefore bound the time of the entire execution by simply summing the total number of dollars in all of the buckets at the end of the execution, and then dividing by $\pi_{ave}P$, the number of dollars collected per second.

Lemma 1 *The execution of a fully strict Cilk program with work T_1 executing on a set of P workers, varying in performances, terminates with exactly T_1 dollars in the Work bucket.*

Proof: The *i*th worker places its π_i dollars in the *Work* bucket when it executes instructions. Since there are T_1 instructions in the entire computation, the execution ends with exactly T_1 dollars in the *Work* bucket. ♦

Determining a bound on the total dollars in the P steal buckets requires a more in-depth analysis than the bound on the *Work* bucket, because the dollars in the *i*th steal bucket $Steal_i$ represent the number of instructions that are "wasted" by the *i*th worker when it is trying to steal. Although the workers steal at the same rate, $1/\sigma$ steals per second, the amount of work "wasted" is different, because they have different performances. We are therefore unable to directly place a bound on the total number of instructions "wasted" while stealing.

We place a bound on the total number of steals that are performed in a computation instead. It has been shown that in the Cilk work-stealing algorithm, with probability at least 1-ε, the execution terminates with at most $O(P(T_\infty + \lg(1/\varepsilon)))$ steals performed [2]. So, the expected number of steals is $O(PT_\infty)$. Let n_i denote the number of steals performed by the *i*th worker throughout the computation. Therefore, we have

$$\Sigma_{i=1}^{P} n_i = O(P(T_\infty + \lg(1/\varepsilon))).$$

We can bound the number of dollars in each steal bucket, which is the number of instructions "wasted" stealing by the corresponding worker.

Lemma 2 *For i=1,...,P, after the execution of a fully strict Cilk program, there are $\sigma\pi_i n_i$ dollars in the $Steal_i$ bucket, where σ is the time in seconds needed to perform one steal and n_i is the number of steals performed by the ith worker, which can execute π_i instructions per second.*

Proof: If the *i*th worker performs n_i steals throughout the duration of the computation, that means that the worker has spent σn_i seconds stealing throughout the execution. So, for σn_i seconds of computation, the *i*th worker has been performing steals. Therefore, the worker has been placing its π_i dollars into the $Steal_i$ bucket σn_i times. We conclude that after the program has completed its execution, there are $\sigma\pi_i n_i$ dollars in the $Steal_i$ bucket. ♦

Placing a bound on the number of dollars in the *Wait* bucket follows directly from the contention analysis of Blumofe and Leiserson [2]. According to Blumofe and Leiserson, the total delay incurred by M random requests made by P workers is $O(M + P\lg P + P\lg(1/\varepsilon))$ with probability at least $1-\varepsilon$, and the expected delay is at most $O(M)$. Since we have shown that the number of steals performed in the execution of a fully strict Cilk program running on P workers with varying bandwidths is $O(P(T_\infty + \lg(1/\varepsilon)))$ with probability at least $1-\varepsilon$, the expected total delay incurred in the execution of the program is $O(PT_\infty + P\lg P + P\lg(1/\varepsilon))$. With probability at least $1-\varepsilon$, at most $O(PT_\infty + P\lg P + P\lg(1/\varepsilon))$ dollars are in the *Wait* bucket, at the termination of the program.

Lemma 3 *For any fully strict Cilk program running on P workers, where for all $i=1,\ldots,P$, the ith worker has performance π_i, with probability at least $1-\varepsilon$, a total of*

$$T_1 + \sigma\Sigma_{i=1}^{P}\pi_i n_i + O(PT_\infty + P\lg P + P\lg(1/\varepsilon))$$

dollars are accumulated in all $P+2$ buckets, where $\Sigma_{i=1}^{P} n_i = O(PT_\infty + P\lg(1/\varepsilon))$. The total expected dollars accumulated in all of the bins is $T_1 + \sigma\Sigma_{i=1}^{P}\pi_i n_i + O(PT_\infty)$, where $\mathrm{Exp}(\Sigma_{i=1}^{P} n_i) = O(PT_\infty)$.

Proof: From lemma 1, there are T_1 dollars in the *Work* bucket at the end of the execution. There are, with probability at least $1-\varepsilon$, at most $O(PT_\infty + P\lg P + P\lg(1/\varepsilon))$ dollars in the *Wait* bucket. And from lemma 2, there are $\sigma\pi_i n_i$ dollars in each $Steal_i$ bucket. Summing and reducing yields $T_1 + \sigma\Sigma_{i=1}^{P}\pi_i n_i + O(PT_\infty + P\lg P + P\lg(1/\varepsilon))$ total dollars. The expected bound follows, because the tail of the distribution falls off exponentially. ♦

As stated earlier, the total time of execution is bounded by the number of dollars collected from all the buckets divided by the number of dollars placed in the buckets per second. Since there are $\pi_{ave}P$ dollars collected per second, the expected running time is

$$T_p = O\left(\frac{T_1}{\pi_{ave}P} + \frac{\sigma}{\pi_{ave}P}\sum_{i=1}^{P}\pi_i n_i + \frac{PT_\infty}{\pi_{ave}P}\right).$$

Theorem 1: *The total expected running time of the execution of a fully strict Cilk program running on P workers, where the ith worker runs with a performance of π_i instructions per second, is*

$$T_p = O\left(\frac{T_1}{\pi_{ave}P} + \sigma T_\infty\frac{\pi_{\max}}{\pi_{ave}}\right),$$

where π_{max} is the performance of the fastest worker and π_{ave} is the average performance of the P workers.

Proof: Since for all i=1,..,P, the ith worker has performance $\pi_i \le \pi_{max}$, the expected running time is

$$O(T_1/(\pi_{ave}P) + \sigma\sum_{i=1}^{P}(\pi_i n_i/(\pi_{ave}P)) + PT_\infty/(\pi_{ave}P))$$
$$\le O(T_1/(\pi_{ave}P) + \sigma\sum_{i=1}^{P}(\pi_{\max} n_i/(\pi_{ave}P)) + PT_\infty/(\pi_{ave}P)).$$

From lemma 3, the expectation of $\sum_{i=1}^{P} n_i$ is O(PT_∞). Substituting and reducing yields the theorem. ♦

According to our analytical time bound, having workers of different speeds running a Cilk program only affects the amount of time spent on the critical path. All of the workers, fast and slow, are contributing to the reduction of the total work T_1. The only penalty we pay for having slow workers is the factor of π_{max}/π_{ave} being spent working along the critical path of length T_∞. We can also view this penalty factor as $\pi_{max}P/\sum_{i=1}^{P}\pi_i$, from the definition of π_{ave}. If we view this summation in a continuous setting, we note that the penalty we pay is the ratio of $\pi_{max}P$ to the integral of π_i, from 1 to P. We show this in figure 2. We can view the penalty we pay for heterogeneity to simply be the ratio of the shaded area to the total area. Therefore the closer in speed the workers are, the less the penalty.

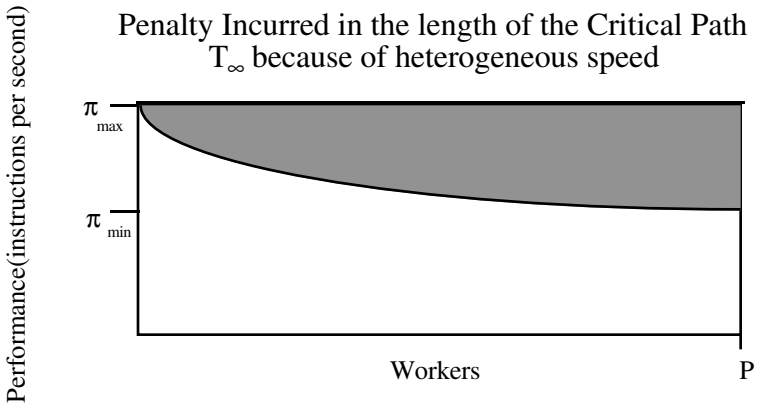

Figure 2: The penalty in the reduction of the critical path for having workers which have different bandwidths is the ratio of the area of the rectangle to the area of the white region. The ratio of the shaded area to the total area of the rectangle reflects the penalty for having heterogeneity in the computation.

We can produce more informative bounds on the expected running time of a fully strict Cilk program if we have more information about the heterogeneous environment. If we know that the performance of the fastest worker π_{max} is at most a factor of α more than the performance of the slowest worker π_{min}, i.e. $\pi_{max} \leq \alpha\pi_{min}$, then the expected running time of a fully strict Cilk program is $T_P = O(T_1/(\pi_{ave}P) + \sigma\alpha T_\infty)$ in this environment. We can derive this quite simply from the fact $\pi_{ave} \geq \pi_{min}$. The penalty of having heterogeneous workers is only in the time it takes to perform work along the critical path, and that penalty is at most the ratio of the fastest worker to the slowest worker.

We define T_1/T_∞ to be the "average parallelism" of a computation, T_1/T_P to be the speedup of the computation, and the maximum number of workers allowable to still guarantee linear speedup (i.e. the speedup T_1/T_P grows linearly with P) to be the effective parallelism.

Theorem 2: *The effective parallelism of a fully strict Cilk program running on P workers with total work T_1 and critical path of length T_∞ is*

$$\frac{T_1}{T_\infty}\frac{1}{\sigma\pi_{\max}},$$

where π_{max} is the performance of the fastest worker and σ is the time to perform a steal operation.

Proof: The effective parallelism of a computation is defined to be the maximum number of workers that allows linear speedup. From theorem 1, the bound on the execution time is $T_P = O(T_1/(\pi_{ave}P) + \sigma T_\infty\pi_{max}/\pi_{ave})$. Linear speedup can only be achieved when $T_P = O(T_1/(\pi_{ave}P))$. Therefore, we only obtain linear speedup when

$$T_1/(\pi_{ave}P) \geq \sigma T_\infty\pi_{max}/\pi_{ave}.$$

Performing some simple reductions, we obtain $P \leq (T_1/T_\infty)(1/(\sigma\pi_{max}))$. ♦

From our time bound and the derived expression for the effective parallelism, we conclude that Cilk programs running in a heterogeneous environment are efficient: all of the performance $\pi_{ave}P$ is applied to the total work T_1. The only penalty we incur for having workers with differing performances is that the effective parallelism is a factor of $\sigma\pi_{\max}$ smaller than the average parallelism T_1/T_∞.

5 Heterogeneity in Scale

In this section, we investigate the issues involved in extending Cilk to run on a set of workers composed of computers of different scales, such as workstations and multiprocessors. We provide theoretical bounds on the time, space, and communication costs on the execution of a fully strict Cilk program in such an environment. The bounds on the space and communication costs are the same as the homogeneous case. We show that the execution time of the program is $T_P = O(T_1/(\pi_{ave}P) + \sigma_{max}T_\infty\pi_{max}/\pi_{ave})$. We also conclude that attempting to maximize local communications, such as communication between nodes of a multiprocessor, may be a good heuristic, but should not be a rule for the Cilk work-stealing algorithm.

The primary difference between an environment in which workers are heterogeneous in speeds and an environment where the workers can differ in scale is that the communication time between workers is no longer constant. For example, a node in a multiprocessor communicates faster with another node than with a workstation on the network. For the execution of a fully strict Cilk program, these differences in communication times do not affect our bounds for space and communication costs, only for the execution time. And, our derivation of a bound for the execution time extends well for nonconstant communication.

In a completely heterogeneous environment, in which workers differ in their speed and scale, the expected execution time of a fully strict Cilk program is $T_P=O(T_1/(\pi_{ave}P) + \sigma_{max}T_\infty\pi_{max}/\pi_{ave})$, where π_{ave} is the average performance of P workers, π_{max} is the performance of the fastest worker, and σ_{max} is the longest time, measured in seconds, for one worker to steal from another. The proof of this follows directly from our derivation with constant communication, if we simply treat σ to be a function, rather than a constant. From this bound, we obtain an effective parallelism of $(T_1/T_\infty)(1/(\sigma_{max}\pi_{max}))$.

Because of our time bound, integrating workers of different speeds and scale only affects the amount of time it takes to reduce the critical path by a factor of $\sigma_{max}\pi_{max}/\pi_{ave}$. All of the computer resources are being utilized to reduce the total work T_1.

One downside of incorporating workers of different scales is that the time spent working on the critical path is increased by the time σ_{max} of the slowest communication. Maximizing local communication whenever possible is one way to reduce this factor. The current Cilk work-stealing algorithm states that if a worker is idle however, it attempts to steal from another worker chosen at random. Maximizing local communication requires the work-stealing algorithm to try to steal from a local worker, as opposed to a global worker. Maximizing local communication does not guarantee,

however, that if workers are stealing, progress is being made on the critical path. Intuitively, if workers are busy, they are contributing to reducing the total work T_1. If sufficient workers are unsuccessfully stealing, we are confident that progress is being made on the critical path. After P random steals, with reasonable probability, we know that work has been made on the critical path. Maximizing local communication defeats this random choice of stealing, so that even after P steals, we have no guarantee that progress has been made along the critical path. Consequently, the execution time could be even worse, because we can have workers who are not contributing to the reduction of the total work or to progress along the critical path. We conclude that the naive idea of trying to maximize communication may be a good heuristic for choosing a worker from which to steal, but it should not be implemented as a rule, since it could actually degrade performance.

The effective parallelism of a fully strict Cilk program running in a completely heterogeneous environment in which workers differ in performance and scale is

$$\frac{T_1}{T_\infty} \frac{1}{\sigma_{\max} \pi_{\max}}.$$

The proof of this follows directly from our derivation with constant communication, if we simply treat σ to be a function, rather than a constant. Therefore, the only penalty for running a program in a completely heterogeneous environment, instead of a homogeneous one, is a factor of $\sigma_{\max}\pi_{\max}$ decrease in the effective parallelism. Our derived time bound shows that all of the computational performance is applied to the total work, so the computation is efficient.

We were able to show the practicality of implementing Cilk in an environment of workers of different scale by implementing a version of Cilk on the PVM message-passing layer. PVM enables the construction of a worker pool consisting of workers of different scales. Since we constructed Cilk on top of this layer, we are able to run Cilk programs across such a worker pool.

6 Implementation

In this section, we present some of the details and problems that were encountered when we modified the implementation of Cilk that ran on the MIT Phish network of workstations [3] and the implementation on the PVM message-passing layer [6].

The major implementation difficulty encountered was that the mechanism for starting up a Cilk program was no longer valid. Originally, a Cilk program was executed by simply running the program simultaneously across the nodes of a multiprocessor. But in a network of heterogeneous workers, a program cannot be started simultaneously on all

the workers. Instead, we modified the startup of a Cilk program so that one worker was the host for the program. After it starts, it spawns itself off onto the rest of the workers.

The implementation of Cilk on the MIT Phish network of workstations was limited because it could only run on a network of heterogeneous workstations. In order to construct a version of Cilk that could run on a completely heterogeneous environment, in which workers could differ in data format, speed, or scale, we chose to implement Cilk on the PVM message-passing layer because PVM was available on most platforms. Furthermore, PVM also provides a mechanism for running programs on machines of different scales and speeds [6]. Because of this mechanism, we were able to concentrate our efforts on building applications for Cilk and doing theoretical analysis.

7 Conclusion

We have shown that it is practical to extend the Cilk runtime system from running on a homogeneous environment to running in a heterogeneous one. We have also proven analytically that running Cilk on a heterogeneous environment is efficient: all of the computational performance is applied to the total work of a program. The only penalty incurred is a slight decrease in the effective parallelism.

From a practical standpoint, we have proposed implementations of Cilk that run on heterogeneous environments. We explained our mechanism for translation so that two workers of different data formats could communicate with each other. We have also constructed two implementations of Cilk, one running on the MIT Phish network of workstations, the other on the PVM message-passing layer, which shows that it is possible to extend Cilk to heterogeneous environments. We have also tested these implementations by running several programs on them: recursive Fibonacci, the nqueens problem (placing n queens on a nxn chessboard so that no two queens can attack each other), a ray-tracing algorithm, and the *Socrates chess program.

From an analytical view, we have shown that in a heterogeneous environment in which the communication rate is constant across all workers (as is the case when all workers communicate through a network), the execution time of a Cilk program running on P workers with total work T_1 and a critical path of length T_∞ is $T_P = O(T_1/(\pi_{ave}\ P) + \sigma T_\infty \pi_{max}/\pi_{ave})$ where σ is the time it takes to perform a steal, π_{max} is the average performance of the workers, and π_{ave} is the performance of the fastest worker. In an environment where workers can differ in scale as well, the execution time is $T_P = O(T_1/\pi_{ave}P + \sigma_{max}T_\infty \pi_{max}/\pi_{ave})$ where σ_{max} is the maximum time it takes for one worker to steal from another. Finally, we showed that the effective parallelism in such a

computation is $\frac{T_1}{T_\infty}\frac{1}{\sigma_{\max}\pi_{\max}}$. Therefore, the effective parallelism in a heterogeneous environment is a factor of $\sigma_{max}\pi_{max}$ smaller than the effective parallelism T_1/T_∞ in a homogeneous environment.

Acknowledgments

I would like to thank all of the members of the Cilk project at the MIT Laboratory for Computer Science. Their help, advice, and insights have provided me with a great deal of support. I would especially like to thank Mr. Robert D. Blumofe and Professor Charles E. Leiserson for all of their long hours of support.

References

[1] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. To appear in the *Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, California, July 1994.

[2] Robert D. Blumofe and Charles E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. In *Conferences in Foundations of Computer Science*, Santa Fe, New Mexico, November 1994. IEEE.

[3] Robert D. Blumofe and David S. Park. Scheduling Large-Scale Parallel Computations on Network of Workstations. In *Proceedings of the Third International Symposium on High Performance Distributed Computing*, San Francisco, California, August 1994.

[4] D. Cohen. "On Holy Wars and a Plea for Peace". *Computer*, Vol. 14, No. 10, October 1981, pp. 48-54.

[5] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, Klaus Erik Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992. ACM.

[6] V.S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315-339, December 1990.