6.895 Theory of Parallel Systems Kunal Agrawal, Siddhartha Sen

Adaptively Parallel Processor Allocation for Cilk Jobs

Final Report

Abstract

An adaptively parallel job is one in which the number of processors that can be used without waste varies during execution. This paper presents the design and implementation of a dynamic processor-allocation system for adaptively parallel jobs. We focus on the specific case of parallel jobs that are scheduled with the randomized work-stealing algorithm, as is used in the Cilk multithreaded language. The jobs are run on a shared-memory symmetric multiprocessor (SMP) system.

We begin by presenting a simple heuristic for estimating the processor desire of a Cilk job, based on the number of incomplete Cilk procedure frames queued on each of the job's processors. Then, we show how these estimates can be used to allocate processors to multiple running Cilk jobs in a fair and efficient way.

We present empirical evidence verifying the correctness of our dynamic processor-allocation system and evaluating its overhead.

1 Introduction

The problem of allocating processor resources fairly and efficiently to parallel jobs has been studied extensively in the past. Most of this work, however, assumes that the instantaneous parallelism of the jobs is known and used by the job scheduler to make its decisions. In practice, this information is not easily discernible, which is why many systems resort to static algorithms when deciding how to allocate processors to jobs. The current release of Cilk (Cilk-5.3.2), for example, expects the user to manually enter the parallelism of his job in the command line itself.

In this paper, we present a simple algorithm for estimating the instantaneous parallelism of a job, and show how this information can be used to dynamically allocate processors to jobs in a fair and efficient manner.

We define an *adaptively parallel job* to be a job for which the number of processors that can be used without waste—representing the instantaneous parallelism of the job—varies during execution. Our goal is to design and implement a dynamic processor-allocation system for adaptively parallel jobs. We call the problem of allocating processors to adaptively parallel jobs the *adaptively parallel processor-allocation problem* [6].

This paper investigates the adaptively parallel processor-allocation problem for the specific case of parallel jobs that are scheduled with the randomized work-stealing algorithm, as is used in the Cilk multithreaded language [5, 1, 4]. We present the design of a processor allocation system that allocates processors fairly and efficiently to multiple Cilk jobs running on a shared-memory symmetric multiprocessor (SMP) system. We define the terms "fair" and "efficient", and other relevant terms, in Section 2 below. The design of our system can be divided into three primary tasks:

1. Dynamically estimate the number of processors desired by each Cilk job on the system; this is our measure of the instantaneous parallelism of each job.

2. Dynamically determine the number of processors that should be allocated to each job such that the resulting allocation is fair and efficient.

3. Dynamically add or remove processors from a job if the number of processors granted in task 2 is less than or greater than the current allotment, respectively.

The notion of "dynamic" here varies from task to task; an important part of our design is to determine when and how frequently the above tasks should be performed. We answer these questions in later sections.

The remainder of this paper is organized as follows. In Section 2, we define the relevant terms of the adaptively parallel processor-allocation problem and state our assumptions. Section 3 gives an overview of some of the relevant previous work done on the problem, and Section 4 gives a brief overview of the Cilk runtime system. In Section 5, we present the design of our dynamic processor-allocation system, based on the three tasks listed above. In Sections 6 and 7, we describe the prototype implementation of our system and summarize the experimental results that we obtained. We present possible extensions to the prototype in Section 8, and we conclude in Section 9.

2 Definitions and Assumptions

In this section, we define the relevant terms of the adaptively parallel processor-allocation problem and state our assumptions. Section 4 defines the terminology that is specific to Cilk jobs.

Fairness and Efficiency

We consider a shared-memory SMP system with P processors and J jobs. In general, we follow the terminology and conventions used in [6], but for a shared-memory system instead of a distributed one. At any given time, each job has a desire d_j, representing the maximum number of efficiently usable processors, and an allotment a_j, representing the number of processors allocated to it. If $a_j < d_j$, we say that the job is deprived, since it has a deprived allotment; if $a_j = d_j$, we say that the job is satisfied, since it's desire has been met. We introduce an additional term, p_j, to represent the number of processors currently being used by the job; ideally, $p_j = a_j$, but this statement may not always be true, as explained in our assumptions later on.

Our problem, as stated in Section Section 1, is to find a fair and efficient allocation of processors among all J jobs in the system. We define the terms "fair" and "efficient" identically to [6]. An allocation is said to be *fair* if whenever a job receives fewer processors than it desires, then no other job receives more than one more processor than this job received (the allowance of one processor is due to integer roundoff). Mathematically, an allocation is fair if the following condition holds:

1. $\exists j \in \{1, 2, \ldots, J\}$ such that $a_j < d_j \Longrightarrow \forall i \in \{1, 2, \ldots, J\}$, $m_i \leq a_j + 1$.

An allocation is *efficient* if no job receives more processors than it desires, and the maximum number of processors that can be used are being used. Mathematically, this definition translates to the following two conditions:

1. $\forall j \in \{1, 2, \ldots, J\}$, $a_j \leq d_j$.

2. $\exists j \in \{1, 2, \ldots, J\}$ such that $a_j < d_j \Longrightarrow \sum_{j=1}^{J} a_j = P$.

Unlike the definition of efficiency presented in [6], we do not allow the allotment of a job to exceed its desire, even if there are unused processors in the system. To see why condition 2 is necessary, consider an allocation where each of the J jobs receives 0 processors. In this scenario, no job has more processors than it desires—so condition 1 is satisfied—but none of the jobs can make any progress and all P processors are going to waste. Condition 2 prevents situations like this by ensuring that we always use as many processors as possible.

An example of a fair and efficient allocation, according to the above definitions, is shown in Figure 1. From the example, we observe that the deprived job with the smallest allotment restricts the maximum number of processors any other job can have—since, by the fairness condition, no job can have more than one more processor than this job.

Assumptions

We make several assumptions in our solution to the adaptively parallel processor-allocation problem. These assumptions are summarized in the list below:

1. All jobs on the system are Cilk jobs.

2. Jobs can enter and leave the system and change their parallelism during execution.

3. All jobs are mutually trusting, in that they will stay within the bounds of their allotments and communicate their desires honestly.

4. Each job has at least one processor.

5. Jobs have some amount of time to reach their allotments.

Assumption 5 is a consequence of task 3 from Section 1; in particular, since there is necessarily some overhead associated with adding or removing processors from a job, it may take some time before a job's processor usage (p_j) actually reaches its granted allotment (a_j). This delay is illustrated in Figure 2; it is also measured in one of the experiments performed in Section 7.

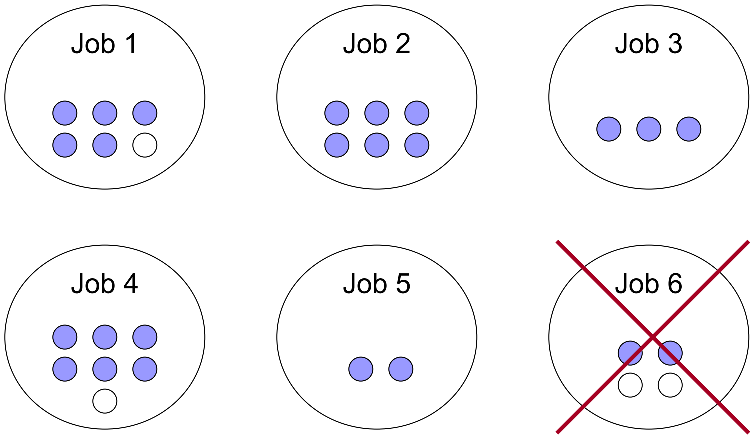

Figure 1: An example of a fair and efficient allocation. The small circles represent the desires of each job (shaded circles represent allocated processors; empty circles represent desired processors that have not been allocated). The deprived job with the smallest allotment, Job 1, restricts the maximum number of processors any other job can have (6 processors). Since Job 6 has a deprived allotment that is less than Job 1's, it cannot be a part of this allocation without violating the fairness condition.

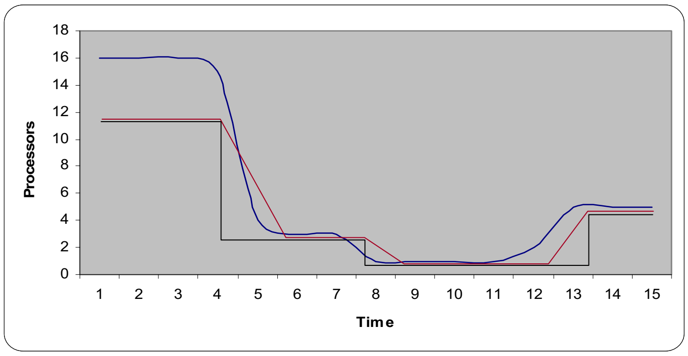

Figure 2: The processor desire (shown in blue), allotment (shown in black), and usage (shown in red) of an imaginary adaptively parallel job. The delayed response of the current usage (red line) to changes in the job's allotment (black line) are caused by the overhead of adding or removing processors during execution.

3 Previous Work

The literature on the adaptively parallel processor-allocation problem is quite extensive. We limit our discussion of previous work to the results that are pertinent to our definition of the problem, and which we draw from in designing our system.

In [6], Song presents a randomized processor-allocation algorithm, called the SRLBA algorithm, for the adaptively parallel procesor-allocation problem in a distributed system of P processors and J jobs. The SRLBA algorithm is a variant of the Randomized Load-Balancing (RLB) algorithm that consists of rounds of load-balancing steps in which processor migration (from one running job to another) may occur. A round consists of P sequential load-balancing steps, in which each processor is given a chance to initiate a load-balancing step. SRLBA differs from the RLB algorithm in two ways: 1) SRLBA operates in the *sequential perfect-information model*, which assumes that all load-balancing steps occur serially, and that all job allotments are updated promptly after each step; and 2) SRLBA introduces an adversary into the system who, at the beginning of each step i in a given round, having seen the previous $i-1$ steps, chooses a processor not previously chosen in the round to initiate the ith load-balancing step [6]. The main result of [6] states that if all J jobs have a desire greater than the *absolute average allotment* P/J of processors, then within $O(\lg P)$ rounds of the SRLBA algorithm, the system is in an *almost fair and efficient allocation* with high probability, meaning that every job has within 1 of P/J processors with high probability.

Blumofe, Leiserson, and Song develop a processor-allocation heuristic in [3] that uses the "steal rate" of adaptively parallel jobs to estimate their desires during runtime. The jobs considered are those whose threads are scheduled with the randomized work-stealing algorithm, as is used in Cilk. Blumofe et. al. implement their processor-allocation heuristic in a job scheduler for Cilk, called the Cilk Macroscheduler; the scheduler uses the steal rates of multiple, concurrent Cilk jobs to achieve a fair and efficient allocation among the jobs. The Macroscheduler is implemented in user space: it consists of a processor manager for each Cilk job as well as a job registry (in shared memory) through which the processor managers can communicate. The job registry contains the processor allotments and desires of every Cilk job; since each processor manager can view this information, it can easily calculate the number of processors its corresponding job should possess in a fair and efficient allocation, and adjust the processor allotment accordingly. The performance analysis of the Macroscheduler is incomplete; the only result provided in [3] is an empirical analysis of the scheduler's overhead, in which various Cilk programs are run on a multiprocessor machine two times, the first time with the original Cilk runtime system and the second time with the Macroscheduler. Each program run occurs in isolation—that is, the program is the only job running on the machine.

Although the work in this paper is largely influenced by [6] and [3] above, it is different in several key ways. Unlike the SRLBA algorithm, we are solving the adaptively parallel processor-allocation problem for a shared-memory multiprocessor system. Consequently, we eliminate the need for rounds of load-balancing steps between pairs of processors, since the jobs in our system have complete knowledge of the processor allotments and desires of all other jobs (via shared memory). In this respect, our processor allocation system for Cilk is similar to the Macroscheduler presented in [3]. Unlike the Macroscheduler, however, we use a completely different heuristic for estimating the processor desires of Cilk jobs. We argue that this heuristic is a more direct and timely measure of a job's parallelism, allowing the system to respond more quickly and effectively to changes in this parallelism.

Sections 5 and 6 explain the differences highlighted above in more detail.

4 The Cilk Runtime System

We provide a brief overview of the Cilk language and runtime system, and we define the terminology used in subsequent sections to describe Cilk jobs.

Cilk is a language for multithreaded parallel programming based on ANSI C that is very effective for exploiting highly asynchronous parallelism. Cilk differs from other multithreaded programming systems by being algorithmic in nature—that is, it employs a scheduler in its runtime system that guarantees provably efficient and predictable program execution performance [5]. The current release of Cilk is designed to run efficiently on shared-memory SMPs.

When a Cilk program is run, the user has the option of specifying the maximum number of "processors" or *workers* that the Cilk job may attempt to use (if no number is specified, a default value is used). The runtime system then creates this many workers (implemented using operating system threads or processes) and schedules the user computation on the workers using a randomized, work-stealing algorithm [2, 4]. The work-stealing algorithm is simple: during program execution, if a worker runs out of work to do, it chooses another worker at random (called the *victim*) and tries to *steal* work from the victim. If the steal is successful, the worker begins working on the stolen piece of work; otherwise, it picks another worker (uniformly at random) to be the victim and continues work-stealing until the steal is successful. Cilk relies on the operating system to schedule the workers onto the physical processors of the SMP.

In the remainder of this paper, the term "processor" is used analogously with "worker", and is distinguished from the term "physical processor", which refers to an actual processor on the SMP. As we mentioned above, when a Cilk program is run, the user is expected to tell the runtime system how many processors to run the job on. In other words, the user must provide a static (one-time) estimate of the job's parallelism, or else a default number of processors is used. As we discussed in Section 1, it is difficult to measure the instantaneous parallelism of a job, let alone provide a one-time estimate for it, since a job's parallelism can vary significantly during its execution. Thus, while a static allocation scheme might be simpler to implement in the Cilk runtime system, it does not guarantee efficient use of the available processors. This inadequacy of Cilk is the primary motivation behind the processor allocator system presented in this paper.

Terminology for Cilk Jobs

A Cilk program execution consists of a collection of *procedures*, each of which is broken into a sequence of nonblocking *threads* [5]. Each processor of a Cilk job maintains a *deque*, or doubly-ended queue, of *frames* to keep track of the current scheduling state. In Cilk terminology, a *frame* is a data structure that can hold the state of a procedure—including, among other things, the thread of the procedure that is ready to execute next. Within a given processor, execution always occurs at the bottom of the frame deque: procedure state is saved by pushing a frame onto the bottom of the deque, and procedures are executed by popping the bottommost frame on the deque. When a processor work-steals, it takes the topmost frame (the least recently pushed frame) from the victim's deque and places it in its own deque [5].

As we show in Section 5, the size and contents of the deques of a Cilk job provide direct information on the amount of pending work the job has accumulated so far; this information can help us estimate the number of processors that the job can most efficiently use.

5 Design Overview

At any instant, there may be many independent jobs in the system. Each of these jobs operates according to the assumptions in Section 2. There is a global allocator which has all the information about the desires and allotments of each job and it is the responsibility of this allocator to keep the allocation fair and efficient.

As mentioned in Section 1, there are three major tasks in the processor allocation system, shown in more detail steps in Figure 3. The job calculates its desire and communicates it to the allocator (steps 1 and 2). This change might have made the previous allocation unfair or inefficient. If that is the case, the allocator recalculates the allotments of jobs according to the change, so as to make the allocation fair and efficient again (step 3). The job can then enquire about its current allotment (step 4) and adjust the number of processors it is using to be equal to its current allocation (step 5).

The main tasks of the system are described in detail in the following sections.

Estimating the Job Desire

As we mentioned above, it is the responsibility of the jobs to calculate their desires and communicate them accurately to the allocator. In this prototype, we only consider Cilk jobs that employ the ramdomized work-stealing algorithm. Thus, we can use some heuristics based on the properties of the jobs to calculate their desires. [3] uses a heuristic based on steal rate; the idea here is that if a job has too many processors that are stealing (i.e. it has a high steal rate), then most of the processors dont have sufficient work to do, and so the desire of the job is low. On the other hand, if a job has a significantly low number of steals, then most of the processors must have work to do, so the desire of the job is high. In [3], the processor manager of a job keeps track of the number of steals, and if this number is less than a certain value, the desire of the job is doubled; if the number of steals is greater, the job's desire is reduced.

In this paper, we use another heuristic based on the number of frames in the ready deques of a job's processors. The number of frames can be used as an estimate of the job desire because it represents the number of threads that are ready to execute, or the work that could be done now if the job had an infinite

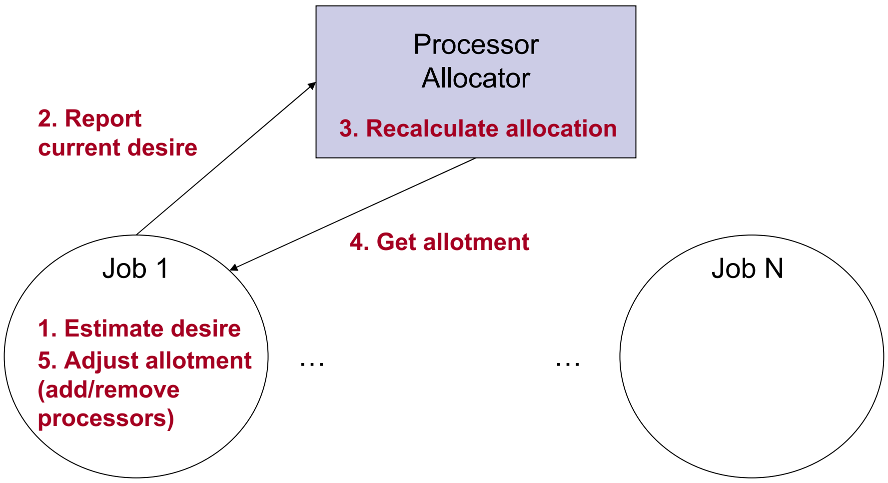

Figure 3: System overview: the five steps of the processor allocation system.

number of processors. We decided to use this approach instead of the steal rate idea because it is a more globally-aware heuristic. If one processor of a job has a lot of work to do, other processors might be stealing and increasing the steal rate, when in fact the job actually has a lot of ready work. Also, if the desire of a job is low, and we wait until there are a lot of steals, we waste a lot of time during which we could have already decreased the allocation. Since steal attempts and steals are expensive operations, and do not contribute at all to getting the work done, it is better to avoid them if possible. In our system, we designate a background processor to periodically add up the frames in all the processor deques and report the desire as a function of this sum. For the purpose of our prototype, we simply divide the total number of frames by a constant integer, as shown in Figure 4. This constant is a tunable parameter; we are not yet sure if there is a single value of k that is best for all situations.

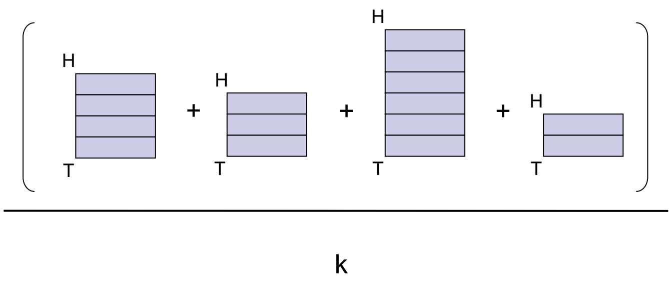

Figure 4: Our algorithm for measuring a job's desire: the number of frames on the deques of the processors are added together and divided by a tunable constant (k). There are 4 processors in this example.

Calculating a Fair and Efficient Allocation

The allocator which calculates the fair and efficient allocation is a global allocator that knows the desires and allotments of all the jobs in the system; these values are stored by the allocator in a table. The jobs may update their desires at any time, and the allocator changes the allotments so that the allocation is always fair and efficient. We implement an equipartitioning algorithm, similar to the one described in [6]. In the beginning, one processor is allocated to each job. Any jobs that reach their desires drop out. The scheduler repeats this process with the remaining jobs until either there are no remaining jobs or all the processors have been allocated. An allocation produced by equipartitioning is fair and efficient. However, we do not want to run the entire algorithm every time any job changes its desire. Thus, we use a subset of the algorithm which has the same effect and correctness, but completes faster than running the entire algorithm every time.

To explain our modified equipartitioning algorithm, we define a few terms:

- The maximum allotment of any job in the system at the current time is ***maxAlloc***; $maxAlloc = \max(a_i)$.
- The minimum allotment of any deprived job in the system at the current time is the ***fairShare***. $fairShare = \min(a_i)$, where i is a deprived job.

As explained in Section 2, the deprived job with the smallest allotment restricts the maximum allotment to be at most one more than this allotment. Thus, $fairShare \leq maxAlloc \leq fairShare + 1$. For simplicity, when there are no deprived jobs we define $fairShare = maxAlloc$.

Whenever a job j changes its desire, the allocator might have to change the allotments of j and some other jobs to maintain the fairness and efficiency of the allocation. The following code describes the actions of the allocator when job j changes its desire from d_j to d'_j. A new job entering the system is a special case of increasing an existing job's desire (from 0 to a positive integer). Similarly, when a job completes its execution, it decreases its desire to 0.

```
ChangeDesire(d_j, d'_j)
    if (d_j = d'_j)
        then return
    if (d'_j ≥ d_j)
        then a_j = min(a_j + freeProcs, d'_j)
        if (a_j ≥ fairShare)
            then return
        else IncreaseAllocation(j, a_j, d'_j)
    if (d'_j ≤ d_j)
        then if (a_j ≤ d'_j)
            then return
            else free = a_j − d'_j
            a_j = d'_j
            Reallocate(free)
IncreaseAllocation(j, a_j, d_j)
    while ((a_j ≤ d'_j) and (a_j ≤ fairShare))
        find a job i where (a_i = maxAlloc)
        Decrement a_i
        Increment a_j

Reallocate(free)
    while ∃i such that a_i ≤ d_i
        find a job i where (a_i = fairShare) and (a_i ≤ d_i)
        Decrement free
        Increment a_i
    freeProcs = freeProcs + free
```

The above code shows how the allocator behaves when job j changes its desire from d_j to d'_j. There are three possible cases:

1. **Desire of the job j increases**: If there are free processors available, then the allocator gives the free processors to the job until either the job has as many processors as it desires or we run out of free processors. But if there aren't enough free processors, then we might have to reallocate the processors—i.e, reduce the allotment of other processors to increase this job's allotment. There are two cases:

   (a) Case 1: The allocation of the job j is more than or equal to the $fairShare$. In this case, the job already has its fair share of processors and thus the allocation remains unchanged.

   (b) Case 2: The allotment of the job j is less than $fairShare$. In this case, the job doesn't have its fair share. Thus, we take processors from jobs which have more than their fair share and give them to this job.

2. **Desire of job j decreases**: There are again two cases:

   (a) Case 1: If the desire of j is still more than its current allocation, then there is no need to perform a reallocation since j can still efficiently use the processors that have been allocated to it.

   (b) Case 2: If the desire of j is now less than its current allocation, then we have to reduce the allotment of j and increase the allotment of other deprived jobs. Thus, we find deprived jobs that have an allotment equal to the $fairShare$ and increase their allotment by 1. We go on until we have run out of free processors or all jobs have allotments equal to their desires.

Theorem 1 *The allocation is always fair and efficient.*

Proof The proof is by induction.

Base Case: There are no jobs in the system to begin with. Thus, $freeProcs$ = total number of processors. When the first job, $Job1$, enters the system, if $d_1 \leq freeProcs$ then $a_1 = d_1$; otherwise, $d_1 = totalnumberofprocessors$. This allocation is obviously fair and efficient.

Inductive case: Suppose the allocation is fair and efficient at time t. At this time, job j changes its desire from d_j to d'_j. We consider what the above algorithm does in all cases and prove that the resulting allocation is also be fair and efficient. As stated earlier, the entrance of a new job into the system can just be treated as an increase of the desire of that job from 0 to d'_j. Similarly, when a job finishes its execution, it can be considered to reduce its desire to 0.

1. $d'_j \geq d_j$: The following three cases are possible

   (a) According to the above algorithm, if there are any free processors, then these free processors are given to job j. We assumed that the allotment was fair and efficient at time t. Hence, if there were free processors, none of the jobs were deprived at time t, due to the efficiency condition in Section 2. According to the code above, the free processors will be given to job j until

      i. $a_j = d'_j$: That is, when the allotment of j is equal to its desire. In this case, since we have established that all other jobs already have allotments equal to their desires, the allocation is fair and efficient.

      ii. $freeProcs = 0$ and $a_j \leq d'_j$: We have allocated all the processors and j is still deprived. In this case, we fall into the cases below, when there were no free processors to begin with.

   (b) There are no free processors and $a_j \geq fairShare$: The allocation was efficient at time t. Thus, no job had more processors than it desired, and we haven't given any more processors to any other job. Also, for job j, the allotment is greater than or equal to the fair share. As a result, all other jobs have at most one more processor than j. At this point, the fairness criteria has been satisfied, and the allocation is already fair and efficient (no reallocation is necessary).

(c) There are no free processors and $a_j \leq fairShare$: Now, job j doesn't have a fair share of processors and we need to reduce the allotment of other jobs to increase the allotment of j. This is done in the procedure INCREASEALLOCATION above. Here, we find a job i where $a_i = maxAlloc$ and reduce the allotment of i by 1. Now if $maxAlloc = fairShare + 1$, then new $a_i = fairShare$. Thus i still has its fair share. If $maxAlloc = fairShare$, then the fairShare will reduce by 1 and i will be the job with the smallest deprived allotment, so no other job can have more than $a_1 + 1$ processors. Thus, the allocation is still fair and efficient.

2. $d'_j \leq d_j$: When the desire of a job decreases, we might have to take processors from job j and allocate them to other jobs.

   (a) $a_j \leq d'_j$: When the allotment of j is still less than the new desire, we do nothing. Since the old allocation was fair, the new one remains fair too. Also since job j can still use its processors efficiently, the allocation remains efficient too.

   (b) $a_j \geq d'_j$: In this case, the allotment of j is now greater than its new desire, and we are violating the efficiency rule. Thus, we have to reduce the allotment of this job. Since $d'_j \leq a_j \leq fairShare + 1$, $d_j \leq fairShare$. So now $a_j = d'_j$, and we free the remaining processors. These processors have to be distributed to other jobs which are deprived. Since none of the deprived jobs can have allotments less than $fairShare$, we give these processors one by one to the deprived jobs and remain fair and efficient.

□

Adjusting a Job's Allotment

We discussed above that the job periodically calculates its desire and communicates it to the allocator, which adjusts the allotments accordingly. If the allotment of a job changes, it has to increase or decrease the number of processors it is currently using to be the same as its allotment. That is, the job might have to increase or decrease its p_i.

1. **Increase** p_i: If the allotment of a job increases, the job increases its p_i by starting to use more processors. We already know that the jobs report their desires periodically to the allocator. The allotments change immediately after this. At this point, the job gets its new allotment and if the number of processors it is using is less than its allotment, that is $p_i \leq a_i$, then it starts using more processors and makes $p_i = a_i$

2. **Decrease** p_i: If the allotment increases, the job has to decrease its p_i by reducing the number of processors in use. This can again be done when the desires are reported and the job gets new allotment. But this is complicated since all the processors might have work to do and the processors would have to move work from one processor to another. Thus, for ease of implementation, we decided not to do this in the prototype. Instead, every time a processor runs out of work and tries to steal, it checks the allotment. If the allotment is less than the number of processors being used, that is $p_i \geq a_i$, then this processor stops working for this job. At this point, this processor did not have any unfinished work (since it was stealing). Thus, this strategy is easy to implement. The disadvantage of the strategy is that p_i might remain more than a_i for considerable period of time depending on how often the job i steals. But this is unlikely, since the allocation of i has reduced, it is possible that its desire had also decreased before that, and thus it has more processors than it needs and there are likely to be more steal attempts as a result.

6 Implementation

We implemented a prototype of the dynamic processor allocation system as an extension to the Cilk runtime system. The implementation details of the three tasks described in Section 5 are given below.

Estimating the Job Desire

When a job starts up, it registers itself with the allocator and sets its initial desire as 1. After this, the job starts executing and periodically a background thread counts up the number of frames in the deque and updates the desire in the allocator. How often the desire is estimated is a tunable parameter and can be changed according to the number of jobs in the system and how often they change their desires.

Allocating the Processors

The processor allocator is implemented as a group of functions that share a global data-structure. This data-structure is a memory mapped file which is basically a big array that holds the desires and allotments of all the jobs in the system. The parameter $maxAlloc$ described in Section 5 is also stored in this file. Note that the parameter $fairShare$ can be easily inferred from the value of $maxAlloc$. When any job changes its desire, it just calls a function that simply executes the algorithm for fair and efficient allocation described in Section 5 and changes the allotments. To find the jobs to take or give processors to, it randomly picks a place in the array to start from and scans the array until it is done. A job can get the allotment from this array at any time by simply reading the value of its allotment from this file.

Adjusting the allotment

When a job allotment changes, the job has to increase or decrease the number of processors that are being used. In this prototype, the processors are implemented as virtual processors or to be more exact pthreads. When a job starts, it creates number of pthreads that are equal to the number of processors in the machine. On getting its first allotment (which is 1), the job puts most of the pthreads to sleep. After this whenever the allotment of a job increases, some of these pthreads are woken up. When the allotment decreases, the pthreads are put to sleep. We mentioned above that the number of processors are decreases during steals. This is when the pthreads are put to sleep. Thus we ensure that threads go to sleep only when they dont have any work.

7 Experimental Results

We have performed some preliminary tests on our processor allocation system to measure its correctness, overhead, and responsiveness to allocation changes. The system we are using is an SGI Origin 2000 SMP with 1 Gb of memory and 32 195 MHz processors. The tests we have performed do not constitute a full performance analysis of the system, but they serve as a good foundation for future experiments. We explain each of the tests, and the results we obtained, in the sections below.

For all tests, we estimate the processor desire every 10 milliseconds and use a value of 6 for k. These parameters are not optimal—they are simply the ones we settled on for the purpose of running the experiments.

Correctness

To verify the correctness of our processor allocation system, we run a few sample programs with different inputs and ensure that the outputs are correct in every case. The programs we use are the `nfib` program for calculating the nth Fibonacci number (included in the latest distribution of Cilk), and an optimized

Input (n)	1 processor (seconds)	32 processors (seconds)	Our System (seconds)
10	0.01	0.09	0.06
15	0.02	0.1	0.1
20	0.08	0.12	0.16
25	0.78	0.17	0.66
30	8.55	0.7	2.47
35	97.3	6.48	11.39
40	1048.8	70.4	154.7

Table 1: The elapsed time measured for separate runs of the `nfib` program on different inputs (n represents the nth Fibonacci number).

Input (n)	1 processor (seconds)	32 processors (seconds)	Our System (seconds)
256	1.74	0.6	1.2
512	11.91	2.15	3.58
1024	84.8	11.29	17.37
2048	591.2	72.3	117.7

Table 2: The elapsed time measured for separate runs of the `strassen` program on different inputs (n is the size of the matrices).

`strassen` program for multiplying two $n \times n$ matrices using Strassen's Algorithm. The optimization in the `strassen` program uses a fast, serial algorithm to multiply the submatrices when $n \leq 128$.

For both the `nfib` and `strassen` programs, all of the outputs obtained using our system are correct: the correct Fibonacci number is computed using `nfib`, and the correct matrix product is calculated using `strassen`. As an additional check, we ran the same tests without our processor allocation system (i.e. using the original Cilk runtime system), and we see that the corresponding outputs are identical.

Overhead

To measure the overhead of our processor allocation system, we run the `nfib` and `strassen` programs on different inputs, once using our system and again using the original Cilk runtime system. When using the original Cilk runtime system, we need to specify the number of processors to run the programs on: we perform each run using 1 processor (referred to as the 1 processor system) and again using 32 processors (referred to as the 32 processor system). During a given run, the program in question is the only job running on the system. The results from our tests are summarized in the tables below.

From Table 7, we see that the `nfib` program runs faster using our system than using the 32 processor system, for jobs that have insufficient parallelism to use all 32 processors efficiently. Our system performs better in this scenario because it is able to dynamically reduce the number of processors in use, so that no processors are going to waste. The performance of our system is still slower than the 1 processor system because of the overhead associated with creating and manipulating the worker threads (recall from Section 6 that P threads are created for each job at startup, even though most of them are put to sleep).

For higher values of n, the `nfib` program runs about twice as slow using our system than on the 32 processor system. This slowdown is primarily due to the work overhead incurred by the background thread when measuring and reporting the job's current desire, as well as the overhead incurred by the processor allocator when recalculating the allocation. It is possible that our system would perform better if the value of k is decreased, since this change would increase the estimated desires of the jobs (and the results for the

32 processor system seem to indicate that the higher runs of `nfib` benefit from having more processors).

For the `strassen` program, the results in Table 7 indicate that our system is always less than twice as slow as the 32 processor system. The reasons for this slowdown are identical to those for the higher runs of `nfib` explained above. We expect that a decreased value of k could also improve the performance of our system in this case.

Responsiveness

To measure the responsiveness of the processor allocation system, we measure the time it takes for a job to decrease its processor usage when the granted allotment is reduced by the allocator (recall that processors are reduced during work-stealing only). Since we are only running one job on the system at a time, however, the granted allotment usually drops by 1 processor at a time, and so the measured response is almost instantaneous. We do not measure the response time for increases in job allotments because this is handled synchronously by the background thread.

We tried to artificially decrease the allotment of a job and measure the system's response time. For `nfib` run on $n = 30$, it took between 15 and 50 milliseconds for the job's processor usage to decrease from 10 to 5 processors. The same experiment took between 50 millisecond and 2 seconds for `strassen` run on $n = 1024$. In general, it is not a good idea to interpret these results: since jobs only reduce their processor usage while work-stealing, even if we artificially reduce a job's allotment, the number of processors it uses does not immediately decrease if the job's desire has not really changed.

Additional Experiments

There are a number of additional tests that we would like to perform. Most of these tests involve running multiple, concurrent Cilk jobs and evaluating the performance of our processor allocation system (we have not had time to do this yet). In particular, we would like to perform the following tests:

1. Vary the frequency with which the job desires are estimated and measure the system's performance when running multiple jobs (we already tried this test for single jobs but did not notice any significant improvements/degradation in performance).

2. Vary the value of k and measure the system's performance both when running single jobs and when running multiple.

3. Run a pair of jobs together that have complementary parallelism profiles (e.g. one job starts out with high parallelism and ends with low parallelism and the other job starts out with low parallelism and ends with high parallelism). Try to find a pair of jobs that performs better using our system than with the original Cilk runtime system.

4. Run a group of jobs with arbitrary parallelism profiles and measure the system's performance.

8 Future Work

There are a number of extensions to the processor allocation system that we would like to explore in the future. We divide these extensions according to the task they pertain to, and discuss them below.

Estimating the Job Desire

We would like to consider ways of combining the steal rate idea used in [3] with the heuristic used in our system, to see if some combination of the two allows us to measure a job's parallelism more accurately.

Allocating the Processors

There are many ways to improve both the design and implementation of the processor allocator. In an implementation level, we can consider using a sorted data structure to store the job entries instead of the linear array that we currently use; the jobs could be sorted using their desires or their allotments, or perhaps the difference between the two numbers. Another alternative would be to divide the jobs into separate groups: jobs whose desires are satisfied, deprived jobs that have the smallest allotment, and deprived jobs that have the maximum allotment.

On a design level, we would also like to consider alternative definitions for fairness and efficiency. For instance, we could define fairness in a way that incorporates the histories of processor usage for each job, or we could design a mechanism for respecting different job (and user) priorities. As a longterm goal, we would like to generalize our desire estimation and processor allocation algorithms to work for all types of parallel jobs; this way, the processor allocation system can be generalized into a core operating system service, and moved into the kernel itself.

Adjusting the Allotment

One improvement to our method of adjusting allotments that is relatively easy to implement is to allow processors to be removed by the background thread (instead of during work steals). This optimization would result in a faster response time to changes in a job's allotment, but would also require a mechanism for putting processors with pending work to sleep (and waking them up when increasing the allocation).

9 Conclusion

In this paper, we introduced a simple heuristic for estimating the instantaneous parallelism of an adaptively parallel job, based on the amount of pending work queued on the job's processors. For the specific case of parallel jobs scheduled with the randomized work-stealing algorithm, we demonstrated how this heuristic can be used to design a processor allocation system that allocates processors to jobs in a fair and efficient manner. We tested our design by implementing a dynamic processor allocation system for Cilk jobs running on a shared-memory symmetric multiprocessor (SMP) system. We provided empirical evidence evaluating the correctness and overhead of our system, and outlined the experiments to be run in the future. Finally, we discussed various extensions and improvements to our processor allocation system, which we plan to investigate in the future.

References

[1] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995.

[2] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computation by work stealing. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, November 1994.

[3] R. D. Blumofe, C. E. Leiserson, and B. Song. Automatic processor allocation for work-stealing jobs.

[4] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *ACM SIGPLAN Conference on Programming Language*, 1998.

[5] Supercomputing Technologies Group. *Cilk 5.3.2 Reference Manual.* MIT Lab for Computer Science, November 2001.

[6] B. Song. Scheduling adaptively parallel jobs. Master's thesis, Massachusetts Institute of Technology, January 1998.