6.895 Theory of Parallel Systems	Kunal Agrawal, Siddhartha Sen

Adaptively Parallel Processor Allocation for Cilk Jobs

Project Proposal

1 Problem Statement

Our goal is to design and implement a dynamic processor-allocation system for adaptively parallel jobs. *Adaptively parallel jobs* are jobs for which the number of processors which can be used without waste varies during execution—these are known as *adaptively parallel jobs.* We call the problem of allocating processors to adaptively parallel jobs the *adaptively parallel processor-allocation problem* [2].

We propose to investigate the adaptively parallel processor-allocation problem for multiple Cilk jobs running on a shared-memory multiprocessor (SMP) system. Our goal is to design a processor-allocation system that achieves a "fair" and "efficient" allocation among all jobs; we define the terms "fair" and "efficient", and other relevant terms, in Section 2 below. The design of such a system can be divided into two primary subtasks:

1. Dynamically determine the number of processors desired by each Cilk job on the system.
2. Dynamically determine the number of processors that should be allocated to each job such that the resulting allocation is fair and efficient.

The definition of "dynamic" here is to be determined; that is, we need to consider when and how frequently the job desires and allocations are recomputed.

2 Definitions and Assumptions

In this document, the term "processor" is used interchangeably with the term "thread", so as not to limit our discussion to multiprocessor systems, or limit the number of jobs running on the system to the number of physical processors contained in it.

We consider a shared-memory multiprocessor (SMP) system with P processors and J jobs. In general, we follow the terminology and conventions used in [2], 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 m_j, representing the number of processors allocated to it. Our problem, as stated in 1, is to find a fair and efficient allocation of processors among these jobs.

We define the terms "fair" and "efficient" identically to [2]. In particular, an allocation is *fair* if whenever a job receives fewer processors than it desires, then no other job receives more than one more processors than this job received (the allowance of one processor is due to integer roundoff). An allocation is *efficient* if no job receives more processors than it desires.

Note that if $J > P$, one or more processors in the system will be allocated to more than one job, meaning that multiple threads of execution (where each thread belongs to a distinct job) may be running on the same processor. This is why we drew the equivalence between the terms "processor" and "thread" at the beginning of this section.

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

- All jobs on the system are Cilk jobs.
- Jobs can enter and leave the system and change their parallelism during execution.

- All jobs are mutually trusting, in that they will stay within the bounds of their allotments and communicate their desires honestly.
- Each job has at least one processor to start with.

3 Previous Work

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

In [2], Song presents a randomized processor-allocation algorithm, called the SRLBA algorithm, for the adaptively parallel procesor-allocation problem. The SRLBA algorithm is a variant of the Randomized Load-Balancing (RLB) algorithm consisting of rounds of load-balancing steps in which processor migration (from one running job to another) may occur. 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. The main result of [2] states that if all jobs have a desire greater than the *absolute average allotment* P/J of processors, then within $O(\lg P)$ rounds, 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 implement a job scheduler for the Cilk [1] runtime system 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). The "steal rate" is used by the scheduler to implement a simple processor-allocation algorithm that achieves a fair and efficient allocation for all Cilk jobs. The job scheduler is implemented in user space; it consists of a processor manager for each Cilk program and a job registry (stored in shared memory) through which the processor managers can communicate.

In [4], Waldspurger and Weihl present a novel randomized resource-allocation mechanism called *lottery scheduling*. Lottery scheduling provides efficient, responsive control over the relative execution rates of computations, making it useful in systems that service jobs of varying importance. Resource sharing is described at the level of threads running concurrently on the same processor (although nothing prevents us from viewing the processors as resources themselves). Lottery scheduling works by using lottery tickets to represent resource rights, and holding a lottery to determine the winner of a given allocation (i.e. the resource is granted to the client with the winning ticket). A currency abstraction is also introduced to insulate resource allocation policies and protect resource rights across logical trust boundaries—e.g. between different classes of jobs, in our case.

4 Design Alternatives

We now present the preliminary design options for our solution to the adaptively parallel processor-allocation problem. We divide the design issues into two categories: algorithmic issues and implementation issues.

4.1 Algorithmic Issues

There are two main algorithmic questions that need to be addressed, based on our problem statement in Section 1: 1) how to dynamically estimate the processor desires of jobs, and 2) what processor-allocation algorithm should be used.

Estimating Processor Desires

The following is a preliminary list of ideas for estimating the processor desires of jobs:

1. Since Cilk uses a work-stealing algorithm, we can leverage the steal rate idea from [3].

2. We could use heuristics based on the number of activation records on the stack view of each processor.

3. We could use heuristics based on the number of threads on the ready deques of each processor.

Processor-Allocation Algorithm

The following is a preliminary list of ideas for what processor-allocation algorithm to use:

1. We can use the SRLBA algorithm as inspiration, although that algorithm is for distributed multiprocessor systems, and we are focussing on SMPs.

2. We can analyze, modify, and extend the algorithm used in the Cilk macroscheduler implementation in [3].

3. Consider using some of the techniques and ideas from the lottery scheduling algorithm presented in [4].

4.2 Implementation Issues

There are several implementation issues that we need to consider. These issues are summarized in the list below:

1. The processor-allocation system can be implemented in either user-space or kernel-space. We have chosen to do it in user-space for simplicity and portability.

2. The algorithm for estimating processor desires and the procesor-allocation algorithm should be implemented as separate modules that can be plugged into the overall system and replaced easily.

3. The jobs need to communicate their desires to our processor-allocation system; there are a few ways of doing this (different forms of IPC, memory-mapped files, RPC, etc.).

4. We need to determine the frequency with which jobs will communicate their desires or current usage to the processor-allocation system. This may or may not depend on the algorithm being used to estimate the processor desires.

5. We need to decide the actual mechanism for increasing and decreasing the allocations of jobs (putting threads to sleep/waking them up, creating threads/killing them, etc.).

5 Progress Milestons and Possible Extensions

We will start with a very simple implementation of the processor-allocation system that uses stubs in place of the main algorithms. Once we are able to get this skeletal system working, we will begin analyzing and implementing the different algorithms we have proposed in 4.

If time permits, we will consider the following extensions to our project:

1. Implement the processor-allocation system in kernel space.

2. Extend the system to handle other (non-Cilk) types of adaptively parallel jobs.

3. Consider a mechanism for assigning different priorities to users or jobs (note that this may go against our fair-and-efficient requirement).

References

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

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

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

[4] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the First Symposium on Operating System Design and Implementation*, pages 1–11. USENIX, November 1994.