Executing Communication-Intensive Irregular Programs Efficiently

Vara Ramakrishnan* Isaac D. Scherson

Department of Information and Computer Science
University of California, Irvine, CA 92697
{vara,isaac}@ics.uci.edu

Abstract. We consider the problem of efficiently executing completely irregular, communication-intensive parallel programs. Completely irregular programs are those whose number of parallel threads as well as the amount of computation performed in each thread vary during execution. Our programs run on MIMD computers with some form of space-slicing (partitioning) and time-slicing (scheduling) support. A hardware barrier synchronization mechanism is required to efficiently implement the frequent communications of our programs, and this constrains the computer to a fixed size partitioning policy.
We compare the possible scheduling policies for irregular programs on fixed size partitions: local scheduling and multi-gang scheduling, and prove that local scheduling does better. Then we introduce competitive analysis and formally analyze the online rebalancing algorithms required for efficient local scheduling under two scenarios: with full information and with partial information.

1 Introduction

The universe of parallel programs can be broadly divided into regular and irregular programs. Regular programs have a fixed number of parallel threads, each of which perform about the same amount of computation. Irregular programs have a varying number of parallel threads and/or threads which perform unequal amounts of computation. Since the behavior of regular programs is predictable, scheduling them well is relatively easy. There are several static (job arrival time) and dynamic scheduling methodologies [3] which work very well on regular programs. For this reason, we focus on irregular programs, which waste computing resources if managed poorly.

There are two main classes of parallel computers, MIMD (multiple-instruction multiple-data) and SIMD (single-instruction multiple-data). MIMD computers have full-fledged processors which fetch, decode and execute instructions independently of each other. SIMD computers have a centralized control mechanism which fetches, decodes and broadcasts each instruction to specialized processors, and they all execute the instruction simultaneously.

* This research was supported primarily by PMC-Sierra, Inc., San Jose, California. http://www.pmc-sierra.com

We assume that the parallel computer is an MIMD computer. MIMD computers have overwhelming advantages in cost and time-to-market, provided by using off-the-shelf processors, whereas SIMD computers need custom-built processors. MIMD computers can also be used more efficiently for the following reasons: 1. An MIMD computer does not force unnecessary synchronization after every instruction, or unnecessary sequentialization of non-interfering branches of computation, as an SIMD computer does. 2. Many jobs can be run simultaneously on different processors (or groups of processors) of an MIMD computer. These factors, among others, justify our assumption and explain the continuing market trend towards MIMD computers, exemplified by the Thinking Machines CM-5, the Cray Research T3D and, more recently, the Sun Microsystems Enterprise 10000 and the Hewlett-Packard HyperPlex. To enable sharing by multiple programs, MIMD computers usually provide some form of time-slicing (scheduling) or space-slicing (partitioning) or both.

Parallel irregularity can be classified as follows: variation in parallelism (number of threads) during execution (termed *X-irregularity*), and variation in the amount of computation performed per thread (termed *Y-irregularity*). Of course, a program may be both X and Y-irregular: *completely irregular*, which is the class of programs we consider. This means our programs definitely exhibit X-irregularity, the behavior of spawning and terminating parallel threads at run-time. We then have to make run-time decisions on where to schedule newly spawned threads, and whether to rebalance remaining threads after threads terminate.

Several examples of communication-intensive irregular parallel programs can be found among parallelized electronic design automation applications, and parallel search algorithms used in internet search engines and games. Besides being irregular, these programs require frequent communications among several or all of the running threads.

In Section 2 of this paper, we discuss why the most efficient way of implementing frequent communications is barrier synchronization, using a hardware tree mechanism. For a detailed discussion of barrier synchronization and implementation methods, see [4]. Hardware barrier synchronization trees impose fixed size processor partitions on the computer. In Section 3, we discuss the scheduling strategies available for fixed size partitions. We show that local scheduling does better than a policy based on gang scheduling for Y-irregular programs, provided we are able to balance threads across processors.

In Section 4, we discuss the problem of balancing threads across processors. Due to frequent communications between threads, it is essential to schedule a newly spawned thread almost immediately. Therefore, the only practical way to schedule a new thread is to temporarily run it on the same processor as its parent thread, and periodically rebalance the threads in a partition to achieve an efficient schedule. We outline an optimal online algorithm which decides when to rebalance, and prove that it does no worse than 2 times the optimal off-line strategy. The optimal online algorithm requires complete information about the scheduling state of the partition, which is prohibitively expensive to gather.

Therefore, we propose a novel way to gather partial state information by using the barrier synchronization tree available on the computer. Then, we propose and analyze an online algorithm which uses only this partial state information. We show that our online algorithm based on partial state information performs no worse than n times the optimal off-line strategy, where n is the number of processors. This implies that our worst case performance is limited to the performance of running the program sequentially. We intend to expand on this work by demonstrating experimentally that the average case performance of our partial state online algorithm is close to the optimal online algorithm.

2 Constraints on Execution

This section describes the constraints placed by our program characteristics on their efficient execution. We outline the most effective way to implement the frequent communications of our programs, and its impact on partitioning and scheduling options.

2.1 Barrier Synchronization

The overhead of implementing frequent communications as multiple pairwise communications across the data network is very high. An efficient alternative is *barrier synchronization*: a form of synchronization where a point in the code is designated as a barrier and no thread is allowed to cross the barrier until all threads involved in the computation have reached it. Usually, threads are barrier synchronized before and after each communication. This ensures that all data values communicated are current, without doing any pairwise synchronizations between threads. Barrier synchronization can be implemented in software or in hardware.

Software barriers are implemented using shared semaphores or message passing protocols based on recursive doubling. Software implementations provide flexible usage, but they suffer from either the sequential bottleneck associated with shared semaphores or the large communication latencies of message passing protocols. The time for a software barrier to complete is measured in data network latencies, and is proportional to n for shared semaphores and to $\Theta(\log n)$ for protocols based on recursive doubling, where n is the number of processors.

Hardware barriers are implemented in their simplest form as a single-bit binary tree network, called a barrier tree. The barrier tree takes single-bit flags from the processors as its inputs, and essentially implements recursive doubling in hardware, using AND gates. The time for a hardware barrier to complete is measured in gate delays, and is proportional to the height of the tree, or equivalently, $\Theta(\log n)$.

Although barrier trees are commonly implemented as complete binary trees, this is not essential. A barrier tree can be any spanning tree of the processors, and in fact, such an implementation facilitates partitioning of the computer for multiple programs [5].

Instead of a single-bit tree, the barrier tree can be constructed with m-bit edges, and any associative logic on m-bit inputs (such as maximum, minimum, sum) can replace the AND gates in the barrier tree. In this case, barriers will complete after a hardware latency no worse than $\Theta(\log n \log m)$. [4] shows that a tree which computes the maximum of its input flags, a *max-tree*, is useful in synchronizing irregular programs efficiently. Later in this paper, we discuss a way to also use such a tree in scheduling irregular programs.

In general, hardware barrier trees are intrinsically parallel and have very low latency, so they are at least an order of magnitude faster than software barriers.

2.2 Fixed Size Processor Partitions

If barrier synchronization is implemented in software, it imposes no constraints on the partitioning policy. For example, a partition may be defined loosely as the set of processors assigned to the threads of a job, and it may be possible to increase the partition size whenever a new thread is spawned. Due to this flexibility, scheduling new threads is an easier problem on such computers.

If barrier synchronization is implemented as a hardware tree, the barrier tree dictates the possible processor partitions on the computer. This is because we need to ensure that a usable portion of the barrier tree is available on each partition. Then, space-slicing can only be done by assigning fixed size partitions (usually of contiguous processors) to each program. Assuming there is only one barrier tree available per partition, partitions must be non-overlapping. It may be possible to resize partitions dynamically, but this is a slow operation since all the threads running on the partition must be quiesced and the barrier tree reconfigured for the new partition size. Therefore, resizing has to be done at infrequent enough intervals that we may assume a fixed size partition for the duration of a program. We treat a run-time partition resizing as a case where a program terminates all its threads simultaneously, and a new program with the same number of threads starts on the new partition. Note that if the computer does not support partitioning at all, then the entire computer can be treated as one fixed size partition for our purposes.

Scheduling in the presence of a hardware barrier tree is a harder problem and more applicable to the class of programs we are interested in. Therefore, in the rest of this paper, we assume that we are executing on computers with a hardware barrier synchronization mechanism, specifically a max-tree, and fixed size, non-overlapping partitions.

3 Scheduling on Fixed Size Partitions

Since barrier synchronization is implemented in hardware, when a new thread is spawned by a program, it must be scheduled on one of the processors within the fixed size partition. Given this fixed size partitioning policy, let us consider what the scheduling options are:

If there is no time-slicing available on the computer, threads cannot be preempted, meaning that a job will relinquish its entire partition only upon completion. Without time-slicing, each processor can run no more than one thread, so there is no way to run a job whose number of threads exceeds the number of processors in the largest partition (which may be the entire computer). This also means it is only possible to run X-irregular jobs which can predict the maximum number of threads they may have at any time during execution (this may not be feasible), and there is a large enough partition to accommodate that number. Therefore, to run X-irregular jobs without restrictions, it is essential that time-slicing be available on the computer.

One form of time-slicing called *gang scheduling* is possible on fixed size partitions. In gang scheduling, each processor has no more than one thread assigned to it, and threads never relinquish processors on an individual basis. At the end of a time-slice, all threads in the partition are preempted simultaneously using a centralized mechanism called multi-context-switch. Then, new threads (either of the same or a different job) are scheduled on the partition. (Note that it is possible to schedule different jobs in each time-slice because the state of the barrier tree can be saved along with the job's other context information, effectively allowing the barrier tree to be time-sliced as well.) By gang scheduling threads of the same job in more than one time-slice, called *multi-gang scheduling*, it is possible to run X-irregular jobs. However, this is inefficient because each processor will be idle after its thread reaches a barrier, wasting the rest of the time-slice. To address this, the time-slice can be selected to match the communication frequency, but if the job is Y-irregular, the time-slice has to be large enough to allow the longest thread to reach its barrier. In such a case, it would be helpful to allow the barrier tree to trigger the multi-context-switch hardware when the longest thread reaches the barrier, rather than using fixed length time-slices (this feature is not available on any computers we know of). There would still be some idling on most of the other threads' processors due to Y-irregularity, but this cannot be completely eliminated in multi-gang scheduling.

An alternate form of time-slicing, called *local scheduling*, mitigates the idling caused by Y-irregularity. Local scheduling is possible within fixed size partitions, and requires that each processor is capable of individually time-slicing multiple threads allocated to it. These threads must all belong to the same job, since threads from multiple jobs cannot share a barrier tree simultaneously and there is only one barrier tree per partition. In a local scheduled partition, the processor preempts each thread when it reaches a barrier, giving other threads a chance to run and reach the barrier. The processor only sets its flag on the barrier tree after all its threads have reached the barrier. In other words, the processor locally synchronizes all its threads and places the result on the barrier tree.

3.1 Handling Y-Irregularity

Barriers divide the program execution into *barrier phases*, and any Y-irregularity in the program is fragmented into these barrier phases. There will be some processing resources wasted in each barrier phase because not all threads have

the same amount of computation to perform before reaching the next barrier. We show that local scheduling can usually do better than multi-gang scheduling in eliminating some of this waste.

Theorem. *Given a Y-irregular program with a large number of threads distributed evenly across processors, multi-gang scheduling cannot perform any better than local scheduling.*

Proof. Let the total number of threads in the job be N, and the number of processors in the partition be a much smaller number n. The number of threads on each processor is either $m = \lceil \frac{N}{n} \rceil$, or $m - 1$.

Let the time for thread i on processor j to reach the barrier be t_{ij}, with discrete values varying between 0 and M. (If a processor j has only $m - 1$ threads, then $t_{mj} = 0$.) In multi-gang scheduling, the time for one barrier phase to complete is

$$T_m = \sum_{i=1}^{m} \max_{j=1}^{n}(t_{ij})$$

while in local scheduling, the time for one barrier phase to complete is

$$T_l = \max_{j=1}^{n} \sum_{i=1}^{m}(t_{ij})$$

T_m is the sum of the maximum t_{ij} values across all the processors. T_l is the maximum among the sums of the t_{ij} values on each processor. The only way T_l could be as large as T_m is when the largest t_{ij} values all happen to occur on exactly the same processor, whose sum would then be selected as the maximum. For all other cases, T_l would be smaller than T_m. Therefore, $T_m \geq T_l$. This proves the theorem. □

Since the odds of all the largest t_{ij} values occurring on the same processor are very low, local scheduling generally does better than multi-gang scheduling. Intuitively, local scheduling tends to even out differences in barrier phase times across processors by averaging across the local threads.

For the above reason, as well as the fact that barrier tree triggered multi-context-switch mechanisms are not available on any existing computer, we assume that local scheduling as opposed to multi-gang scheduling is used to run X-irregular jobs on our computer. This gives rise to the problem of ensuring that the job's threads are distributed evenly across processors, which is addressed in Section 4.

Pure local scheduling has the disadvantage of not allowing a partition to be shared by more than one job. This means that a decision to run a particular job may have a potentially large, persistent and unpredictable impact on future jobs wanting to run on the computer [2]. To avoid this problem, a combination of both forms of time-slicing called *family scheduling* [1] is possible as well. In

family scheduling, it is assumed that the number of threads in a job is larger than the partition size, and they are distributed across the processors as evenly as possible (the number of threads on any two processors may differ by at most 1). Multiple jobs are gang scheduled on the partition. Within its allotted partition and gang scheduled time-slice, multiple threads of the job are local scheduled on each processor. For our purposes, we can treat family scheduling and local scheduling as equivalent, since the gang scheduling time-slice is usually much larger than barrier phase times on our jobs.

3.2 Handling X-Irregularity

Since communications are implemented using barrier synchronizations, they cannot complete unless all threads participate. Therefore, to ensure job progress, all threads must be executed simultaneously or at least given some guarantee of execution within a short time bound. This means newly spawned threads must be scheduled almost immediately to keep processors from idling. Since we have a fixed partition size, the only practical option is to schedule a newly spawned thread on the same processor as its parent thread. This will temporarily cause an imbalance on that processor (and violate the family scheduling rule that the number of threads on any two processors differ by at most 1).

A thread's probability of spawning other threads may be data dependent, causing some processors to become heavily loaded compared to others. Therefore, to ensure efficient execution, spawned threads will have to be migrated to other processors. Similarly, when threads terminate, some processors may be underutilized till the remaining threads are migrated to rebalance the load.

4 Online Rebalancing of Threads

At every barrier synchronization point, we have the opportunity to gather information about the state of the partition. If we find an imbalance in the number of threads across processors, we can make the decision whether to correct the imbalance or to leave the threads where they are and continue running till the imbalance gets worse. In making this decision, we must weigh the cost of processor idling due to the imbalance against the cost of rebalancing. In addition, there is also the cost associated with gathering information about the state of the system to enable our decision making, but we will ignore this for the moment. Note that we use the term *cost* to indicate the time penalty associated with an action.

We must make our decisions *online*: at a given barrier, we have knowledge of the previous imbalances in the system, and the current state. We also know the cost of rebalancing the system, which varies as a function of the imbalance. With this partial knowledge, we must decide whether to rebalance the system or not. In contrast, a theoretical *offline* algorithm knows the entire sequence of imbalances in advance and can make rebalancing decisions with the benefit of foresight.

Consider a fixed sequence of imbalances σ. Let $C_{OPT}(\sigma)$ be the cost incurred by the optimal offline algorithm on this sequence. Let $C_A(\sigma)$ be the cost incurred by an online algorithm A on the same sequence. Algorithm A is said to be r-competitive if for all sequences of imbalances σ, $C_A(\sigma) \leq r \cdot C_{OPT}(\sigma)$. The competitive ratio of algorithm A is r. This technique of evaluating an online algorithm by comparing its performance to the optimal offline algorithm is called *competitive analysis*. It was introduced in [6] and has been used to analyze online algorithms in various fields. The optimal offline algorithm is often referred to as an *adversary*.

Note that a thread's time to reach a barrier may be data dependent, which would mean that all threads cannot be counted as equals when rebalancing decisions are made. However, due to the dynamic nature of thread behavior over the duration of a program, it is very hard to predict a thread's time to reach a barrier. Even if that information were predictable, using it to further improve scheduling decisions is usually not feasible. This is because the variation in times to reach a barrier is limited by the very small time for each barrier phase (due to communication frequency), and it is difficult to make scheduling decisions with low enough overhead to actually recover some of that small time wasted in each barrier phase.

In the rest of this paper, we assume that each thread utilizes its barrier phase fully for computation. In addition, we also assume that the barrier phase time does not vary significantly over the duration of the program. The above two assumptions enable us to treat threads of a job as equals, and rebalancing has the sole objective of equalizing the *number* of threads across processors.

4.1 Costs of Imbalance and Rebalancing

Initially, we assume that complete information about the number of threads on all processors in the job's partition is available at each processor. This information is actually expensive to gather, but we will discuss this expense and alternatives in the next section.

We first define two terms which are used to analyze the costs of imbalance and rebalancing:

- The *system imbalance*, δ is the maximum thread imbalance on any processor. If there are n processors and N threads, let $m = \lfloor \frac{N}{n} \rfloor$ denote the average number of threads in the system. If k_j is the number of threads on processor j, then $\sum_{j=0}^{n-1} k_j = N$. The thread imbalance on processor j is $\delta_j = k_j - m$. Note that δ_j values can be either positive or negative, and their maximum, by definition, has a value of 0 or higher. The system imbalance, $\delta = \max_{j=0}^{n-1} \delta_j$.
- The *aggregate system imbalance*, Δ is the number of threads that need to be moved in order to balance the system.
  Let δ_{Aj} denote the absolute value of the thread imbalance on each processor. In other words, for each processor, j, $\delta_{Aj} = |\delta_j|$. The aggregate system imbalance, $\Delta = \frac{\sum_{j=0}^{n-1} \delta_{Aj}}{2}$.

The cost of running without rebalancing at a barrier is c, where $c = t \cdot \delta$, and t is the time for any thread to reach any barrier, which we have assumed to be a constant for a given program. The cost of rebalancing the tree is C, where $C = x + y \cdot \Delta$, and x and y are constants which depend on the implementation of the data network on the machine. These costs c and C are used to analyze online rebalancing algorithms. For our analysis, we assume that x is negligibly small compared to $y \cdot \Delta$. Note that for communication-intensive programs, t is very small, so C is significantly larger than c on any machine.

When the job is initially scheduled or after the last rebalancing, the threads in the partition are evenly distributed across all processors. At this point, δ may be 0 or 1, depending on whether N is a multiple of n. Therefore, the lowest value of c which represents a correctable system imbalance is generally $2t$. It is reasonable to assume that the system places a limit M on the maximum number of threads that can be run per processor, and this limit is helpful in bounding the values of c and C later on.

An online algorithm would maintain a sum Σ of all the c values encountered (resetting Σ whenever c has a value of 0 or t, corresponding to $\delta = 0$ or 1). When Σ equals some threshold T, a system rebalance is triggered, at a cost of C.

4.2 Optimal Online Algorithm

Case 1: Consider an algorithm which selects a T smaller than C. Therefore, $T = C - \epsilon$. To minimize the performance of this algorithm, the adversary would keep the system at a minimum imbalance at all times, by spawning or terminating threads. Therefore, the algorithm pays the cost C of rebalancing the system, while the adversary never pays a rebalancing cost. However, both the algorithm and the adversary pay the cost $\Sigma = T$ of running with an imbalance. The algorithm's cost is $C+T$, while the adversary's cost is T, making the algorithm's competitive ratio $r = \frac{2C-\epsilon}{C-\epsilon} = 1 + \frac{C}{C-\epsilon}$. Therefore, $r \geq 2$ for all values of ϵ, and has a minimum value of 2 occurring when $\epsilon = 0$.

Case 2: Consider an algorithm which selects a T larger than C. Therefore, $T = C + \epsilon$. To minimize the performance of this algorithm, the adversary would rebalance its system as early as possible, therefore paying no costs for running with an imbalance. Once again, the algorithm's cost is $C + T$, while the adversary's cost is C, making the algorithm's competitive ratio $r = \frac{2C+\epsilon}{C} = 2 + \epsilon/C$. Again, $r \geq 2$ for all values of ϵ, and has a minimum value of 2 occurring when $\epsilon = 0$.

From the above two cases, we see that the optimal online algorithm selects $T = C$, and has a competitive ratio of 2.

4.3 Low Overhead Alternatives

The cost of gathering complete information about the system configuration at each barrier is too high, requiring n phases of communication over the data network, where each processor is allowed to declare how many threads are assigned to it. To enable hardware barrier synchronization, we assumed that a max-tree

is available on the computer. This tree can be used to inexpensively compute the maximum and minimum number of threads per processor on our system. This is done in two phases:

1. Each processor j places its k_j in its max-tree flag, and the tree returns their maximum, $\max_{j=1}^{n} k_j$ to all the processors. This value directly corresponds to the maximum number of threads running on any processor.

2. Each processor places $M - k_j$ in its max-tree flag, and the tree returns their maximum, $\max_{j=1}^{n}(M - k_j)$ to all the processors. By subtracting this value from M, each processor calculates the minimum number of threads running on any processor, since $M - \max_{j=1}^{n}(M - k_j) = M - [M - \min_{j=1}^{n} k_j] = \min_{j=1}^{n} k_j$.

We wish to consider algorithms which estimate costs only based on the difference between the minimum and maximum number of threads on the processors, without knowing the actual average number of threads on the system.

The average number of threads on the system has to lie between the minimum and maximum number of threads on any processor. Therefore, the algorithm faces the worst uncertainty in guessing the average when the maximum and minimum are as far apart as possible. This happens in system configurations where at least one processor has 0 threads and at least one has M threads. We refer to the set of system configurations with this property as *0-M* configurations. We need to consider only *0-M* configurations since we are interested in estimating the algorithm's worst case behavior.

The rebalancing period p is the number of barriers run with an imbalance, after which the algorithm chooses to rebalance. The algorithm would have to guess values of c and C, and select a value for p that is as close as possible to $\frac{C}{c}$.

If the algorithm underestimates p, it would rebalance too often, and the adversary's strategy would be to keep the system at a minimum imbalance at all times. The adversary would never pay a rebalancing cost, while the algorithm would pay it more often than it would with complete information. Both pay the cost of running with the minimum imbalance at all times.

If the algorithm overestimates p, it would rebalance too infrequently, and the adversary's strategy would be to cause the maximum system imbalance and rebalance immediately. The adversary pays the rebalancing cost, while the algorithm pays the rebalancing cost as well as the cost of running with the maximum imbalance for longer than it would with complete information.

Now we analyze *0-M* configurations to arrive at the minimum and maximum values that δ and Δ can take. For any *0-M* configuration, without loss of generality, we can also assume that processor 0 has 0 threads, and processor $n-1$ has M threads.

Note that the system imbalance δ is always attributable to the processor with the largest number of threads running on it. Therefore, we may assume that $\delta_{n-1} = \delta$.

The maximum δ, denoted by δ_{MAX}, has to occur when $\delta_{n-1} = M - m$ is maximized, which is when m is minimized. This happens when processors 0 through $n-2$ all have 0 threads, making $m = \frac{M}{n}$, and $\delta_{n-1} = M - \frac{M}{n} = \frac{M(n-1)}{n}$. (Any other configuration would have more threads on some of the processors 0

through $n-2$, making the mean m higher and reducing the value of δ_{n-1}.) Therefore, δ_{MAX} is $\frac{M(n-1)}{n}$. For this configuration, referred to as *Configuration* α, $\Delta = \frac{(n-1)\cdot\frac{M}{n}+M-\frac{M}{n}}{2}$, which simplifies to $\frac{M(n-1)}{n}$.

Similarly, the minimum δ, denoted by δ_{MIN}, has to occur when $\delta_{n-1} = M - m$ is minimized, which is when m is maximized. This happens when processors 1 through $n-1$ all have M threads, making $m = \frac{M(n-1)}{n}$, and $\delta_{MIN} = M - \frac{M(n-1)}{n} = \frac{M}{n}$. For this configuration, referred to as *Configuration* β, $\Delta = \frac{\frac{M(n-1)}{n}+(n-1)\cdot[M-\frac{M(n-1)}{n}]}{2}$, which simplifies to $\frac{M(n-1)}{n}$.

Note that Δ has the same value, $\frac{M(n-1)}{n}$ at δ_{MIN} and at δ_{MAX}. This is also the minimum number of threads that need to be moved to balance any *0-M* configuration. Therefore, the minimum value of Δ, Δ_{MIN} is $\frac{M(n-1)}{n}$.

The maximum value of Δ, Δ_{MAX} occurs when half the processors have M threads and the other half have 0 threads. In this configuration, $m = \frac{M}{2}$. To balance this configuration, $\frac{M}{2}$ threads have to be moved from $\frac{n}{2}$ processors (who have M threads) to the others. Therefore, $\Delta_{MAX} = \frac{Mn}{4}$, and corresponds to $\delta = \frac{M}{2}$. We refer to this as *Configuration* γ.

Assume there are no limits (other than M) on the number of threads that can be spawned or terminated in one barrier phase. Considering only *0-M* configurations, δ can take any value in the range $[\frac{M}{n}, \frac{M(n-1)}{n}]$ at a barrier, regardless of its previous value. Similarly, Δ can take any value in the range $[\frac{M(n-1)}{n}, \frac{Mn}{4}]$ at a barrier, depending only on δ and regardless of its previous value. However, to analyze the worst case behavior of any algorithm, it is sufficient to consider configurations α, β and γ, since these provide the worst case values of δ and Δ.

$c = t \cdot \delta$, with three choices of δ values: $\frac{M}{n}$, $\frac{M}{2}$ and $\frac{M(n-1)}{n}$. $C = y \cdot \Delta$, with just two choices of Δ values: $\frac{M(n-1)}{n}$ and $\frac{Mn}{4}$.

If the algorithm underestimates p, its cost is $p \cdot t \cdot \frac{M}{n} + y \cdot \frac{Mn}{4}$, while the adversary's cost is $p \cdot t \cdot \frac{M}{n}$. This makes the competitive ratio $r = 1 + \frac{yn^2}{4pt}$.

If the algorithm overestimates p, its cost is $p \cdot t \cdot \frac{M(n-1)}{n} + y \cdot \frac{M(n-1)}{n}$, while the adversary's cost is $y \cdot \frac{M(n-1)}{n}$. This makes the competitive ratio $r = \frac{pt+y}{y}$.

We propose an algorithm which selects $p = \frac{y}{t} \cdot \frac{n}{2}$, corresponding to configuration γ. By exhaustively considering all possible combinations of δ and Δ values, one can prove that any algorithm does best by choosing this value, thus showing that our algorithm is optimal among incomplete information alternatives. (The proof is omitted here due to lack of space.) By substituting the value of p in the equations for r above, we see that the competitive ratio of our algorithm is n.

5 Summary and Future Work

In this paper, we classify irregular parallel programs based on the source of their irregularity, into X-irregular, Y-irregular and completely irregular programs. Our programs are communication-intensive besides being completely irregular. This

limits us to fixed size partitions, since frequent communication is efficiently implemented using a hardware barrier tree. We compare the possible scheduling algorithms for completely irregular programs on fixed size partitions: multi-gang scheduling and local scheduling, and show that local scheduling does better to mitigate the inefficiencies of Y-irregularity. However, to handle the effects of X-irregularity, threads need to be rebalanced on processors periodically. We propose and analyze online algorithms for rebalancing threads, including an n-competitive algorithm which is efficient due to its low information gathering cost.

We intend to run simulations based on our program characteristics to experimentally show the following: Although our algorithm's worst case behavior is in $\Theta(n)$, its average behavior is fairly close to the performance of the 2-competitive, optimal algorithm which has a far greater information gathering overhead. Based on our experiments, we also intend to propose algorithms for machines where the rebalancing cost $C = x + y\Delta$ has a large value of x, making infrequent rebalancing advantageous.

References

1. R. M. Bryant and R. A. Finkel. A stable distributed scheduling algorithm. In *International Conference on Distributed Computing Systems*, April 1981.
2. D. G. Feitelson and M. A. Jette. Improved utilization and responsiveness with gang scheduling. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing - Lecture Notes in Computer Science*, volume 1291, pages 238–261. Springer Verlag, 1997.
3. D. G. Feitelson and L. Rudolph. Parallel job scheduling: Issues and approaches. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing - Lecture Notes in Computer Science*, volume 949, pages 1–18. Springer Verlag, 1995.
4. V. Ramakrishnan, I. D. Scherson, and R. Subramanian. Efficient techniques for fast nested barrier synchronization. In *Symposium on Parallel Algorithms and Architectures*, pages 157–164, July 1995.
5. V. Ramakrishnan, I. D. Scherson, and R. Subramanian. Efficient techniques for nested and disjoint barrier synchronization. *Journal of Parallel and Distributed Computing*, 58:333–356, August 1999.
6. D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208, February 1985.