Low Overhead Parallel Schedules for Task Graphs[*]

(Extended Abstract)

Richard J. Anderson
Paul Beame
Walter L. Ruzzo

Department of Computer Science and Engineering
University of Washington[†]

Abstract

We introduce a task scheduling model which is useful in the design and analysis of algorithms for small parallel machines. We prove that under our model, the overhead experienced in scheduling an $n \times n$ grid graph is $O(\log\log n)$ for p processors, $p \geq 2$. We also prove a matching lower bound of $\Omega(\log\log n)$ for p processors, $p \geq 2$. We give an extension of the model to cover the case where the processors can have varying speed or are subject to delay.

1 Introduction

In this paper we explore a task graph model for parallel programming. We were originally motivated to look at this model based on experience gained implementing algorithms on a small parallel machine [1]. The task graph model we consider naturally arises in programming and leads to efficient and general programs for a number of problems. It provides a fundamental abstraction with wide applicability to parallel computation. A task graph consists of a collection of computational tasks to be performed, together with precedence constraints on the order in which they may be executed. Task graph models have been used for a wide range of applications. For example, they arise in parallel program design, scheduling theory, and code optimization. Many existing algorithms can be conveniently translated into task graph form.

The task graph model has a number of practical advantages for implementing parallel algorithms. Representing a program as a task graph allows a high degree of parallelism to be simply expressed, while compartmentalizing all synchronization in a few scheduling routines. This greatly facilitates debugging and enhances portability. Further, it compartmentalizes knowledge of the number of available processors, making it easy to specify some programs for a range of processor and data sizes, and to make algorithms robust to variations in the number of available processors.

A key consideration in practice is whether implementing an algorithm with a task graph introduces an unacceptable overhead. Our results show that for some very important families of task graphs and parallel machines the overhead is negligible.

We consider overhead in a domain where communication is not a significant cost. This is a realistic assumption to make in the case of many small, shared memory parallel machines. On the other hand, we do not assume that synchronization is "free". This is also a realistic assumption

[*]This work supported by the NSF grants CCR-8657562, CCR-8703196, CCR-8858799, NSF/DARPA grant CCR-8907960 and Digital Equipment Corporation.

[†]Seattle, WA 98195. E-mail: anderson@-, beame@-, and ruzzo@cs.washington.edu.

on most MIMD machines, where synchronization and scheduling can easily be comparable in cost to single tasks in a fine-grain task decomposition.

In a task graph with no precedence constraints, it is trivial to attain perfect speedup in parallel execution: t unit-time tasks can be executed by p processors in time $\lceil t/p \rceil$ with no effort wasted on coordination, synchronization, or scheduling other than to establish an arbitrary pre-agreed partition of the tasks into p equal size groups. Precedence constraints introduce two distinct sources of overhead — time during which processors are idle (because too few unprocessed tasks have had their precedence constraints satisfied) and time performing synchronization or scheduling (to maintain the required ordering of task execution).

In the common case where there are many more tasks than processors, it is natural to aggregate tasks into *jobs* subject to the tasks' precedence constraints. This may greatly reduce the scheduling overhead since fewer, larger units of work need to be scheduled. However, it also could greatly increase the overhead due to idleness, since idle processors may need to wait longer for the completion of these larger units. At one extreme, minimal idleness (i.e. maximal parallelism, but also maximal scheduling cost) is likely to occur when each task is scheduled individually. At the other extreme, minimal scheduling cost (but also minimal parallelism) is attained by aggregating all tasks into a single large job, at the cost of leaving all but one processor idle for the duration of that job.

The overall problem we wish to solve is to determine, for a given task graph and number of processors, the best (lowest overhead) schedule. That is, we seek the best way to aggregate tasks into jobs, and to schedule those jobs, so as to minimize the sum of the two kinds of overhead. In the next section we propose a simple model for the costs of synchronization and idle time overhead, and for studying task graph scheduling.

The main example of a task graph that we consider is the *grid graph* — an n by n array of tasks, each of which is dependent on the prior execution of its two neighbor tasks to the left and above it in the grid. This is the graph underlying a variety of important algorithms including the Gauss-Seidel relaxation method in numerical linear algebra and many dynamic programming algorithms such as the Cocke-Kasami-Younger context-free language recognizer [2] and the longest common subsequence algorithm [6]. The grid graph is also a prototype for an interesting class of problems with "slowly growing" parallelism, in contrast to, say, a balanced binary tree or the FFT graph, where a high degree of parallelism is apparent and easily exploited.

Our results show that for $n \times n$ grid graphs, scheduling adds only $O(\log \log n)$ overhead on p processors, $p \geq 2$, where the constant of proportionality depends only on p, and that this is optimal. This overhead is remarkably low — for example, with 2 processors the overhead for grid graphs with over 350 million unit-cost tasks is at most 37 time units, accounting for an execution time at most 19 steps beyond that for perfect speed-up.

As mentioned above, our results were motivated by experience with small parallel machines such as the Sequent Symmetry [1]. Such machines form an important class of existing parallel computers and seem likely to remain a highly cost-effective alternative to conventional sequential machines. Although in some cases we were able to come very close to full speedup over a sequential algorithm, the existing theory of parallel algorithms did not serve as a useful guide.

The majority of the theoretical work on parallel algorithms does not adequately address this class of machines. The most common approach in developing parallel algorithms has been to view the number of processors as a quantity that varies with the problem size. The emphasis has been on discovering very fast algorithms (for example the class $\mathcal{NC}$) while using a number of processors that is very large. Work on designing "efficient" or "optimal" parallel algorithms, where the time–processor product is on the order of the sequential run time, potentially has application to small parallel machines. However, the constant factors that arise in designing the "efficient" algorithms are often quite large. The goal when using a p processor machine is to solve a problem p times faster than with a single processor. Constant factors can easily overwhelm the performance gain of parallelism when p is small. In addition, much of the theoretical work on parallel algorithms ignores the cost of synchronization. Small parallel machines are typically MIMD machines, so in practice processors are not fully synchronized. Synchronization costs in real algorithms have significant performance implications, and are often a much more serious factor than, say, contention for access to the shared memory.

One of our goals is to try to develop a theory that aids us in understanding algorithms for small parallel machines. We believe this work shows that it is possible to say theoretically interesting things about such algorithms, and about synchronization costs in particular.

The remainder of our paper is organized as follows. The next section explains our formal task scheduling model. Section 3 introduces some easy basic results about the model on a variety of task graphs. Section 4 proves that the overhead for the $n \times n$ grid graph for the 2 processor case is $O(\log\log n)$. In Section 5 we then sketch the extensions of this result to the p processor case. In Section 6 we give the lower bound that shows our algorithms are optimal. We conclude in Section 7 by discussing how the model can be extended to the case where the processors vary in speed or are subject to delay.

2 The Task Scheduling Model

2.1 Aggregation

The task graph model assumes that a problem's solution can be decomposed into a set of primitive computational *tasks*. Precedence constraints on the execution of these tasks may be necessary to achieve correctness. The task graph model represents the solution as a directed acyclic graph (dag), with the individual tasks corresponding to vertices, and the edges giving the precedence constraints.

It is natural to aggregate tasks into *jobs* subject to the tasks' precedence constraints. Each job is executed on a single processor by executing its component tasks in an arbitrary order consistent with the *intra*-job precedence constraints. Consistency with *inter*-job precedence constraints is the responsibility of the *scheduler*. A correct scheduler must ensure that the jobs are executed in an order that is consistent with the task graph, and must do this independently of the rate of processing of the individual jobs. Conceptually, we assume that a central scheduler calculates a beneficial aggregation of tasks into jobs and dynamically assigns the jobs to idle processors in a way that ensures consistency with the task graph.

We are not making any strong assumptions about the actual implementation of the "scheduler." Semaphores or a variety of other techniques could be used in place of central scheduling routines to achieve the desired effect. Conversely, there is no essential loss in generality in assuming the presence of a central scheduler, since other synchronization methods can be easily recast into this model: simply view the set of tasks executed by one processor between synchronization events as a "job." The key point is that some synchronization or scheduling method is necessary, they all introduce an amount of overhead that is roughly proportional to the number of jobs scheduled, and this overhead should be counted.

Different choices of aggregation into jobs can have radically different overheads. Furthermore, given an aggregation into jobs, the ordering on the jobs chosen by the scheduler may also effect the overhead. Thus the general problem we wish to solve is: given a task graph and number of processors, find an aggregation of tasks into jobs and an ordering of these jobs consistent with the task graph that together produce minimum overhead. It will turn out that the overhead of all the schedules we describe is completely insensitive to the ordering of jobs, so our schedules can be implemented using a very simple barrier synchronization technique.

In keeping with the idea that idleness and scheduling are the primary sources of overhead we study the following simple model of overhead in the task graph model. Each task is assumed to require one time unit to execute. Likewise, scheduling each job is assumed to require one time unit. (Changing the ratio between task execution time and scheduling time won't fundamentally alter our results.) *Overhead* is defined to be the total over all processors of the number of steps they spend idle plus the number of steps spent scheduling jobs. (The latter quantity is simply the total number of jobs.) Intuitively, overhead tells us the amount of CPU time that is used non-productively. It measures how close we have come to achieving perfect speedup; if we have p processors and n units of work to perform, overhead measures p times the difference between the schedule's execution time and n/p.

One subtlety here is that we are using an essentially synchronous cost model to estimate the performance of an asynchronous system. This is reasonable since processors, although not tightly synchronized, generally execute at very nearly the same rates. Hence a synchronous cost model will provide reasonably accurate performance estimates. A correct program, however, must be able to accommodate variations in execution rates due to a variety of factors, and so must explicitly synchronize as necessary to ensure that precedence constraints are respected. Hence our model charges for synchronization as well. At the end of the paper we extended the model to the case where the schedule must be dynamically adjusted to reflect the actual delays discovered during its execution.

2.2 Examples

To give some intuition for the problem, we will briefly consider some two processor decompositions for the $n \times n$ grid. Aggregation of tasks into $k \times k$ jobs is a very natural approach to try. It's easy to see that choosing $k = \sqrt{n}$ balances scheduling overhead against idle time, giving a total overhead of $\Theta(n)$; see Figure 1(a).

A second natural approach, based on a simple divide-and-conquer strategy, is sketched in Figure 1(b). Although it has succeeded in creating two large ($n/2 \times n/2$) independent jobs, it also creates a large number of small jobs, hence also yields only $\Theta(n)$ overhead.

A somewhat more complex refinement of the $k \times k$ strategy giving sublinear overhead is also possible; see Figure 1(c). To reduce the overhead, we must both reduce the number of jobs and avoid leaving any long periods of idle time. We reduce the number of jobs by breaking into bigger squares: we divide the grid into $n^{2/3}$ squares, each of size $n^{2/3} \times n^{2/3}$. To reduce the idle time associated with the large corner squares, we subdivide them into $n^{2/3}$ smaller squares of size $n^{1/3} \times n^{1/3}$. This approach (with attention to a few details), yields $n^{2/3}$ overhead. This method generalizes to give a solution with $2^{O(\sqrt{\log n})}$ overhead. We can do better still. Our main results are that overhead $O(\log\log n)$ is achievable for the grid graph, for any fixed number of processors, and that this is optimal. We found this result very surprising, since we did not expect that the overhead could be so small.

2.3 Formal Model

We model a computation as a dag G whose nodes are the primitive computation steps (tasks) to be performed and whose edges represent precedence between the tasks. A *schedule* S of G consists of a partition of the set of tasks of G into *jobs* $J_1, \ldots, J_m$ and a *schedule graph*, a dag with these jobs as nodes and with edges representing constraints on the order of execution of the jobs. The precedence constraints on the jobs must be consistent with the precedence of their constituent tasks. That is, for any jobs J_k, J_l in S there must be an edge from J_k to J_l whenever job J_k contains a task that is a predecessor of a task in job J_l. Note that the schedule graph may contain edges other than the edges enforcing task graph constraints. A schedule is a *p-processor* schedule if its schedule graph is the union of at most p (not necesssarily disjoint) chains of jobs. (That is, the maximum parallelism in the schedule is at most p.)

The *cost* of a job containing t tasks is $t+1$, accounting for the t time units to perform its constituent tasks and one time unit for scheduling. The *completion time* of schedule S is the length of the longest path in the schedule graph of S, where the length of a path is the sum of the costs of its jobs. The *overhead* of a p-processor schedule S is p times its completion time minus the number of tasks it contains. In other words the overhead of a schedule is simply the sum of the idle times of the processors plus the number of jobs in the schedule.

The *task graph scheduling problem* is to find, given a number of processors p and a task graph G, a p-processor schedule of G that minimizes overhead.

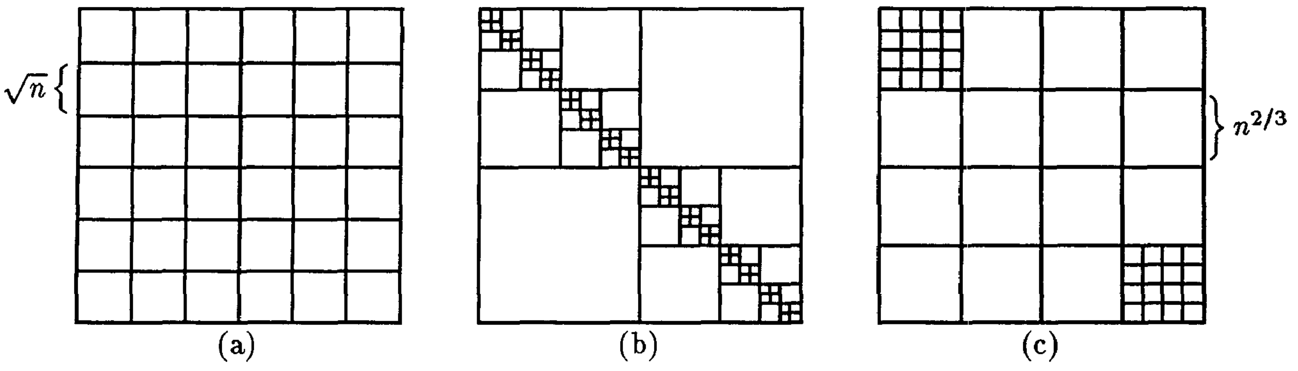

Figure 1: Grid Graph Decompositions

2.4 Related Work

Papadimitriou and Ullman [4] and Papadimitriou and Yannakakis [5] consider the parallel execution of task graphs, concentrating on some of the same graphs as we do. Both papers differ from ours in that they assume synchronous parallel execution. In a synchronous model the overhead for scheduling is not an issue and instead they concentrate on relationships between communication and time. Their task graphs contain edges representing data communication between tasks whereas ours only have the smaller number of edges needed to ensure precedence. For example, in their model, grid graphs are not applicable to dynamic programming because of the communication of earlier values in these problems. Papers on the optimization of nested loops [7], [3] also look at grid graphs, but do not present solutions similar to ours. Our results may be applicable in this context.

3 Basic Results

It should not be surprising that our basic problem is $\mathcal{NP}$-hard for $p \geq 2$ processors. The $\mathcal{NP}$-hardness result is not a serious obstacle to our study, since we are interested in studying families of graphs that arise from actual algorithms as opposed to arbitrary dags.

For several families of graphs it is very easy to get nearly optimal results. For example, we could take complete binary trees as our family of graphs. In the case where the number of processors is a power of two, a simple solution is to have one processor process the the top $\log_2 p$ levels, and then give each processor a subtree of size n/p. This leads to an overhead of roughly p (independent of n). If the number of processors in not a power of two, there is a little work to do in load balancing, but even in this case the overhead is proportional to p. It is possible to get very similar results for the FFT graph. The reason that the results for these cases are almost trivial is that the graphs are very shallow, and by performing a small amount of work, it is possible to generate very large independent jobs.

Throughout the rest of this abstract we will only consider (two-dimensional) grid graphs. In the full paper we will also present results on scheduling higher dimensional grids.

4 Upper Bound for 2 Processors

In this section we will sketch the scheduling method used to solve the $n \times n$ grid with overhead $O(\log \log n)$.

Theorem 4.1 *For any $n \geq 2$, an $n \times n$ grid graph can be scheduled with at most 9 units of idle time and at most $8 \log_2 \log_2 n + 20$ jobs.*

Proof: The key subproblem is to efficiently schedule an $n \times n$ grid with a $k \times k$ subgrid removed from both the upper left and lower right corners, for any even k, $2 \leq k \leq n/2$.

Given this procedure, the $n \times n$ grid problem is easily solved. In the first step, Processor A does the four tasks in the 2×2 square at the upper left corner. In the last step, Processor A does the 2×2 lower right corner. In between, the two processors

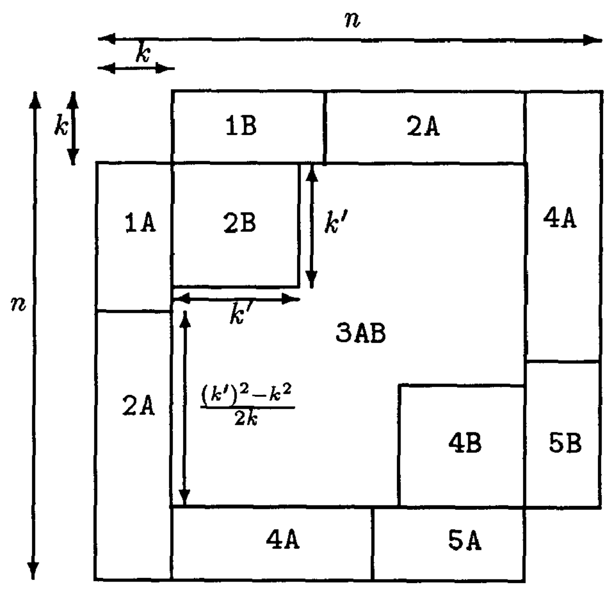

Figure 2: The Two Processor Schedule

cooperatively solve the problem for an $n \times n$ grid with its upper left and lower right 2×2 corners removed. This costs 8 units of overhead (Processor B is idle while Processor A does the corners), plus the overhead for the subproblem.

The key subproblem is handled in four phases plus the recursive solution of an $n' \times n'$ grid with $k' \times k'$ corners removed, where k' is roughly $\sqrt{2kn}$. Since the corners grow rapidly, after $O(\log\log n)$ recursive levels, the corners touch, leaving two completely independent subproblems to solve at the bottom level.

Figure 2 shows how the $n \times n$ decomposition is performed. The labels in the various regions indicate both the phase in which the region is processed, and the processor that does it. For example, the region labeled "1B" is the phase 1 job executed by Processor B, and the two disconnected regions labeled "2A" comprise the phase 2 job for Processor A. The two jobs in each phase are exactly the same size, so the solution provides perfect parallelism: there is no overhead due to idle processors, except for the 8 units at the top level for the upper left and lower right 2×2 corners, and one unavoidable unit in the base case when n is odd.

The sizes of the various regions shown in Figure 2 are determined as follows. The area of region 2B is obviously $(k')^2$. Region 2A is chosen to have the same area. Regions 1A and 1B are chosen to fill the remainder of the width k strips along the top and left of the grid, and so have length $n - 2k - ((k')^2 - k^2)/2k$. The phase 4 and phase 5 regions are exactly the same size and shape as the corresponding phase 2 and 1 regions, resp. The recursively solved region, 3AB, is an $n' \times n'$ grid with $k' \times k'$ corners removed, where $n' = n - 2k$.

The goal is to make the $k' \times k'$ region 2B as large as possible. There are two constraints on how large k' can be. The most significant is that it cannot extend past the ends of the 1A and 1B regions without violating precedence constraints. Hence, we must have

$$n - 2k - ((k')^2 - k^2)/2k \geq k'. \qquad (1)$$

Second, all the regions should be rectangles, as shown in Figure 2. (This is not necessary for correctness, and worsens our bounds by a small constant factor, but is preferred for simplicity of exposition, and is likely to be valuable in practice as well.) For the regions to be rectangular, it is necessary that $((k')^2 - k^2)$ be a multiple of $2k$; we will use the slightly simpler condition that k be even and k' be a multiple of k. Hence, we want the greatest integer l such that $k' = kl$ is a solution to the inequality (1). Thus

$$n - 2k - ((kl)^2 - k^2)/2k \geq kl.$$

Equivalently,

$$(kl)^2/2k + kl - n + 3k/2 \leq 0$$

$$l^2 + 2l - 2n/k + 3 \leq 0$$

Thus l is the greatest integer such that

$$l \leq \frac{-2 + \sqrt{4 + 8n/k - 12}}{2}$$

or

$$l = \left\lfloor \sqrt{2n/k - 2} - 1 \right\rfloor$$

and so it suffices to choose

$$k' = k \left\lfloor \sqrt{2n/k - 2} - 1 \right\rfloor.$$

i	n_i	k_i
0	100,000	2
1	99,996	630
2	98,736	10,080
3	78,576	30,240

Table 1: Decomposition Sequence for a Large Grid

Thus, we obtain the following recurrences for the sizes of the successive subproblems.

$$\begin{aligned} k_0 &= 2 \\ n_0 &= n \\ k_{i+1} &= k_i \left\lfloor \sqrt{2n_i/k_i - 2} - 1 \right\rfloor \quad \text{for } i \geq 0 \\ n_{i+1} &= n_i - 2k_i \quad \text{for } i \geq 0 \end{aligned}$$

Table 1 gives an example of the values of these sequences for $n = 100,000$.

Roughly speaking, this recursive decomposition continues until regions 2B and 4B touch, which happens when k/n exceeds $\approx .15$.

These recurrences can be bounded as follows. Let $a = .13$, $b = 1/(1+2a)$, and $c = b(\sqrt{1-a} - \sqrt{2a})^2 \approx .14$. Note that $c > a$. Let j be the least integer such that $k_j/n_j > a$. Then, for all $i < j$, we claim:

$$\begin{aligned} k_i/n_i &\leq a & (2) \\ bn &\leq n_i \leq n & (3) \\ 2cn^{1-2^{-i}} &\leq k_i \leq 2n^{1-2^{-i}} \quad \text{and,} & (4) \\ j &\leq \log_2 \log_2 n. & (5) \end{aligned}$$

Inequality (2) is immediate from the definition of j. Inequalities (3)–(5) are shown as follows. First note

$$\left\lfloor \sqrt{2n_i/k_i - 2} - 1 \right\rfloor \geq \left\lfloor \sqrt{2/a - 2} - 1 \right\rfloor \geq 2.$$

Thus the k_i's are at least doubling with each step up to the j-th, and so for all $i < j$ we have

$$n_i = n - 2 \cdot \sum_{m=0}^{i-1} k_m \geq n - 2 \cdot k_i \geq n - 2 \cdot an_i,$$

which implies (3).

Inequality (4) is shown by induction on i. The basis is straightforward. For the upper bound induction step, note

$$\begin{aligned} k_{i+1} &= k_i \left\lfloor \sqrt{2n_i/k_i - 2} - 1 \right\rfloor \\ &\leq k_i \sqrt{2n_i/k_i} \\ &= \sqrt{2n_i k_i} \\ &\leq \sqrt{2nk_i} \\ &\leq \sqrt{2n 2n^{1-2^{-i}}} \\ &= 2n^{1-2^{-(i+1)}}. \end{aligned}$$

For the lower bound

$$\begin{aligned} k_{i+i} &= k_i \left\lfloor \sqrt{2n_i/k_i - 2} - 1 \right\rfloor \\ &\geq k_i \left(\sqrt{2n_i/k_i - 2} - 2 \right) \\ &= \sqrt{2n_i k_i - 2k_i^2} - 2k_i \\ &= \sqrt{2n_i k_i} \left(\sqrt{1 - k_i/n_i} - \sqrt{2k_i/n_i} \right) \\ &\geq \sqrt{2\,bn\,2cn^{1-2^{-i}}} \left(\sqrt{1-a} - \sqrt{2a} \right) \\ &= 2n^{1-2^{-(i+1)}} \sqrt{bc} \left(\sqrt{1-a} - \sqrt{2a} \right) \\ &= 2cn^{1-2^{-(i+1)}}. \end{aligned}$$

The fifth step uses Inequalities (2), (3) and the induction hypothesis. The last step relies on the choice of c.

Inequality (5) is easily seen by contradiction. If it does not hold, then for $i = \log_2 \log_2 n$, Inequalities (4) and (3) hold for i, so we have

$$\begin{aligned} k_i/n_i &\geq k_i/n \geq 2cn^{1-2^{-i}}/n = 2cn^{-2^{-i}} \\ &= 2cn^{-2^{-\log_2 \log_2 n}} = 2cn^{-(1/\log_2 n)} = c > a, \end{aligned}$$

contradicting the minimality of j.

This recursive decomposition continues until k is a large enough fraction of n that regions 2B and 4B can be made to just touch (or overlap by one unit if n is odd). Namely, the base case is reached when $k' = \lceil n/2 \rceil - k$ is a solution to inequality (1). Except for small, odd n, this point is reached when k/n exceeds $\approx .15$. In the base case the schedule can be completed with at most 10 jobs (and 1 unit of idle time if n is odd) using a decomposition similar to the one used above. (The details are omitted from this abstract.) Unlike the recursive decomposition presented above, in the base case the jobs might include "L"-shaped regions. If desired, these regions could be split into rectangles, with a small increase in overhead.

To finish the analysis, observe that in $j \leq \log_2 \log_2 n$ steps, k_i has grown to be at least $.13n_i$. If the base case hasn't already been reached, then in one more step, while it is possible that k_i won't increase, n_i will decrease enough that $k_{j+1}/n_{j+1} > .28$, and the base case will be reached. This gives at most $l = \log_2 \log_2 n + 2$ recursive levels (including the base case). The total overhead for l levels is $8l + 4$ jobs (8 per level, plus two extra for the last level, plus two for the top level) with 8 units idle time for the top level, and one unit in the base case if n is odd. For all $n \geq 2$ this totals at most $8 \log_2 \log_2 n + 29$, as claimed. ▮

We also have computed the number of recursive levels exactly for many values of n and find that 2 levels suffice for all $n \leq 138$, 3 levels suffice for all $n \leq 19{,}260$, and 4 levels suffice for many (perhaps all) $n \leq 300$ million. Clearly, the overhead will be negligible for all but the smallest problems of practical size.

It is worth noting that except for a few jobs at the base level, all jobs are simple, rectangular regions. Hence, we expect the method will be simple and practical to implement.

5 Upper Bound for p Processors

In this section we will give a very brief sketch of the ideas underlying the $O(\log \log n)$ upper bound for $p > 2$ processors. Although somewhat more complicated than the two processor case, the main idea is the same: work on narrow strips along the side of the grid until a large "corner" can be removed. In this case it is most convenient to remove an antidiagonal corner rather than a square one, and it is necessary to work on about $p/2$ narrow strips, of differing widths, rather than just one.

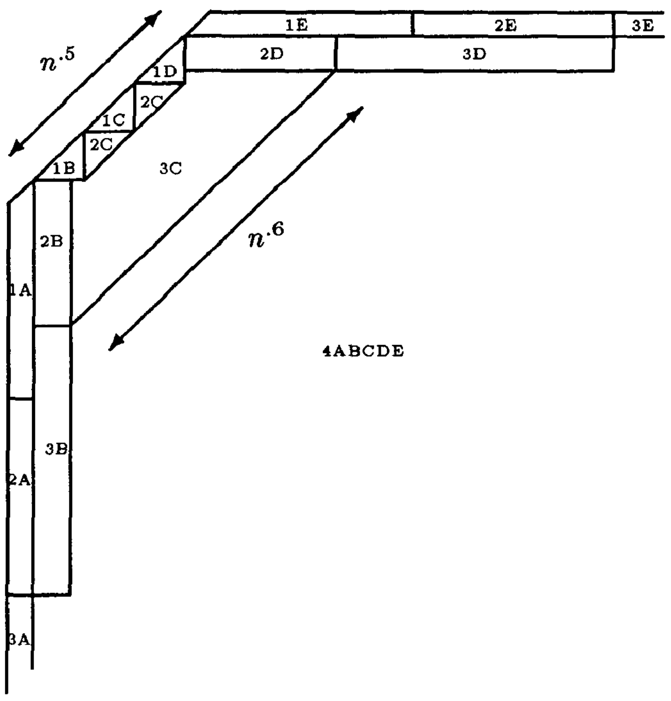

Figure 3: Part of a Five Processor Schedule

Figure 3 illustrates the schedule used when $p = 5$, starting from a grid with an $n^{.5}$ corner cut off. In phase 1, Processors B, C and D complete triangles of area $\Theta(n)$, while A and E do $\Theta(n^{.2} \times n^{.8})$ strips. In phase 2, A, C and E repeat this, while B and D do $\Theta(n^{.4} \times n^{.6})$ strips. In the third phase, C clips off a corner of size $\Theta(n^{.6} \times n^{.6})$ while A and E do $\Theta(n^{.2} \times n^{1})$ strips, and B and D do $\Theta(n^{.4} \times n^{.8})$ strips.

The widths of the strips need to be chosen so that the longest one fits within the grid, and they need to increase in width so that, e.g., B doesn't overtake A in the last phase. Also, in the general case, we won't be starting from a "clean" antidiagonal corner of width k. Rather, there will be partially completed strips of width $o(k)$ passed down from higher levels in the recursion. This also complicates the base case. Nevertheless, when appropriately parameterized, these ideas can be extended to give the desired $O(\log \log n)$ schedule for any constant p. Details are deferred to the full paper.

6 Lower Bound

The following lower bound shows that the schedules of the previous sections are essentially the best possible. There is a strong connection between the intuition of those algorithms and the intuition for the proof of the lower bound.

Theorem 6.1 *Any schedule for the $n \times n$ grid graph with $p \geq 2$ processors has overhead at least $\Omega(\log\log n)$.*

Proof: Fix some schedule for the $n \times n$ grid graph. Define *diagonal job* J_i to be the i-th job in the schedule that involves a task $t(j,j)$ on the diagonal of the grid. Let s_i be the largest index j of a task $t(j,j)$ in job J_i. We claim roughly that either s_i grows slowly so that there are at least $\log\log n$ diagonal jobs or a processor is idle for at least $\log\log n$ units of time up to the end of job J_i.

More precisely let

$$\begin{aligned} t_1 &= \sqrt{\log\log n} \\ t_i &= t_{i-1} + \sqrt{2nt_{i-1} + \log\log n} \quad \text{for } i > 1. \end{aligned}$$

Claim: If for any i, $s_i \geq t_i$ then a processor is idle for at least $\log\log n$ units of time up to the end of job J_i.

Suppose that $\exists i,\ s_i \geq t_i$. Let i be the smallest such value. We have two cases:

Case 1: $i = 1$ The first job J_1 includes $t(1,1)$ and during the execution of J_1 any other processors are idle. If $s_1 \geq t_1 = \sqrt{\log\log n}$ then J_i must contains more than $\log\log n$ tasks. Thus the other processor(s) are idle for at least $\log\log n$ steps.

Case 2: $i > 1$: Since i was the smallest value such that $s_i \geq t_i$, $s_{i-1} < t_{i-1}$. Because of the precedence enforced by the grid graph and the definition of job J_i, no job other than J_i that begins prior to the completion of job J_i can involve any task $t(j,k)$ where both j and k are larger than $s_{i-1} < t_{i-1}$. Thus all jobs that run in parallel with job J_i must involve tasks $t(j,k)$ that have either j or k less than t_{i-1}. Using a very crude upper bound we see that there are at most $2nt_{i-1}$ such tasks. However the fact that $s_i \geq t_i$ implies that $s_i - s_{i-1} > t_i - t_{i-1} = \sqrt{2nt_{i-1} + \log\log n}$. Note that for $i > 1$, job J_i contains at least $(s_i - s_{i-1})^2$ tasks since it must contain all tasks in the square bounded by $t(s_{i-1}+1, s_{i-1}+1)$, $t(s_i, s_i)$, $t(s_{i-1}+1, s_i)$, and $t(s_i, s_{i-1}+1)$. Thus J_i must contain at least $2nt_{i-1} + \log\log n$ tasks. Since at most $2nt_{i-1}$ can be run in parallel with J_i, at least one other processor is idle for at least $\log\log n$ tasks.

The claim follows. Now since each unit of time during which an individual processor is idle contributes a unit of overhead, the proof is finished once it is shown that $t_i \geq n$ implies i is $\Omega(\log\log n)$ since this implies $\Omega(\log\log n)$ overhead due to the initiation of the diagonal jobs.

It is easy to see that for $i > 1$, $t_i \leq 2\sqrt{nt_{i-1}}$. Then an easy induction yields $t_{i+1} \leq 2^{2-2^{-i}} t_1^{2^{-i}} n^{1-2^{-i}}$ for all $i \geq 0$. Substituting $t_1 = \sqrt{\log\log n}$ and solving for i shows that $t_i \geq n$ implies that i is at least $\Omega(\log\log n)$ as required. ∎

7 Extended Model for Processor Delays

Up to now, our analysis has assumed that the computation is synchronized and all processors proceed at the same speed. However, there are many factors that can influence the rate that processors proceed and barring explicit synchronization we cannot expect that the rate of work will be uniform. For example, on the microscopic level, processors can be delayed by hardware interrupts, by performing tasks that require several extra machine instructions, and by delays in communication. On the macroscopic level some processors may run slower than others, and a job may be swapped out so that a processor executes another user's job. Since there are many sources of delay, we are not able to make any assumptions about how delay occurs, such as assuming that the processing times of jobs are independent random variables. We instead model delay by allowing an adversary to set the delay, so that our results apply in the worst case.

Our delay model is to assume that there are D units of delay available. An adversary decides when these units of delay are used. If a job usually takes t units of time, the adversary may decide that it takes $t + s$ units of time, for $0 \leq s \leq D$, expending s units of the adversary's delay. There is no restriction on which jobs the adversary may delay, other than the restriction bounding the total amount of delay that the adversary can apply. In this model, we allow more than one processor to be assigned to a job.* This means that the adversary cannot force all processors to idle by blocking a single job. We keep the same measure of performance for an algorithm: the number of jobs plus the amount of wasted computation, where wasted computation consists either of idleness or having more than one processor work on a job.

Since the adversary has D units of delay available, it can cause up to pD units of computation to be wasted if all processors must wait for the delayed job. The goal is to reduce the overhead to as close to D as is possible. We can show that the adversary can force some additional delay, although the additional overhead is asymptotically less than D.

Theorem 7.1 *Scheduling an $n \times n$ grid graph with delay D requires overhead $D + \Omega(D^{1/2})$ for 2 processors.*

Sketch: We show that an adversary can cause the algorithm to spend $\Omega(D^{1/2})$ time with both processors working on the same jobs. As soon as both processors are working on jobs inside the lower right $D^{1/2} \times D^{1/2}$ corner, we delay processor A. We delay the processor until processor B starts on the job that the A was working on and then release processor A, so that both of the processors execute the job. This results in at least $D^{1/2}$ duplicated work. ▮

Theorem 7.2 *An $n \times n$ grid graph can be scheduled in spite of delay D with overhead $D + O(D^{3/4})$ when $D > \log n$ on 2 processors.*

Sketch: The main idea is to decompose the upper left and lower right corners into $O(D^{3/4})$ square jobs. If the entire grid were decomposed into $D^{1/4} \times D^{1/4}$ square jobs, then an algorithm that assigned a second processor to a delayed job if there were no other available jobs, or if the job had been delayed $D^{3/4}$ time units would lead to a schedule where the amount of duplicated work was $O(D^{3/4})$. However, the total number of jobs would be too large. The solution is to use $D^{1/4} \times D^{1/4}$ jobs close to the corners, and progressively larger jobs closer to the center, so as to keep the total number of jobs $O(D^{3/4})$. ▮

*A technical issue arises as to whether or not we require all processors that start a job to finish the job. The upper and lower bounds we have apply to both cases.

8 Conclusions

We have introduced a task scheduling model that applies to small parallel machines. Our main technical result is that it is possible to schedule grid graphs with remarkably small overhead. The most interesting directions for future work are to tighten and extend the results in the model with processor delay.

References

[1] R. J. Anderson and D. D. Chinn. An experimental study of algorithms for shared memory multiprocessors. Extended Abstract, July 1989.

[2] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[3] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 319–329, 1988.

[4] C. H. Papadimitriou and J. D. Ullman. A communication-time tradeoff. *SIAM Journal on Computing*, 16(4):639–646, 1987. Preliminary version appeared in 25th Symposium on Foundations of Computer Science.

[5] C. H. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. *SIAM Journal on Computing*, 19(2):322–328, 1990. Preliminary version appeared in 20th ACM Symposium on the Theory of Computing.

[6] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, Jan. 1974.

[7] M. Wolfe. Advanced loop interchanging. In *International Conference on Parallel Processing*, pages 536–543, 1986.