PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT. © ICOT, 1988

PARALLEL APPROXIMATION ALGORITHMS

Ernst W. Mayr*

Department of Computer Science
Stanford University, California 94305, USA

ABSTRACT

Many problems of great practical importance are hard to solve computationally, at least if exact solutions are required. We survey a number of ($\mathcal{NP}$- or $\mathcal{P}$-complete) problems for which fast parallel approximation algorithms are known: The 0-1 knapsack problem, binpacking, the minimal makeshift problem, the list scheduling problem, greedy scheduling, and the high density subgraph problem. Algorithms for these problems are presented highlighting the underlying techniques and principles, and several types of *parallel approximation schemes* are exhibited.

1 Introduction

Many problems of great practical importance are computationally very difficult and seem to require ever larger computing power. Advances in modern circuit technology have made parallel computation an intriguing possibility, pushing the limits of what can practically be computed. Also, the theory of computational complexity has established that the complexity of a large number of practically relevant problems, including many combinatorial optimization problems, is intrinsically related (in the sense that if one of them can be solved in polynomial time, so can all). Since so far, despite of immense efforts, no efficient algorithm has been found for any of these so-called $\mathcal{NP}$-complete problems, many people take $\mathcal{NP}$-completeness as evidence that no practically feasible solutions exist.

In a similar vein, the fact that a certain problem is $\mathcal{P}$-complete (*i.e.*, complete under logspace reductions for the class $\mathcal{P}$ of problems solvable in polynomial time) is commonly interpreted to mean that the problem cannot be solved by efficient parallel algorithms. Quite often, however, even though it may not be possible to solve $\mathcal{NP}$- or $\mathcal{P}$-complete problems efficiently in practice (by sequential respectively parallel algorithms), good approximations to the exact or optimal solution can indeed be found.

Thus, there are at least two motivations to study parallel approximation algorithms. The first is to speed up sequential, polynomial time approximation schemes for some $\mathcal{NP}$-hard optimization problems of practical importance. The other is to find good approximate solutions fast in parallel for problems that most likely cannot be solved exactly by efficient parallel algorithms since they are $\mathcal{P}$-complete. In this paper, we discuss parallel approximation schemes for both types of problems.

We shall base most of our discussions onto a theoretical machine model for parallel computation called the *Parallel Random Access Machine*, or *PRAM* (Fortune and Wyllie 1978). In this model, there is an unbounded number of identical *Random Access Machines* (or *RAM*'s), and an unbounded number of global, shared memory cells. The processors work synchronously, controlled by a global clock. Each processor can access any memory cell in one step. The *concurrent read exclusive write* variant of the model (CREW-PRAM) allows that more than one processor read the same memory cell in one step, but it disallows *concurrent writes* to the same memory cell. The *exclusive read exclusive write* variant (EREW-PRAM), on the other hand, forbids concurrent access completely. Most of the algorithms discussed in this paper run on the second, weaker model.

The shared memory feature of the PRAM model is somewhat idealistic. A more realistic machine model consists of a *network* of (identical) processors with *memory modules* attached to them. The processors are connected via point-to-point communication channels. Each processor can directly access only cells in its own memory module, and it has to send messages to other processors in order to access data in their modules. To respect technological constraints, the number of channels per processor is usually bounded or a very slowly growing function of the number of processors. Examples for such networks of processors are the Hypercube (Seitz 1985), the Cube-Connected-Cycles network (Preparata and Vuillemin 1979), or the Ultracomputer (RP3) (Pfister 1985, Schwartz 1980).

The parallel complexity class $\mathcal{NC}$ of problems that can be solved in time polylogarithmic in the problem size

*The author was partly supported by NSF grant DCR-8351757.

on a PRAM using a polynomial number of processors is commonly thought to characterize the class of efficient parallel algorithms (Pippenger 1979). The class $\mathcal{NC}$ is robust in the sense that it doesn't change when we replace the PRAM by one of the more realistic models mentioned above. It is also quite easy to see that $\mathcal{NC}$ is a subset of $\mathcal{P}$. Since $\mathcal{NC}$ computations can be simulated by sequential Turing machines using only polylogarithmic space, membership of a $\mathcal{P}$-complete problem in $\mathcal{NC}$ would imply that $\mathcal{P}$ is contained in POLYLOGSPACE which is considered very unlikely. This is why $\mathcal{P}$-completeness of a problem is usually considered as evidence that the problem cannot be efficiently parallelized.

Many of the algorithms and results in this paper have appeared elsewhere, as given in the reference section. We have tried to present different types of parallel approximation schemes, with the amount of resources depending in different ways from the problem size and the required accuracy. Similar to the notion of *(fully) polynomial approximation schemes* in sequential computation, we define an $\mathcal{NC}$ *approximation scheme* to be a family of algorithms parametrized by ϵ, the required accuracy, such that, for ϵ fixed to any value > 0, the resulting algorithm is in $\mathcal{NC}$. Such a family of algorithms is called a *full* $\mathcal{NC}$ *approximation scheme* if the running time of every algorithm in the family is bounded by a polynomial in the logarithms of the input size and ϵ^{-1}, and the number of processors each algorithm uses is bounded by a polynomial in the input size and ϵ^{-1}.

In this paper, we have also tried to categorize parallel approximation algorithms in the literature by the underlying approach, thus extracting some commonality for some of them and exhibiting some more widely usable techniques. Of course, we do not, and cannot, claim any completeness since research into parallel approximation algorithms is quite active and a certain selection had to be made. The remainder of the paper is organized as follows: Section 2 discusses approximation algorithms based mainly on discretization; we present a full $\mathcal{NC}$ approximation scheme for the 0-1 knapsack problem, and $\mathcal{NC}$ approximation schemes for binpacking and the makespan problem. In section 3, we discuss the parallel complexity of list scheduling which has a full $\mathcal{NC}$ approximation scheme based on scaling the execution times of the tasks. Section 4 presents parallel approximation algorithms based on combinatorial properties. Here, we discuss cases where the achievable accuracy of approximation is bounded away from 1 unless, say, $\mathcal{P} = \mathcal{NC}$. Section 5 contains some concluding remarks and open problems.

2 Discretize to Approximate

2.1 The 0-1 Knapsack Problem

In this section, we study several parallel approximation algorithms where the principal underlying paradigm is the reduction of many possible values to a few, depending on the required quality of the approximation. The first problem we discuss is the *0-1 knapsack problem:* We are given n items of weight $w_1, \ldots, w_n > 0$, with associated profits $p_1, \ldots, p_n > 0$, and a bound C. We are supposed to pack a subset of the items into a knapsack of capacity C (*i.e.*, the total weight of the packed items must not exceed C), in such a way that the profit associated with the packed items is maximized. The knapsack problem is a well-known combinatorial optimization problem. It is $\mathcal{NP}$-complete, and we therefore expect its exact solution to be practically infeasible (Garey and Johnson 1974).

Among the problems discussed in this section, the knapsack problem seems to have the most structure and thus allow the most efficient scheme for finding an approximate solution. A number of approximation algorithms has been presented for it and the related subset sum problem in the literature, many based on ideas in (Ibarra and Kim 1975). We also refer the reader to (Lawler 1979), (Peters and Rudolph 1984), and (Gopalakrishnan et al. 1986). All these algorithms find, for a given $\epsilon > 0$, a feasible packing of the knapsack whose associated profit is at least $(1-\epsilon)$ times the optimum.

We use the following notation. For $S \subseteq \{1, \ldots, n\}$, $w(S) = \sum_{i \in S} w_i$ is the total weight of the items given by S, and $p(S) = \sum_{i \in S} p_i$ their total associated profit. A set $S \subseteq \{1, \ldots, n\}$ is called *feasible* if $w(S) \leq C$. P^* is the optimal profit over all feasible solutions, and S^* a feasible set of items resulting in profit P^*.

The most efficient approximation schemes for the 0-1 knapsack problem (whether sequential or parallel) are based on three main ideas:

1. Let S be any subset of $\{1, \ldots, n\}$, and let C_S be the sum of the weights of the items from S in some optimal solution for the given instance of the knapsack problem. Then the subset of items from S picked in an optimal solution maximizes the associated profit subject to the condition that its total weight is at most C_S. Thus, *assuming* that we know C_S, we can independently optimize on S.

2. The possible profits are discretized to a "small" grid of possible values. For all these values, and for appropriate subsets of all the items, optimal solutions are computed acccording to 1.

3. Let $S, S' \subseteq \{1, \ldots, n\}$ be such that

$$p(S) \geq p(S') \text{ and } w(S) \leq w(S').$$

Then, in any solution $S' \cup S''$ with S'' disjoint from

S and S', S' can be replaced by S, increasing the resulting profit without increasing the weight. We say that S *dominates* S'.

We now discuss the arguments underlying the discretization approach. Assume without loss of generality that the items are numbered in non-increasing order of *profit density*, *i.e.*

$$\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \ldots \geq \frac{p_n}{w_n},$$

and that all $w_i \leq C$. Find the maximal $k \leq n$ such that

$$w_1 + \ldots + w_k \leq C.$$

By assumption, $k \geq 1$, and either $k = n$ in which case the optimal profit P^* is clearly $\sum_{i=1}^{n} p_i$, or $k < n$ and

$$\hat{P} = \sum_{i=1}^{k} p_i \leq P^* < \sum_{i=1}^{k+1} p_i \leq 2\hat{P}$$

since the first $k+1$ items maximize the profit density but exceed the capacity, and since $w_{k+1} \leq C$ and thus $p_{k+1} \leq \hat{P}$.

Given an accuracy $\epsilon > 0$ we round all profits p_i to the next lower integer multiple of $\frac{\epsilon \cdot \hat{P}}{n}$:

$$\tilde{p}_i = \frac{\epsilon \cdot \hat{P}}{n} \left\lfloor \frac{n \cdot p_i}{\epsilon \cdot \hat{P}} \right\rfloor.$$

Theorem 1 *Let $\tilde{P}^*$ be the optimal profit for the knapsack problem with weights w_i and profits $\tilde{p}_i$. Then*

$$(1-\epsilon) \cdot P^* \leq \tilde{P}^* \leq P^*.$$

Proof: Only the first inequality requires proof. Let $\tilde{S}^*, S^* \subseteq \{1, \ldots, n\}$ be such that $\tilde{p}(\tilde{S}^*) = \tilde{P}^*$ and $p(S^*) = P^*$. Since there are at most n items in a solution, and since $\tilde{P}^*$ is optimal, we have

$$\tilde{P}^* = \tilde{p}(\tilde{S}^*) \geq \tilde{p}(S^*) \geq p(S^*) - |S^*| \cdot \frac{\epsilon \cdot \hat{P}}{n} \geq (1-\epsilon)P^*.$$

□

For the 0-1 knapsack problem with weights w_i and profits $\tilde{p}_i$, all feasible solutions have a profit value which is one of the first $m = \lfloor \frac{2n}{\epsilon} \rfloor$ integer multiples of $\frac{\epsilon \cdot \hat{P}}{n}$. For all of these m values, we compute (if they exist) feasible solutions on items $1, \ldots, n$ by recursively computing feasible solutions on items $1, \ldots, \lfloor \frac{n}{2} \rfloor$ and on $\lfloor \frac{n}{2} \rfloor + 1, \ldots, n$, and combining these solutions. According to the first fundamental property stated above, we only keep, for each profit value that occurs, a feasible solution with minimal weight: This solution dominates all other solutions with the same profit value, which therefore can safely be discarded. It is clear how to compute the array of possible profit values, together with a minimal weight solution set for a given profit, in the base case of one item. In the recursive step, two such arrays of length m are easily combined forming their "cross-product" and eliminating solutions which are dominated by other feasible solutions with the same weight.

Clearly, there are $\log n$ levels to the recursion. Every combining step (and the base step) can be carried out using m^2 processors of an EREW-PRAM, in time $O(\log m)$. Since at most $\frac{n}{2}$ instantiations of the recursive procedure are ever carried out in parallel, we obtain

Theorem 2 *Given $\epsilon > 0$ and an instance of the 0-1 knapsack problem, a solution with a profit at least $(1-\epsilon)$ times optimal can be found in time $O(\log n(\log n + \log \frac{1}{\epsilon}))$ on an EREW-PRAM with $\frac{n^3}{\epsilon^2}$ processors.*

In (Gopalakrishnan et al. 1986), an interesting variation of the combining step is shown that allows to trade a factor of $\sqrt{\frac{n}{\epsilon}}$ in the number of processors for a $\log n$ factor in time.

2.2 The Binpacking Problem

Another classical optimization problem is *binpacking:* Given n items of size $0 < s_1, \ldots, s_n < 1$, these items are to be packed into as few bins of unit size as possible. Again, this problem is $\mathcal{NP}$-complete (Garey and Johnson 1974). The binpacking problem apparently has less structure than the knapsack problem since it is not clear how to treat disjoint sets of items independently. However, "efficient" parallel approximation schemes for the binpacking problem are also based on discretization. The algorithm presented here is a rather straightforward parallelization of the algorithm given in (Fernandez de la Vega and Lueker 1981).

Let $\epsilon > 0$ be given as the required accuracy, and let $B(\mathcal{A})$ and B^* denote the number of bins used by algorithm $\mathcal{A}$ and in an optimal solution, respectively, for a given instance B of the binpacking problem. We shall discuss a parallel approximation algorithm $\mathcal{A}$ that finds a solution with

$$B(\mathcal{A}) \leq (1+\epsilon)B^* \text{ for } B^* \rightarrow \infty.$$

First, we temporarily remove all items of size $< \epsilon$. Let n' be the number of remaining items, and assume without loss of generality that their sizes are $s_1 \leq s_2 \leq \ldots \leq s_{n'}$. Also, let $m = \lceil \epsilon^{-2} \rceil$, $m' = \lfloor \epsilon^{-1} \rfloor$, $k = \lfloor n'/m \rfloor$, and $r = n' \bmod k$. We divide the n' items into (roughly) m sets, the first, S_0, consisting of the r smallest of the s_i (ties broken arbitrarily), then $S_1, S_2, \ldots, S_m$, each consisting of the k next larger elements. Note that S_0 may be empty. Using parallel selection (Vishkin 1987), the sets S_i can easily be determined in time $O(\log n)$ on an EREW-PRAM with $n\epsilon^{-2}$ processors.

Let u_i be the maximum of the sizes in S_i, and let U_i consist of $|S_i|$ copies of u_i, for $i = 0, 1, \ldots, m$. We approximate the original bin packing problem (without the items of size $< \epsilon$) by the one given by the items in the sets $U_0, U_1, \ldots, U_m$, and we solve this problem

exactly, as follows. Since the U_i together contain at most $m+1$ distinct values, and since each of these values is $\geq \epsilon$, each bin can only be packed in a limited number of ways: Call a packing of a unit size bin to be of *type* $t = (t_0, t_1, \ldots, t_m)$ if it contains t_i items from U_i (of size u_i). Of course, $\sum_{i=0}^{m} t_i u_i \leq 1$ and $\sum_{i=0}^{m} t_i \leq m'$. Thus, there are at most

$$\binom{m+m'+1}{m+1} = \binom{m+m'+1}{m'}$$

different types. We also define a *partial packing* to be any vector $(p_0, p_1, \ldots, p_n)$ consisting of nonnegative integers $p_i \leq k$. The number of different partial packings is bounded by $(k+1)^{m+1}$.

We define an auxiliary digraph, which we call the *structure graph* associated with the problem, in the following way. The nodes of the digraph are the partial packings. There is an arc from a partial packing p to a partial packing p' whenever $p' - p$ is the type of a packing of a unit size bin. Note that, for fixed ϵ, the size of the structure graph is polynomial in n. Also, it is quite easy to see that an optimal packing of the given items can be read of a shortest path in the structure graph from node $(0, 0, \ldots, 0)$ to node $(r, k, \ldots, k)$. On an EREW-PRAM, and for fixed $\epsilon > 0$, this optimal packing can be determined in $O(\log^2 n)$ time, using a polynomial number of processors. The constants involved, however, are rather big: for the time, there is an ϵ^{-2} factor, and for the number of processors, the degree of the polynomial also contains such a factor.

To complete the packing, we still have to deal with the items smaller than ϵ, which were discarded initially. They are $s_{n'+1}, \ldots, s_n$. Assume that the packing obtained so far uses q bins, and that the space still available in these bins is $\delta_1, \ldots, \delta_q$. The idea for the following algorithm is to fill these bins, in order, to at least $1-\epsilon$, using the small items, also in order. In addition, we append $n-q$ empty bins to the list of partially filled bins. Thus, $\delta_{q+1} = \ldots = \delta_n = 1$. Note that n bins certainly suffice to pack all of the items.

Using a parallel prefix routine (Ladner and Fischer 1980) we determine the partial sums $\sum_{j=1}^{i} s_{n'+j}$, for $i = 1, \ldots, n-n'$. We then define another auxiliary graph, whose vertices are labelled by the pairs in $\{1, \ldots, n\} \times \{1, \ldots, n-n'\}$. Consider a vertex labelled (h, j). *Assuming* that $s_{n'+j}$ is the *first* of the small items to be placed into bin h by the (sequential) filling procedure alluded to above, we compute how many of the small items, starting at $s_{n'+j}$, are needed to fill bin h to at least $1-\epsilon$. Using binary search on the prefix sums, this can be done in $O(\log n)$ time with $n/\log n$ processors. We thus obtain the index $n'+j'$ of the first item that will go into bin $h+1$, or we find that all items fit into the first h bins. In the first case, we create, in the auxiliary graph, an arc from vertex (h, j) to vertex $(h+1, j')$. All such arcs can be computed in parallel, using $n^3/\log n$ processors. Since every vertex in the auxiliary graph has outdegree at most one, we can then use tree traversal techniques (requiring $O(n^2)$ processors and $O(\log n)$ time) to find the maximal path starting at vertex $(1, 1)$. The second component of the label of each vertex on this path gives the start of the segment of small items that are packed into the bin given by the first component.

Theorem 3 *Let $\epsilon > 0$ be fixed. There is a parallel algorithm $\mathcal{A}$ that solves any binpacking problem B such that*

$$B(\mathcal{A}) \leq (1+\epsilon)B^* \text{ for } B^* \to \infty.$$

Let n be the input size of B. Then the algorithm runs in time $O(\log^2 n)$ on an EREW-PRAM with a number of processors bounded by a polynomial in n.

Proof: Consider the algorithm $\mathcal{A}$ given above. We still need to establish the quality of the approximation it computes. For this analysis, we distinguish two cases.

First, we assume that the routine filling in the small items ends up filling to at least $1-\epsilon$ all but possibly one partially filled bin. If B denotes the given instance of the binpacking problem, we have

$$(B(\mathcal{A})-1)(1-\epsilon) \leq \sum_{i=1}^{n} s_i \leq B^*,$$

and hence

$$B(\mathcal{A}) \leq \frac{1}{1-\epsilon}B^* + 1.$$

For the second case, the filling routine packs all small items into the first q bins. Let $\hat{B}$ denote the problem instance given by the sets $U_0, U_1, \ldots, U_m$. Further, let D_i consist of $|S_i|$ copies of the minimum size element in S_i, again for $i = 0, 1, \ldots, m$, and let $\check{B}$ denote the problem instance given by the D_i. When run on $\check{B}$, algorithm $\mathcal{A}$ will also produce an optimal solution, and we have

$$\check{B}^* = \check{B}(\mathcal{A}) \leq B^* \leq B(\mathcal{A}) = \hat{B}^* = \hat{B}(\mathcal{A}).$$

Now, since the items in U_i, for $i = 0, \ldots, m-1$, can replace the items of D_{i+1} in this packing for $\check{B}$, the optimal packing for $\hat{B}$ uses as most $k = |U_m|$ bins more than $\check{B}(\mathcal{A})$. Also, since all items in $\check{B}$ have size at least ϵ, we must have $B^* \geq \epsilon n'$. By the definition of k, we thus conclude

$$B(\mathcal{A}) = \hat{B}(\mathcal{A}) \leq \check{B}(\mathcal{A}) + n'\epsilon^2 \leq \check{B}(\mathcal{A}) + \epsilon B^* \leq (1+\epsilon)B^*.$$

Thus, if we replace the required accuracy ϵ by $\frac{1}{2(1-\epsilon)}$ in the algorithm, we get for both cases

$$B(\mathcal{A}) \leq (1+\epsilon)B^* \text{ for } B^* \to \infty.$$

□

In (Karmarkar and Karp 1982), sequential approximation schemes are presented whose complexity and/or performance is considerably improved. These algorithms

rely on a subroutine solving a linear programming (LP) problem. LP in general is $\mathcal{P}$-complete (Dobkin et al. 1979), and it is presently not clear whether sufficient restrictions apply in the above case so that an $\mathcal{NC}$ algorithm can be found.

In section 4, we shall briefly discuss a different kind of approximation scheme for binpacking problems, one that approximates the *first fit decreasing* heuristic for binpacking.

2.3 Minimizing the Makespan

A third class of problems with parallel approximation algorithms based on discretization is given by the *makespan problem:* An instance of this problem consists of n tasks with positive (integer) execution times $t_1, \ldots, t_n$, and some number m of identical parallel processors. Each task is to be assigned to one of the processors. The execution time for a processor is the sum of the execution times of the tasks assigned to it. The goal is to minimize the makespan, the maximum of the execution times of all the processors. Again, this problem is a well-known $\mathcal{NP}$-complete problem (Garey and Johnson 1974). Parallel approximation algorithms for the problem have been given in (Mayr 1985) and (for the case $m = 2$) (Gopalakrishnan et al. 1986), and sequential versions in (Hochbaum and Shmoys 1985).

Let $\epsilon > 0$ be given. We wish to find a number M such that

$$M^* \leq M \leq (1+\epsilon)M^*,$$

where M^* is the optimal makespan. The optimal makespan problem can be related to the binpacking problem in the following way: We want to find the minimal M such that the tasks, considered as items, can be packed into m bins of size M (or unit size bins, after scaling the execution times by M). We note that the optimal scaling factor $M_{\min}$ clearly satisfies

$$T \leq M_{\min} \leq \lceil \frac{n}{m} \rceil \cdot T \text{ with } T = \max\{t_i;\ i = 1, \ldots, n\}.$$

Starting with these bounds, we perform a binary search, halving in each iteration the possible interval for $M_{\min}$. For each test value s (the midpoint of the current interval), we perform the following steps:

1. scale the execution times t_i by s;
2. temporarily discard all items $< \epsilon$;
3. round off the size of all items to the next lower integer multiple of ϵ^2;
4. pack modified items *optimally* into unit size bins;
5. undo step 3;
6. use the items smaller than ϵ to fill up the bins in such a way that at most one item exceeds the bin capacity of 1.

Step 4 in the above procedure can be carried out using a *structure graph* as with the binpacking problem. Also, step 6 is quite analogous to the step in the binpacking algorithm filling in the small items. We therefore leave the details to the reader.

If the above procedure succeeds in step 4 and, in step 6, manages to pack all small items, we replace the upper bound of the search interval by s, otherwise the lower bound. In the first case, we know that all tasks can be executed within time $(1+\epsilon)s$. In the second case, every bin is filled above capacity, and thus $s < M^*$. Let $[L, U]$ be the search interval after $\log n + \log 1/\epsilon$ iterations of the binary search. Then clearly

$$0 < U - L \leq \epsilon L \text{ and } M^* \in [L, U].$$

This implies that

$$M^* \leq U \leq (1+\epsilon)M^*,$$

and we return $M = U$ as our approximation to the optimal makespan. We conclude

Theorem 4 *For any fixed $\epsilon > 0$, the optimal makespan problem can be solved within $(1+\epsilon)$ of optimal by an $\mathcal{NC}$ algorithm.*

Again, the constants involved depend on ϵ and are rather big. They are of the same order as for the binpacking approximation algorithm.

3 The Scaling Approach

Discretization, of course, is closely related to *scaling* which is successfully used in sequential and parallel algorithms for quite a number of problems, *e.g.,* by Edmonds and Karp (1972), Karp et al. (1986), Gabow and Tarjan (1987, 1988), Goldberg and Tarjan (1987), and Orlin (1988).

Here, we consider the *list scheduling* problem, a $\mathcal{P}$-complete number problem (Helmbold and Mayr 1987a). It involves scheduling independent jobs on two processors. Formally, it is given by a list of n jobs, with (integer) execution times $t_1, \ldots, t_n > 0$. We are to construct a two processor schedule such that the ith job is started no later than the $i+1$st, for $i = 1, \ldots, n-1$, and there is no idle time between jobs. Note that it is straightforward to compute a list schedule by a sequential algorithm.

Lemma 3.1 *Let the execution times in a list scheduling problem be bounded by $L(n)$. Then a list schedule can be computed in parallel in $O(\log L(n) \log n)$ time using $O(n^2)$ processors.*

Proof: An algorithm with the required properties is given in (Helmbold and Mayr 1987a). We give a brief sketch. For definiteness, we assume that when both

processors become idle at the same time, the next job is assigned to the first processor. Let δ_i, for $i = 1, \ldots, n-1$, be the difference between the start times of the ith and the $i+1$st task in the list schedule. Also assume that all t_i are $\leq n$, and therefore all $\delta_i \leq n$. We construct an auxiliary graph, whose vertices are labelled with the pairs in $\{1, \ldots, n-1\} \times \{0, \ldots, n\}$. Let (i, δ) be such a pair. *Assuming* that $\delta_i = \delta$, we compute $\delta' = \delta_{i+1}$ (in constant time) and add an arc from (i, δ) to $(i+1, \delta')$ to the auxiliary graph. As in the binpacking algorithm, the resulting digraph is an in-forest, and we find the path from vertex $(1, 0)$ to the root of its tree. The labels of the vertices on this path can be used to determine, in a completely straightforward manner, the list schedule.

If (some of) the t_i are larger than n, we first divide them by a suitable power of 2 and round the results to the next lower integer, in such a way that the resulting execution times are bounded by n, and use the procedure outlined above. We then increase the number of relevant bits, one bit per phase. This corresponds to first doubling the execution times and then adding 1 to some of them. It turns out that, from phase to phase, the δ_i also first double, and then change (up or down) by an amount bounded by n. Therefore, there are at most $2n$ possible values for the new δ_i, and they can be determined constructing an auxiliary graph (in-forest) in quite the same way as described above.

The number of phases required is $\log L(n) - \log n$, the time per phase is $O(\log n)$, and the number of processors is n^2 on an EREW-PRAM. □

For list scheduling, we define an approximate solution to be a schedule that has the same first come, first served property as a list schedule, but we allow idle time between the jobs. The smaller the total idle time, the better the approximation. Using an $\mathcal{NC}$ algorithm to compute a list schedule for problems with small job times, we can construct an $\mathcal{NC}$ algorithm to approximate list scheduling with the idle time an arbitrarily small fraction of the schedule length.

Theorem 5 *For all $\epsilon > 0$, list scheduling can be approximated by an $\mathcal{NC}$ algorithm such that the total idle time is bounded by ϵ times the length of the schedule.*

Proof: Let $T = \frac{1}{2}\sum_{i=1}^{n} t_i$. If $T \leq \frac{n}{\epsilon}$ we solve the problem exactly, as described in the above Lemma. Otherwise, we first round each execution time up to the next integral multiple of 2^m where m is determined by

$$2^m \leq \frac{\epsilon \cdot T}{n} < 2^{m+1}.$$

The modified execution times have at most $O(\log(n/\epsilon))$ significant bits, and the modified problem can again be solved exactly, using the above algorithm. In the list schedule obtained for the modified problem, we then restore the original execution times, but keep the start times for the jobs. The idle time caused by restoring the execution time of the ith job is less than 2^m, and thus the total idle time is at most

$$2^m n \leq \epsilon \cdot T.$$

Since the length of the list schedule is at least T the claim of the Theorem follows. □

It should be noted that the list scheduling problem as desribed here is quite interesting because of this extremely close relationship between the size of the numbers in an instance, and its parallel complexity.

4 Combinatorial Approaches

In this section, we study approximation algorithms based on more combinatorial approaches. We also encounter the phenomenon that in some cases the (historically or, under certain assumptions, theoretically) best achievable accuracy is strictly bounded away from 1.

4.1 Greedy Scheduling

A *unit execution time task system* (UET task system) is given by (i) a set T of n tasks $t_1, \ldots, t_n$, each requiring unit time for execution, (ii) a partial order $\prec$ over T reflecting the precedence constraints among the tasks, and (iii) some number m of (identical) parallel processors. A *schedule* for a UET task system $(T, \prec)$ on m processors is a mapping of the tasks in T to unit length time intervals (with integral boundaries) such that, if $t \prec t'$ then t's time interval precedes that of t', and at most m tasks are mapped to any one interval. The length of a schedule is the number of distinct time intervals it uses. The *UET scheduling problem* is to find a schedule of minimal length. The problem is $\mathcal{NP}$-complete (Ullman 1975), as are some restricted versions (Mayr 1981). If $m = 2$, optimal schedules can be found in linear time (Gabow and Tarjan 1983), and in $\mathcal{NC}$ (Helmbold and Mayr 1987b).

A schedule is *greedy* if, whenever a timestep has less than m tasks mapped to it, that timestep contains all tasks available for execution, *i.e.*, a greedy schedule leaves no processor unnecessarily idle. It is known that the length of any greedy schedule is at most $2 - \frac{1}{m}$ times optimal, and that this bound is tight (Graham 1969). We show how to compute a greedy schedule fast in parallel.

We first determine for every task t its *level*, defined as the length of a longest path from a source (in-degree zero vertex) to t in the digraph P given by $(T, \prec)$. We then omit from P all arcs not between adjacent levels. Next, the tasks within every level are numbered arbitrarily. Every (directed) path in P starting at the first level (*i.e.*, with a source) is thus uniquely associated with a sequence of numbers. Any two such sequences can be compared lexicographically. We

compute, for every vertex t in P, the lexicographically maximal sequence over all paths starting at a source and ending at t. This computation is performed using a transitive closure routine based on iterated matrix multiplication, with scalar multiplication replaced by path composition and scalar addition replaced by taking the lexicographic maximum. Let $p(t)$ denote the lexicographically maximal sequence computed for t. We sort the tasks, first by ascending level, and then within every level in order of increasing $p(\cdot)$. The resulting list, L, determines a list schedule for $(T, \prec)$: at every step, as many executable tasks as possible (up to m) are scheduled, in the order given by L.

A simple induction shows that every timestep in the list schedule determined by L contains tasks from at most two distinct levels. We claim that the list schedule actually schedules the tasks in list order:

Lemma 4.1 *Let t and t' be two tasks, t' occurring later in L than t. Assume that in some timestep of the list schedule for L, task t is not yet executable. Then neither is t'.*

Proof: Assume to the contrary, and let t be the first task (in list order) that cannot be executed while some successor t' of t in L is executable. Also, let $\hat{t}$ be the immediate predecessor of t with the largest associated sequence that has not yet been executed, and let $\hat{t}'$ be the immediate predecessor of t' in P, also with the largest associated sequence. Then $p(\hat{t}')$ must be lexicographically at least as large as $p(\hat{t})$, and hence $\hat{t}$ is scheduled no later than $\hat{t}'$. Hence, t must be executable; contradiction. □

Given the list L, the corresponding list schedule can be computed fast in parallel: Assuming we know the position in L of the first task scheduled in timestep i, it is straightforward to find the first task scheduled in step $i+1$. Using a path finding technique as in earlier algorithms, we obtain

Theorem 6 *There is an $\mathcal{NC}$-algorithm to find a greedy schedule for UET task systems. The length of the greedy schedule is at most $2 - \frac{1}{m}$ times optimal, where m is the number of parallel processors for the schedule.*

If we define the level of a task to be the maximal distance to a sink (a task with no successor) instead of to a source, and schedule executable tasks on higher levels before those on lower levels, we obtain a so-called *highest level first (HLF)* schedule (also a list schedule). To compute an HLF schedule, however, is $\mathcal{P}$-complete, even when the precedence constraints are restricted to unions of an in-tree and an out-tree (Dolev et al. 1985).

4.2 Large Degree Induced Subgraphs

We shall discuss a $\mathcal{P}$-complete problem for which the best achievable approximation is more than a factor of 2 unless $\mathcal{P}$ equals $\mathcal{NC}$. For a more detailed presentation, the reader is referred to (Anderson and Mayr 1984).

Given a graph $G = (V, E)$ and an integer $d > 0$ the *high degree subgraph problem (HDS)* is to find $HDS_d(G)$, the maximum induced subgraph of G whose nodes all have degree at least d. There is a simple linear time sequential algorithm for this problem. It discards nodes of degree less than d until all remaining nodes have degree at least d, or the graph is empty. The correctness of this algorithm is completely straightforward.

We shall make use below of the following

Lemma 4.2 *If a graph has n vertices and m edges then it has an induced subgraph with minimum degree $\lceil \frac{m}{n} \rceil$.*

Proof: For a proof, see (Erdös 1963). □

We reformulate the high degree subgraph problem as a decision problem HDS by asking if a specific node v is in $HDS_d(G)$. In (Anderson and Mayr 1984) it is shown that

Theorem 7 *For $d \geq 3$, HDS is $\mathcal{P}$-complete. It is also $\mathcal{P}$-complete to decide whether $HDS_d(G)$ is nonempty, for d in the same range.* □

On the other hand, it is quite easy to see that $HDS_2(G)$ can be computed by an $\mathcal{NC}$ algorithm. The algorithm has $\log n$ phases, where each phase removes all *chains*. A chain is a path that starts with a vertex of degree 1 and contains no vertex of degree greater than 2. The chains can easily be identified by path doubling techniques. When the chains are deleted, more nodes of degree 1 might be created, however, each new node of degree 1 required the removal of at least two chains, so the number of chains decreases by at least half at each phase. We thus have the following

Theorem 8 *It is possible to compute $HDS_2(G)$ by an $\mathcal{NC}$ algorithm.* □

Consider the following optimization problem: Given a graph G, find the largest d such that $HDS_d(G)$ is nonempty. We denote this value by $D^* = D^*(G)$. An approximate solution for the problem is an integer D such that, for some positive constant $c > 1$,

$$D^*(G) \leq D \leq cD^*(G).$$

We call such a D a c-approximation for D^*.

Theorem 9 *Let c be a constant greater than 2. Then a c-approximation for $D^*(G)$ can be found by an $\mathcal{NC}$ algorithm.*

Proof: Let $\epsilon = 1 - \frac{2}{c} > 0$, and consider the following pruning procedure (with parameter d):

1. let n' be the number of vertices currently in G;
2. **if** at most $\epsilon n'$ vertices have degree $< d$ **then stop**;
3. remove all vertices from G that have degree $< d$;
4. goto 1.

This pruning procedure removes vertices from G until G is empty or at most a proportion of ϵ of the remaining vertices have degree $< d$. In the first case, G clearly has no induced subgraph with degree at least d. In the second case, the remaining graph contains at least $\frac{1-\epsilon}{2} \cdot dn'$ edges, and hence, by the Lemma, $D^*(G) \geq \frac{1-\epsilon}{2} \cdot d$. Running the pruning procedure for every $d \in [1, n-1]$, let D be the largest such parameter for which the procedure returns a nonempty graph. Then $D^* \leq D \leq cD^*$. □

The next theorem shows that the previous result is essentially the best possible assuming that $\mathcal{P} \neq \mathcal{NC}$. We show that a circuit can be simulated by a graph G with $D^*(G) = 2d$ if the output of the circuit is 1 and $D^*(G) \leq d + 1$ if the output is 0.

Theorem 10 *Let c be a constant, $1 \leq c < 2$. Unless $\mathcal{P} = \mathcal{NC}$, there is no $\mathcal{NC}$ algorithm to compute c-approximations for the high degree subgraph problem.*

Proof: It is not hard to see that the monotone circuit value problem with OR gates having just one output is still $\mathcal{P}$-complete (just use an AND gate with one input tied to 1 to achieve fanout). We shall give a logspace reduction that, given such a circuit and an integer $D > 1$, constructs a graph G with $D^*(G) = 2D$ if the output of the circuit is 1, and with $D^*(G) \leq D + 1$ otherwise. In the reduction, first every gate of the circuit is replaced by a corresponding gadget, as shown in Figures 1 and 2.

In these figures, the circles represent sets of D vertices each, the bold edges denote the edges of a complete bipartite graph on the two vertex sets they connect (*i.e.*, a $K_{D,D}$), and the thin edges in Figure 2 denote a set of D edges giving some arbitrary bijection between the two sets of D vertices each. Figure 2 shows the special case of an AND gate with two outputs. Different fanout can easily be achieved repeating the pattern given in the figure, with D circles connected by bold edges to a line and each of them with a thin line to the corresponding "output" circle. When a wire connects an output of one gate with an input of another gate, the corresponding "output" circle of the first gate is identified with the proper "input" circle of the second. To complete the reduction, we add to the circuit an AND gate one of whose inputs is the original output of the circuit, while the other is tied to 1. Each 1-input of the circuit becomes an output of this AND gate. Of course, this additional gate then is also replaced by the corresponding gadget.

Assume first that the output of the circuit, with the given assignment of input values, is 1. Then the subgraph induced by (the vertices in) the circles in

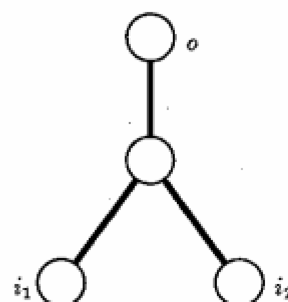

Figure 1: OR gate

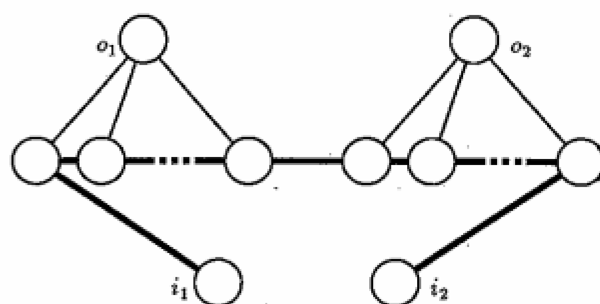

Figure 2: AND gate

gadgets for gates with value 1 (omitting, in the case of OR gates, inputs with value 0) has degree at least $2D$ as one can easily verify. It is also quite easy to see that, in this case, $D^* < 2D + 1$, and thus $D^* = 2D$.

Consider now the case where the output of the circuit is 0. We remove, from the graph constructed in the reduction, vertices of degree at most $D + 2$. Starting at 0-inputs to the circuit, it is clear that the gadget for the output gate, the added gadget feeding the outputs back to the 1-inputs, and finally the vertices in all gadgets get removed. Hence, as claimed, $D^* \leq D + 1$. □

For a detailed discussion of another type of parallel approximations for the high degree subgraph problem (yielding algorithms not necessarily in $\mathcal{NC}$), we refer the reader to (Anderson and Mayr 1984).

4.3 FFD Binpacking

The *first-fit-decreasing* (FFD) method is a simple heuristic for the $\mathcal{NP}$-complete (one-dimensional) binpacking problem with a guaranteed performance bound of $\frac{11}{9}$ times optimal. Unfortunately, computing an FFD packing, given the size of the items, is $\mathcal{P}$-complete in the strong sense (*i.e.*, even if numerical values are given in unary notation) as shown in (Anderson et al. 1988). In this paper, it is also shown that an FFD packing can in fact be computed by an $\mathcal{NC}$ algorithm if all item sizes are bounded from below by some $\epsilon > 0$. This routine can then be used to "approximate" an FFD packing for unrestricted problem instances: First, items of size at least

$\frac{1}{6}$ are packed according to the FFD heuristic, using the $\mathcal{NC}$ routine. The remaining, small items are then used to fill up these and, if necessary, additional bins. This fill-in routine is similar to the ones discussed earlier, and is easily seen to be in $\mathcal{NC}$. For a detailed presentation of the algorithms and results, the reader is referred to (Anderson et al. 1988).

5 Conclusion

We have presented (full) $\mathcal{NC}$ approximation schemes for $\mathcal{NP}$-hard optimization problems like the 0-1 knapsack problem, binpacking, and the makespan minimization problem. We have also shown a full $\mathcal{NC}$ approximation scheme for the $\mathcal{P}$-complete list scheduling problem. Finally, we have discussed some problems for which the best achievable accuracy for approximation schemes is, as far as we know, strictly bounded away from 0.

Of course, there are other parallel approximation algorithms. For example, we did not touch on approximate string matching, nor on the use of the *random* $\mathcal{NC}$ routine in (Karp et al. 1985) for maximum cardinality matching in graphs, which can be employed to build approximation algorithms for maximum weight matchings and maximum flow problems, based on scaling.

We have shown that efficient parallel approximation schemes exist for some $\mathcal{NP}$-complete problems as well as for suitable $\mathcal{P}$-complete problems. Finding an FFD packing and Linear Programming (LP) are strongly $\mathcal{P}$-complete. Nonetheless, FFD can be approximated by $\mathcal{NC}$ algorithms in a certain sense, while no such approach is known for LP. It is also an interesting open problem to find other strongly $\mathcal{P}$-complete problems, and to study whether, and in which sense, exact solutions for them can be approximated efficiently in parallel.

Another challenging area for research is to develop approximation methods for problems of practical interest (as some of those mentioned above) that can be implemented efficiently on more realistic parallel architectures, like the Hypercube. Many of the results shown here are asymptotic bounds and sometimes hide large constants (or, even worse, polynomial degrees).

REFERENCES

[Anderson and Mayr 1984] R. ANDERSON, E. MAYR: *A $\mathcal{P}$-complete problem and approximations to it.* STAN-CS-84-1014, Department of Computer Science, Stanford University (1984).

[Anderson et al. 1988] R. ANDERSON, E. MAYR, M. WARMUTH: *Parallel approximation algorithms for bin packing.* STAN-CS-88-1200, Department of Computer Science, Stanford University (March 1988).

[Dobkin et al. 1979] D. DOBKIN, R. LIPTON, S. REISS: Linear programming is log-space hard for $\mathcal{P}$. *Information Processing Letters*, 8(2):96–97, 1979.

[Dolev et al. 1985] D. DOLEV, E. UPFAL, M. WARMUTH: Scheduling trees in parallel. In P. Bertolazzi, F. Luccio, editors, *VLSI: Algorithms and Architectures. Proceedings of the International Workshop on Parallel Computation and VLSI*, pages 1–30, 1985.

[Edmonds and Karp 1972] J. EDMONDS, R.M. KARP: Theoretical improvements in algorithmic efficiency for network flow problems. *J.ACM*, 19:248–264, 1972.

[Erdős 1963] ERDŐS, P.: On the structure of linear graphs. *Israel Journal of Mathematics*, 1:156–160, 1963.

[Fernandez de la Vega and Lueker 1981] W. FERNANDEZ DE LA VEGA, G.S. LUEKER: Bin packing can be solved within $1+\epsilon$ in linear time. *Combinatorica*, 1(4):349–355, 1981.

[Fortune and Wyllie 1978] S. FORTUNE, J. WYLLIE: Parallelism in random access machines. In *Proceedings of the 10th Ann. ACM Symposium on Theory of Computing (San Diego, CA)*, pages 114–118, 1978.

[Gabow and Tarjan 1983] H.N. GABOW, R.E. TARJAN: A linear-time algorithm for a special case of disjoint set union. In *Proceedings of the 15th Ann. ACM Symposium on Theory of Computing (Boston, Mass.)*, pages 246–251, 1983.

[Gabow and Tarjan 1987] H.N. GABOW, R.E. TARJAN: *Faster scaling algorithms for network problems.* CS-TR-111-87, Department of Computer Science, Princeton University (August 1987).

[Gabow and Tarjan 1988] H.N. GABOW, R.E. TARJAN: Almost-optimum speed-ups of algorithms for bipartite matching and related problems. In *Proceedings of the 20th Ann. ACM Symposium on Theory of Computing (Chicago, IL)*, pages 514–527, 1988.

[Garey and Johnson 1979] M.R. GAREY, D.S. JOHNSON: Computers and intractability: A guide to the theory of NP-completeness. San Francisco: W.H. Freeman (1979).

[Goldberg and Tarjan 1987] A. GOLDBERG, R.E. TARJAN: Solving minimum-cost flow problems by successive approximations. In *Proceedings of the 19th Ann. ACM Symposium on Theory of Computing (New York City)*, pages 7–18, 1987.

[Gopalakrishnan et al. 1986] P.S. GOPALAKRISHNAN, I.V. RAMAKRISHNAN, L.N. KANAL: *Fast approximation schemes for a class of combinatorial optimization problems on a parallel machine.* CS-TR-1725, Department of Computer Science, University of Maryland (October 1986).

[Graham 1969] GRAHAM, R.L.: Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math.*, 17(2):416–429, 1969.

[Helmbold and Mayr 1987a] D. HELMBOLD, E. MAYR: Fast scheduling algorithms on parallel computers. In F.P. Preparata, editor, *Advances in Computing Research; Parallel and Distributed Computing*, pages 39–68, JAI Press, 1987.

[Helmbold and Mayr 1987a] D. HELMBOLD, E. MAYR: Two processor scheduling is in $\mathcal{NC}$. *SIAM J. on Comput.*, 16:747–759, 1987.

[Hochbaum and Shmoys 1985] D.S. HOCHBAUM, D.B. SHMOYS: Using dual approximation algorithms for scheduling problems: Theoretical and practical results. In *Proceedings of the 26th Ann. IEEE Symposium on Foundations of Computer Science (Portland, OR)*, pages 79–89, 1985.

[Ibarra and Kim 1975] O.H. IBARRA, C.E. KIM: Fast approximation algorithms for the knapsack and sum of subset problems. *J.ACM*, 22(4):463–468, 1975.

[Karmarkar and Karp 1982] N. KARMARKAR, R.M. KARP: An efficient approximation scheme for the one-dimensional bin-packing problem. In *Proceedings of the 23rd Ann. IEEE Symposium on Foundations of Computer Science (Chicago, IL)*, pages 312–320, 1982.

[Karp et al. 1985] R.M. KARP, E. UPFAL, A. WIGDERSON: Constructing a perfect matching is in random $\mathcal{NC}$. In *Proceedings of the 17th Ann. ACM Symposium on Theory of Computing (Providence, RI)*, pages 22–32, 1985.

[Ladner and Fischer 1980] R.E. LADNER, M.J. FISCHER: Parallel prefix computation. *J.ACM*, 27(4): 831–838, 1980.

[Lawler 1979] LAWLER, E.: Fast approximation algorithms for knapsack problems. *Mathematics of Operations Research*, 4:339–356, 1979.

[Mayr 1981] E. MAYR: *Well structured parallel programs are not easier to schedule.* STAN-CS-81-880, Department of Computer Science, Stanford University (September 1981).

[Mayr 1985] E. MAYR: Efficient parallel scheduling algorithms. *19th Ann. Asilomar Conf. on Circuits, Systems and Computers, November 7, 1985.*

[Orlin 1988] J. ORLIN: A faster strongly polynomial minimum cost flow algorithm. In *Proceedings of the 20th Ann. ACM Symposium on Theory of Computing (Chicago, IL)*, pages 377–387, 1988.

[Peters and Rudolph 1984] J. PETERS, L. RUDOLPH: Parallel approximation schemes for subset sum and knapsack problems. In *Proceedings of the 22nd Ann. Allerton Conference on Communication, Control and Computing*, pages 671–680, 1984.

[Pfister 1985] G.F. PFISTER: *The architecture of the IBM research parallel processor prototype (RP3).* Research Report RC 11210, IBM Yorktown Heights (June 1985).

[Pippenger 1979] N. PIPPENGER: On simultaneous resource bounds. In *Proceedings of the 20th Ann. IEEE Symposium on Foundations of Computer Science (San Juan, PR)*, pages 307–311, 1979.

[Preparata and Vuillemin 1979] F.P. PREPARATA, J. VUILLEMIN: The cube-connected-cycles: A versatile network for parallel computation. In *Proceedings of the 20th Ann. IEEE Symposium on Foundations of Computer Science (San Juan, PR)*, pages 140–147, 1979.

[Schwartz 1980] J. SCHWARTZ: Ultracomputers. *ACM Trans. on Programming, Languages and Systems*, 2(4):484–521, 1980.

[Seitz 1985] C. SEITZ: The Cosmic Cube. *Comm.ACM*, 28(1):22–33, 1985.

[Ullman 1975] J.D. ULLMAN: $\mathcal{NP}$-complete scheduling problems. *JCSS*, 10(3):384–393, 1975.

[Vishkin 1987] U. VISHKIN: An optimal parallel algorithm for selection. In F.P. Preparata, editor, *Advances in Computing Research; Parallel and Distributed Computing*, pages 79–86, JAI Press, 1987.