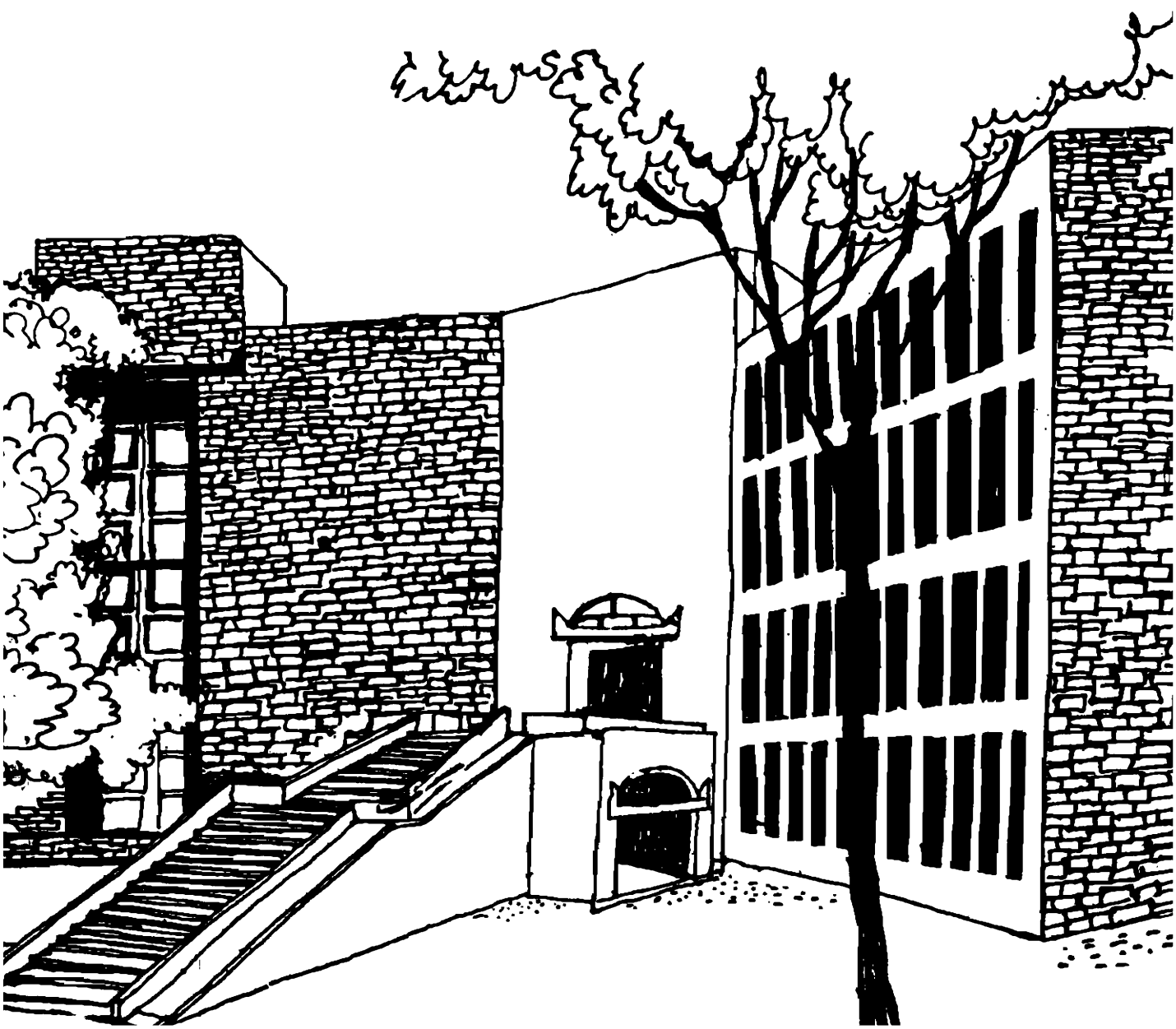
विद्याविनियोगाद्विकासः

IIM AHMEDABAD

Working Paper



SCHEDULING PARTIALLY ORDERED ACTIVITIES UNDER RESOURCE CONSTRAINTS ON PARALLEL MACHINES

By

Sanjay Verma
N.P. Dhavale
Amitava Bagchi

W.P.No. 2000-04-04
April 2000 1596

INDIAN INSTITUTE OF MANAGEMENT
AHMEDABAD-380 015
INDIA

Scheduling Partially Ordered Activities Under Resource Constraints on Parallel Machines

Sanjay Verma
Indian Institute of Management Ahmedabad
Vastrapur, Ahmedabad - 380 015, India
E-mail: sverma@iimahd.ernet.in

and

N P Dhavale and Amitava Bagchi
Indian Institute of Management Calcutta
Joka, Diamond Harbour Road
P O Box 16757, Calcutta - 700 027, India
E-mail: bagchi@iimcal.ac.in

Abstract: A four-step strategy is proposed for scheduling a set of partially ordered resource-constrained activities on a given number of identical parallel machines. For regular measures, the generated schedule is optimal. In the first step, the difficulty level of the problem instance is estimated using the problem parameters as arguments. Easy problems are solved using a best-first tree search algorithm. For harder problems, an approximate algorithm is employed to determine a good upper bound on the measure. This bound is fed to a breadth-first tree search algorithm, making the pruning more effective and reducing the memory requirement of breadth-first search. When the number of parallel machines is 2, 3, 4, 5 or unlimited, this strategy is able to solve all except a small number of the benchmark PROGEN problems on a Linux-based Pentium PC or a UNIX-based RS 6000 machine. On projects without resource constraints, the proposed method is faster than the earlier method of Chang and Jiang [1994] by orders of magnitude.

Keywords: Resource constrained project scheduling, parallel machines, breadth-first search, best-first search

1 Introduction: Perhaps the most commonly studied problem in the area of project scheduling is the *Multiple Resource-Constrained Project Scheduling Problem* (RCPSP). The objective in the RCPSP is to schedule the partially ordered activities belonging to a given project in a non-preemptive manner without violating any resource constraints so as to minimize some regular performance measure such as the schedule length (*makespan*). A *regular* performance measure [French 1982, pp 13-14] is one that is non-decreasing in the completion times of activities; this implies that in an optimal schedule it is never necessary to keep a machine deliberately idle if a ready job is available for scheduling. A number of exact algorithms are currently available for solving instances of the RCPSP [Stinson *et al* 1978], [Patterson 1984], [Christofides *et al* 1987],

[Demeulemeester and Herroelen 1992 1997]. These generally make use of a best-first or depth-first tree search scheme augmented with pruning rules. In the search tree, nodes correspond to partial schedules; in particular, the root node corresponds to the empty schedule and leaf nodes correspond to complete feasible schedules. The effectiveness of a solution method depends mainly on how quickly the pruning rules are able to weed out partial schedules that are provably sub-optimal. The RCPSP is an NP-complete problem [Garey and Johnson 1979, pp 236-242], and even small problem instances involving thirty or fewer activities can be quite hard to solve. For larger instances, both the running time and the memory requirement are often excessive. Not many of the proposed methods are able to solve optimally the complete set of 680 benchmark PROGEN problems [Kolisch *et al* 1995] even on the most powerful PCs and workstations.

Two new exact algorithms for the RCPSP, called Best_MRS and Breadth_MRS, have been proposed recently by Nazareth *et al* [1999]. Best_MRS employs a conventional best-first tree search scheme that makes use of the standard makespan heuristic, while Breadth_MRS employs breadth-first search, in which the nodes in the search tree are processed in a level-by-level manner. Experiments indicate that Best_MRS is generally faster than Breadth_MRS. But on hard problems, Best_MRS occasionally runs out of memory. Most people believe breadth-first search to be inefficient since it is unable to direct the search to a goal. But in the area of resource constrained project scheduling it is superior to best-first search in the utilization of memory. The two algorithms were coded in C and run on a 64 MB 100 MHz Pentium PC in a SCO UNIX environment. Breadth_MRS solved *all* the 680 PROGEN problems (First Lot: 480, Second Lot: 200), but for nine of the problems it took more than 1,000 secs of runtime. Best_MRS ran out of memory on five problems, all belonging to the First Lot; however, the maximum runtime for a problem it did solve was only 384 secs.

A novel feature of the two algorithms is that both permit the *retraction* of activities. An activity in a partial schedule that has already been scheduled and is currently being processed can be *retracted* in a descendant schedule. This means that the processing of the activity can be suspended and its status rolled back, so that it is as if the activity has not been scheduled at all. This feature drastically reduces the branching factor of the search tree, making it possible to solve problem instances of larger size within the given memory limits.

This paper tries to extend the range of application of the tree search schemes described in [Nazareth *et al* 1999]. It is concerned with a variant of the RCPSP called the *Parallel Machine Resource-Constrained Scheduling Problem* (PMRCSP). In the PMRCSP, a set of partially ordered

activities must be scheduled in a non-preemptive manner under resource constraints on a specified number of identical parallel machines so as to optimize some regular performance measure such as makespan or mean flow time. Specifying the number of parallel machines amounts to imposing an upper bound on the number of activities that can be processed simultaneously at any instant of time. Consider a multiprocessor computer system with m processors. In this system at most m programs can run at the same time. We can think of a processor as a type of renewable resource. Its total availability equals m, the number of processors in the system, and a program that is running has a requirement of one processor at each time instant. The partially ordered set of programs need not be viewed as a project; it can be viewed merely as a collection of activities. The objective could be to minimize the makespan or the mean flow time or the maximum tardiness or the number of tardy jobs. In the special case when the number of processors is infinite, the PMRCSP reduces to the RCPSP. No method that is reasonably efficient in solving the PMRCSP is available in the technical literature. The algorithm of Chang and Jiang [1994] does not take resource constraints into account. Our main goal in this paper is to propose a breadth-first search strategy based on Breadth_MRS that can find exact solutions to the PMRCSP for problems such as the ones in the PROGEN set, with an additional constraint on the number of processors. The PROGEN problems were chosen because the difficulty level is high and there is no standard benchmark set for the PMRCSP.

Direct use of Breadth_MRS or Best_MRS fails to solve the PMRCSP for many of the PROGEN problems when the number of available processors is 3 or 4. This led us to adopt the more elaborate four-step strategy OPT_MEASURE described in Section 2. This strategy is exact. Whenever it solves a problem, it finds the optimal solution. In the first step, an attempt is made to estimate roughly the difficulty level of a problem instance from the problem parameters. If the problem instance is found to be easy, Best_MRS is run on it. If it is hard or if Best_MRS fails to solve it, a good upper bound on the performance measure of interest is obtained with the help of a fast approximate algorithm, and then Breadth_MRS is run on it. When an upper bound is supplied to a breadth-first search scheme such as Breadth_MRS, the pruning becomes far more effective. For harder problems the runtime and memory requirement are both drastically cut down, making it possible to solve many more problem instances than by a direct application of Breadth_MRS alone.

OPT_MEASURE has been tested experimentally on a UNIX-based RS 6000 machine and a Linux-based Pentium PC with the number m of parallel machines set to the following values: 2, 3, 4, 5 and infinity. With makespan as the performance measure, for m = 2, 5, and infinity, it found

solutions on the RS 6000 to *all* 480 problems in the First Lot of PROGEN; for $m = 4$ it found solutions to 478 problems, and for $m = 3$ to 472 problems. Thus out of a total of 480 * 5 = 2,400 problems, only 10 were not solved. In the Second Lot, out of a total of 200 * 5 = 1,000 problems, only 41 were not solved. The Linux-based PC solved a slightly smaller number of problems because it had less memory. The approximate algorithm GET_MEASURE that forms part of OPT_MEASURE also gave excellent results. For example, for infinite m it found optimal solutions on the RS 6000 to 406 of the 480 problems in the First Lot within 20,000 iterations taking no more than 37 secs per problem. For the problems it did not solve optimally in 20,000 iterations, it was within two units of the optimal on the average. Section 2 gives a detailed description of OPT_MEASURE and Section 3 contains our experimental results. Section 4 lists some areas in which the work reported in this paper can be extended.

2 The Solution Method: We first formulate the problem, and then describe the solution method OPT_MEASURE in detail.

2.1 Problem Formulation: The problem formulation is similar to that in [Nazareth *et al* 1999]. We are given a set of N activities for some $N > 2$. Activity a_i has duration $d_i \geq 0$, a set P_i of predecessor activities, a set S_i of successor activities, and a requirement of $r_{ij} \geq 0$ units of the jth resource type for each of the $M \geq 1$ distinct types of resource. All resources are assumed to be renewable. The total availability R_j of the jth resource type, $1 \leq j \leq M$, is constant and specified in advance. Resource type M corresponds to parallel machines, so that $R_M = m$, the specified number of parallel machines. Each activity has a requirement of one unit of resource M at every instant during execution. It is assumed that the schedule is non-preemptive, *i.e.*, an activity once started cannot be interrupted. It is also assumed that a unit of resource cannot be simultaneously allocated to two activities. The resources used by an activity are allocated to it when processing begins, and released by it when processing gets completed. At any moment of time, the total number of units of the j^{th} resource type allocated to all the activities taken together cannot exceed R_j. Given d_i, P_i and $\{ r_{ij}, 1 \leq j \leq M \}$ for each activity a_i, $1 < i < N$, and the pool of available resources R_j, $1 \leq j \leq M$, the problem is to determine the start time s_i of each activity a_i such that the appropriate performance measure is minimized. All numeric parameters are integers. Following standard convention we assume that the project has two dummy activities, a start activity a_1 and a finish activity a_N, which have zero duration and do not consume any resources. For convenience we also assume that activities are so numbered that if a_i is in P_j then $i < j$. If f_i represents the finish time of activity a_i, then the makespan problem can be formulated mathematically as follows:

Minimize f_N subject to the conditions

i) $f_i - f_j \geq d_i$ for each precedence constraint (a_j, a_i), $1 \leq i, j \leq N$; and

ii) $\Sigma r_{ij} \leq R_j$ for each j, $1 \leq j \leq M$, at every integer time instant t, $0 \leq t \leq f_N$, where the summation is over all i such that activity a_i is in progress during the time interval [t, t+1).

2.2 The Solution Strategy OPT_MEASURE: Given an instance of the PMRCSP, the solution strategy OPT_MEASURE tries to solve the problem in four steps as follows:

Step 1: Estimate the difficulty level of the given problem from the basic problem parameters such as the numbers of activities and resource types, the resource availabilities, and the activity durations and resource requirements.

Step 2: If the estimated difficulty level is low, try to solve the problem using Best_MRS.

Step 3: If the estimated difficulty level is high, or if Best_MRS in Step 2 runs out of memory, run algorithm GET_MEASURE to determine a good upper bound GM on the specified measure.

Step 4: Run Breadth_MRS with the upper bound GM that has been found in Step 3.

We now describe each step of the solution method in greater detail, assuming that makespan is the performance measure.

2.3 Estimation of Difficulty Level: The first step in OPT_MEASURE involves the estimation of the difficulty level of a problem from the problem parameters. We can compute, as explained in [Kolisch *et al* 1995], the following three characteristics of the given problem: network complexity (NC), resource factor (RF) and resource strength (RS). NC measures the difficulty level of the project network in terms of the number of nodes and edges in the network; the smaller the value of NC, the fewer the precedence constraints and the harder the problem. RF reflects the average resource requirement of activities; RF = 1 means all activities request all types of resource, while RF = 0 means no activity requests any type of resource. RS measures how close the resource requirement is to resource availability. When the number m of parallel machines is infinite, a difficulty level DL can be computed from these three factors using the following thumb rule:

$$DL = NC^4/7.5 + 1/RF + RS^2/25$$

This thumb rule was obtained by trial-and-error. We found experimentally that when $DL \geq 3$, the problem instance can be solved directly by Best_MRS on the RS 6000, but when $DL < 3$, a more elaborate solution procedure has to be adopted.

Unfortunately, DL fails to serve as a reliable indicator of the difficulty level when m is a small integer such as 2, 3, 4 or 5. In such cases an alternative approach is needed. Let m be specified, $2 \leq m \leq 5$. We first compute DL as above for the given problem instance and find an optimal schedule S for an infinite number of processors using the strategy outlined below. A scrutiny of schedule S

reveals the time instants at which the number of activities being processed equals k for k = 1, 2, ..., *m*, *m*+1, ... The significant instants are those at which k exceeds *m*, since at most *m* activities can be processed simultaneously. Suppose we find that there are many instants at which k > *m*, or there is at least one instant at which k exceeds *m* by a significant amount (say three or more). We then infer that the given problem is likely to be hard for Best_MRS for the specified *m*. This serves only as a rough guide; a better method for estimating the difficulty level when *m* is a small integer has not yet been found.

2.4 Procedure GET_MEASURE: Since a fair number of the PROGEN problems are quite hard to solve, a need was felt for a simple approximate scheme that would yield a good upper bound on the specified performance measure. This upper bound can be used for pruning redundant states in a minimization problem. GET_MEASURE is an iterative algorithm that randomly generates feasible

Procedure GET_MEASURE /* Minimization problem assumed */

Step 1: Fix the number *m* of parallel machines. /* *m* is 2, 3, 4, 5 or infinity */

Step 2: For each activity i, $1 \leq i \leq N$, in the given problem, determine its *forward order* forward[i] as follows:

$$\text{forward}[i] = 1 \quad \text{if } i = 1$$
$$= \max \{ \text{forward}[j] + 1 \mid j \text{ in } P_i \} \quad \text{if } i > 1$$

Step 3: Let MAX be a random integer lying in the interval forward[N] ≤ MAX ≤ 2*N. For each activity i, $1 \leq i \leq N$, in the given problem, determine its *backward order* backward[i] as follows:

$$\text{backward}[i] = \text{MAX} \quad \text{if } i = N$$
$$= \min \{ \text{backward}[j] - 1 \mid j \text{ in } S_i \} \quad \text{if } i < N$$

/* Thus backward[i] is an integer that lies between 1 and a maximum value MAX that depends on the number of activities. We can simply take MAX = N, but the algorithm improves slightly in performance if MAX is set to a random integer lying between forward[N] and 2*N. */

Step 4: Let order[1] = 1 and order[N] = MAX. For each activity i, $1 < i < N$, randomly generate an integer order[i] in the interval forward[i] ≤ order[i] ≤ backward[i], ensuring that order[i] > order[j] for each activity j belonging to P_i.

/* It is possible that order[i] = order[j] for two activities i and j that are unrelated in the precedence relationship. Note that order[i] does not depend on *m*. */

Step 5: Assign start times to activities in increasing order of order[i], allowing at most *m* activities to be processed simultaneously. Always assign as low a value to the start time as possible without violating resource or precedence constraints. If two or more activities have the same order then choose one of them randomly and assign a start time to it first. Repeat this step until a complete feasible schedule is generated.

Step 6: Repeat steps 3-5 for the stipulated number of iterations.

Step 7: Output GM, the lowest value of the performance measure obtained among all the iterations. /* The schedule corresponding to GM can also be outputted if needed. */

schedules and selects the best of them. It has been found experimentally to be quite effective. When *m* is infinite, it outputs optimal solutions on the RS 6000 machine in 200 iterations taking less than 0.4 sec per problem to more than half of the 680 PROGEN problems. A significant minority of the problems requires a much larger number of iterations to attain the optimal value, and for a small number even 20 million iterations are inadequate. But the upper bound obtained is always good. Experiments were also conducted with methods such as simulated annealing [Cho and Kim 1997] and genetic algorithms [Mori and Tseng 1997], but none of the alternative schemes could compete with GET_MEASURE in simplicity and effectiveness.

Given a resource-constrained project as input, GET_MEASURE first computes two integer parameters for each activity, the *forward order* and the *backward order*, where forward order ≤ backward order. A random integer k is generated such that: i) forward order ≤ k ≤ backward order, *and* ii) k is greater than the order of all predecessor activities. This k becomes the *order* of the activity. After the orders of all the activities have been determined, the activities are scheduled non-preemptively in the sequence of their orders in such a way that, at every instant, at most *m* activities get processed simultaneously *and* all resource constraints are satisfied. This entire procedure is iterated a given number of times, and the length of the feasible schedule with the shortest length is outputted as the makespan of the project. The procedure can be generalized to work for any regular performance measure.

2.5 Breadth_MRS and Best_MRS: Detailed descriptions of Breadth_MRS and Best_MRS together with illustrative examples are given in Nazareth *et al* [1999]. Both are tree search procedures for solving the RCPSP. A node in the search tree corresponds to a partial schedule. The expansion of a node corresponds to the extension of a partial schedule by the addition of one more activity. The immediate successors of a node correspond to the different ways in which another activity can be added to the partial schedule. Breadth_MRS searches the tree level by level and does not use any heuristic estimate to guide the search. It uses the notion of a *Maximal Resource Satisfying Set* (MRS) to create new nodes. Each MRS corresponds to a partial schedule. When the processing of an activity is completed, some additional activities become ready for processing, and a new set of MRSs get created out of the pool of activities waiting to be processed. The tighter the resource constraints, the fewer the number of activities in an MRS. Formally, an MRS *A* is a set of activities such that:

i) All the activities in *A* are ready to be processed, *i.e.*, all their predecessors have already been processed.

ii) The activities in *A* taken together satisfy all the resource constraints and so can be processed simultaneously.

iii) A is a maximal set in the sense that if another activity that is ready for processing is added to *A*, the resulting set of activities fail to satisfy the resource constraints.

Breadth_MRS uses three pruning rules to cut down the number of MRSs. These are the One Child Rule, the Left Shift Rule and the Dominance Pruning Rule (see [Nazareth *et al* 1999] for details). Best_MRS is a best-first version of Breadth_MRS that employs the standard makespan heuristic to guide the search. It makes use of the same pruning rules as Breadth_MRS.

When Breadth_MRS terminates we want not only the minimum makespan but also a complete schedule that achieves that makespan. To output the schedule, the entire search tree must be stored in memory. This increases the memory requirements slightly. Alternatively, a simple depth-first search will find the schedule once the minimum makespan is known.

Procedure UBDP /* generalized version of Breadth_MRS */

```
begin
    input m, U;                          /* m is the no of processors and U is the upper bound */
    create the root state at level 0 in the search tree; put the root state in OPEN;
    for L = 0 to N-1 step 1 begin
        if there is no state in OPEN at level L then exit with output of 0;        /* failure */
        for each state X in OPEN at level L begin
            determine dpX; /* dpX is the earliest instant at which a running activity completes */
            construct KX;                /* KX is the set of all ready jobs at instant dpX */
            construct all MRSs with at most m activities;
            delete state X from OPEN;
            for each MRS A built out of KX begin
                determine makespan heuristic for A and compute estimated schedule length;
                if this length is greater than U then delete A from the set of MRSs;
            end;
            if the One-Child Rule applies
            then generate one child state of X at level L+1 corresponding to the singular MRS
                and put it in OPEN;
            else for each remaining MRS A begin
                if the Left-Shift Rule applies to A
                then do not generate any child state corresponding to A
                else generate a child state at level L+1 corresponding to A and put it in OPEN;
            end;
        end;
        apply the Dominance Pruning Rule to all states in OPEN at level L+1;
    end;
    output the complete schedule associated with the state at level N;
end.
```

Using the notation of Nazareth *et al* [1999], we present in algorithmic form an enhanced version of Breadth_MRS for *m* parallel machines that incorporates upper bound pruning. To distinguish the generalized version of Breadth_MRS from the earlier versions, we call it UBDP. The upper bound U is obtained from a good approximate algorithm such as GET_MEASURE. Such an upper bound can help a breadth-first search method to prune a large number of states. To make effective use of this upper bound, the algorithm needs to compute the estimated schedule length for each partial schedule; this can be achieved using the makespan heuristic as in Best_MRS. Note that Best_MRS being a best-first algorithm is itself unable to make use of Upper Bound Pruning. As in the original Breadth_MRS, states corresponding to partial schedules are stored in a FIFO list (*i.e.*, a queue) called OPEN. States are taken out of OPEN from one end and new states are entered into OPEN at the other end. UBDP can be used even when *m* is infinite; we just need to specify a large value of *m*, such as 20. The procedure given here assumes that makespan is the measure, but it can be readily generalized to work for any other regular performance measure.

It can be shown that Breadth_MRS outputs optimal solutions for the makespan measure or for any other regular performance measure. The proof is essentially identical to the one given in [Nazareth *et al* 1999].

Theorem 1: For any regular performance measure, UBDP outputs optimal solutions.

Proof: See the proofs of Theorems 1-4 in [Nazareth *et al* 1999]. It is readily verified that these proofs hold for any regular performance measure. We have just added one more resource type, namely the number of parallel machines. The earlier proofs hold for any number of resource types and therefore remain valid even when there is a limit on the number of parallel machines. We first establish that UBDP outputs optimal solutions in the absence of any pruning rules. We then introduce the pruning rules one by one in UBDP and establish that the procedure continues to output optimal solutions. The incorporation of Upper Bound Pruning only gets rid of sub-optimal partial solutions from OPEN.

Example 1: The project shown in Figure 1 has 8 activities. There is only one type of resource with a total availability of 7 units. The optimal schedules for m = 3 and m = 2 as found by Breadth_MRS are shown in Figures 2 and 3. When m = 3, activities 2, 3 and 4 can be processed simultaneously. But the three activities cannot be processed together when m = 2. Hence the optimum makespan is 10 when m = 3 and 11 when m = 2.

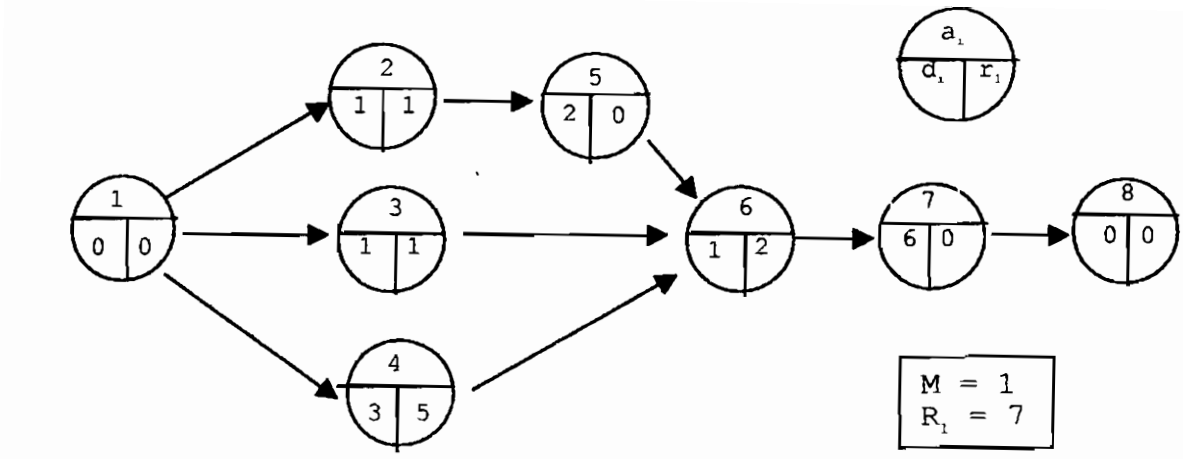


Figure 1: Project for Example 1

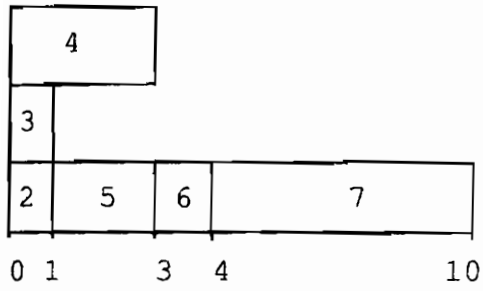


Figure 2: Optimal Schedule for m = 3

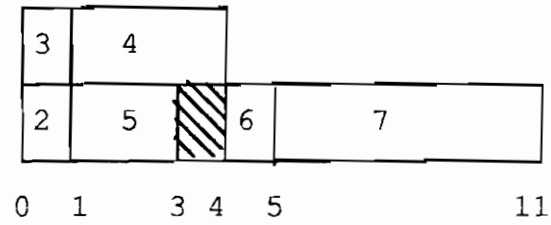


Figure 3: Optimal Schedule for m = 2

To find an optimal schedule, what is a good way to proceed when m is a small integer? The following procedure gives good results. Suppose $m = m_0 \leq 5$. Determine the makespan MS for the given problem for some $m > m_0$, preferably for $m = m_0+1$ but at worst for m = infinity, using the scheme described here. MS serves as a lower bound for the solution when $m = m_0$. The upper bound $U(m_0)$ can be set equal to the output GM of GET_MEASURE(m_0). Since the upper bound $U(m_0)$ and the lower bound MS are both known, the mean U_1 of the two bounds can be fed to UBDP. If the makespan for m processors is no greater than U_1 then UBDP would find it. Otherwise UBDP would indicate that no solution can be found within the given upper bound (*i.e.*, it will prune all partial schedules). In the latter case, feed the mean U_2 of U_1 and $U(m_0)$ to UBDP and run it again. This can be done repeatedly until the solution is obtained; in practice very few iterations are needed. U_1 need not be set to the mean of the upper and lower bounds at start. When the difference between the two bounds is large, the solution is generally quite close to GM, so it is better to choose a value for U_1 that is close to GM, say (GM - 2). For some problem instances

UBDP takes too long to terminate when U_t exceeds the optimal makespan. If this happens we can abort the run and try again with U_t reset to a lower value.

Example 2: To take a hypothetical example, suppose that for a given problem it is known that the makespan is 58 when $m = 5$. We would like to find the makespan for the same problem when $m = 4$. In this case MS = 58. Consider the following two cases:

Case 1: Using GET_MEASURE it is found that GM = 58 for m = 4. Here upper bound = lower bound = 58. By inference the makespan must be 58, so no more runs are necessary.

Case 2: Using GET_MEASURE it is found that GM = 60 for m = 4. Now run UBDP with U_t = 59. If the problem gets solved then the makespan must be either 59 or 58. If UBDP is unable to find a solution within an upper bound of 59 and OPEN becomes empty, we infer that the makespan must be 60. In either situation no more runs are needed.

3 Experimental Results: The PROGEN problems were run on a UNIX-based RS 6000 machine with 512 MB of main memory as well as on a Linux-based 350 MHz Pentium II PC with 128 MB RAM. Makespan was the performance measure. The three algorithms of interest to us were the following: i) BSP: Best_MRS as in [Nazareth *et al* 1999] adapted for m parallel machines with all three pruning rules. ii) UBDP: the UBDP algorithm as given above, i.e., Breadth_MRS for m parallel machines with all three pruning rules plus Upper Bound Pruning; iii) BDP: UBDP as given above with all three pruning rules but *not* with Upper Bound Pruning. All three algorithms were programmed in C. A problem was deemed *unsolved* if all three of BSP, BDP and UBDP ran out of memory during every attempt at solution; if one of BSP, BDP, UBDP ran to termination and gave a positive output, the problem was deemed *solved*. The upper bound GM obtained from GET_MEASURE run for 20,000 iterations was supplied as input to UBDP. In a small number of cases, UBDP had to be called more than once with different upper bounds to solve the problem, as explained in Section 2.5. The results for m = infinity were actually obtained with m set to a large value (typically 20). Since a PROGEN problem never has more than 42 activities, and since there are precedence and resource constraints, the number of activities that can be processed simultaneously is never large.

Tables I and II show the numbers of problems of the PROGEN First and Second Lots that were solved on the RS 6000. A few of the problems could not be solved for some values of m, typically m = 3 or 4. Table III shows the problems that could not be solved on the RS 6000 machine and the corresponding values of m. A few additional problems remained unsolved on the Linux-based PC as shown in Table IV because of memory or time constraints.

Table I: Summary Results: PROGEN First Lot

Machine: UNIX-based RS 6000
Number of Problems = 480

No	Program/Method	Number of Problems Solved				
		m = inf	*m = 5*	*m = 4*	*m = 3*	*m = 2*
1.	Both BDP & BSP	478	472	457	384	356
2.	BDP but not BSP	2	3	15	81	124
3.	BSP but not BDP	0	3	4	0	0
4.	Inference using upper bound	-	0	0	0	-
5.	UBDP but not 1-4 above	-	2	2	7	-
6.	Unsolved	0	0	2	8	0

Table II: Summary Results: PROGEN Second Lot

Machine: UNIX-based RS 6000
Number of Problems = 200

No	Program/Method	Number of Problems Solved				
		m = inf	*m = 5*	*m = 4*	*m = 3*	*m = 2*
1.	Both BDP & BSP	200	186	161	104	72
2.	BDP but not BSP	-	4	12	63	119
3.	BSP but not BDP	-	5	5	1	0
4.	Inference using upper bgound	-	2	1	0	0
5.	UBDP but not 1-4 above	-	2	13	9	0
6.	Unsolved	0	1	8	23	9

Table III: PROGEN Problems Unsolved on RS 6000

Lot	Prob Nos	*m*
First	12, 58, 73, 79, 95, 100	3
	107, 118	3, 4
Second	21	2
	22, 25, 27	2, 3
	23, 24, 26, 29	2, 3, 4
	30	2, 3, 4, 5
	28, 39, 49, 72, 113, 121, 123	3
	127, 151, 159, 178, 188	3
	42, 73, 175	3, 4

Tables I-IV show that only a small number of PROGEN problems remain unsolved on the RS 6000. Out of a total of 2,400 + 1,000 = 3,400, only 10 + 41 = 51 could not be solved. The problems are easiest when m is infinity and hardest when $m = 3$; as m decreases from 3 to 2, the

Table IV: Additional Problems Unsolved on Linux-Based Pentium PC

Lot	Prob Nos	m
First	115	3
Second	42, 49, 73, 151, 188	2
	28, 123	2, 4
	21	3, 4
	114, 173, 179, 180, 189	3
	22, 25, 27	4

Table V: Runtimes on RS 6000 of BSP, BDP and UBDP: PROGEN First Lot

Number of Problems = 480

	Number of Problems Solved					
Time (secs)	m = infinity			m = 3		
	BSP	BDP	UBDP	BSP	BDP	UBDP
0 -1	414	360	-	175	93	-
1 - 10	39	66	-	109	105	-
10 - 100	21	34	470	84	108	361
100 - 1000	4	18	9	16	88	78
1000 - 10000	0	2	1	0	66	32
10000 - 100000	0	0	0	0	5	0
Unsolved	2	0	0	96	15	9

Table VI: Runtimes on LINUX-Based PC of UBDP: PROGEN First Lot

Number of Problems = 480

	Number of Problems Solved	
Time (secs)	m = infinity	m = 3
0 -1	-	-
1 - 10	-	-
10 - 100	476	406
100 - 1000	4	57
1000 - 10000	0	7
10000 - 100000	0	0
Unsolved	0	10

problems again get easier. Of the 21 + 68 = 89 instances not solved by BDP or BSP, only 3 were solved by direct inference using the upper bound, and 11 + 24 = 35 were solved by UBDP.

What do we gain by using UBDP instead of just BDP (or BSP)? Clearly, UBDP serves no useful purpose when a problem is easy to solve. When problem instances are difficult, UBDP is able to solve some instances that BDP cannot solve. Tables V and VI give the distribution of

runtimes of BSP, BDP and UBDP on the RS 6000 machine and on the Linux-based Pentium PC for m = infinity and $m = 3$. The time taken by GET_MEASURE, which lies in the range 30-50 secs for 20,000 iterations, is included in the runtime of UBDP. So UBDP never takes less than 30 secs on any problem. We observe that:

i) BSP is much faster than BDP or UBDP.

ii) There are quite a few problems that BDP can solve but BSP cannot.

iii) When the problems are very difficult, UBDP takes less time than BDP.

It follows that, given a problem of unknown difficulty, an attempt should first be made to solve it using BSP. If m = infinity, the value of DL would provide some guidance about the difficulty level; for $2 \leq m \leq 5$, some indications can be obtained from the schedule for m = infinity. Within 600 secs (10 mins) on the RS 6000, BSP will either solve the problem or run out of memory. In the latter case, we should rerun the problem using GET_MEASURE and UBDP when $3 \leq m \leq 5$, and using BDP alone when $m = 2$.

The runtime of UBDP is a function of the upper bound that is fed to it. Table VII shows how the runtime of UBDP varies with the upper bound for four selected problems. Six values of the upper bound were chosen. One was a very high value; at such a high value, UBDP was effectively running without an upper bound. The five other values lay clustered around the makespan for the indicated value of m. When the upper bound is greater than or equal to the makespan, UBDP outputs the makespan assuming it does not run out of memory and runs to termination. When the upper bound is less than the makespan, UBDP outputs 0 indicating that OPEN has become empty. In this experiment GET_MEASURE played no role, so the runtime shown is for UBDP alone. We notice that the runtime decreases steadily as the upper bound decreases. In some cases the decrease is very rapid. For example, Problem 115 could not be solved by UBDP with the upper bound = 1,000, but the runtime came down to a small fractional value when the upper bound fell below the makespan. In other cases, the decrease in runtime is more gradual (see Problem 44). Problems 95 and 99 show that the runtime is may not be small even when the upper bound < makespan.

The three problems 107, 118 and 121 of the PROGEN First Lot form an interesting set (see Table VIII). These were solved using UBDP, except for Problems 107 and 118 for m = infinity when BDP proved adequate. Whenever UBDP was run, the respective upper bound GM obtained from GET_MEASURE when run for 20,000 iterations was supplied as input; the running time of UBDP includes the time taken by GET_MEASURE. Among the three, Problem 121 is by far the hardest when m = infinity. In fact, when m = infinity, UBDP takes longer to run on Problem 121

than on any other problem of the First Lot. But as m decreases, Problems 107 and 118 suddenly become very difficult. For m = 3 and 4, UBDP is unable to solve these two problems. In contrast, the runtime of Problem 121 increases slowly as m decreases. When m = 2, all three problems are solved by UBDP, but Problems 107 and 118 have much larger runtimes than Problem 121.

Why is the nature of variation of runtime with m so different for the three problems? To get an answer we need to look at the optimal schedules for the m = infinity case. We find that for Problem 121 at most four activities are running simultaneously at any instant of time. This

Table VII: UBDP: Variation of Runtime with Upper Bound
Selected PROGEN First Lot Problems
Machine: UNIX-Based RS 6000

Prob No	m	Makespan		Upper Bound and Runtime					
44	inf	63	Upper Bound	1,000	65	64	63	62	61
			Time (secs)	75	24	19	15	11	7.5
95	4	43	Upper Bound	1,000	45	44	43	42	41
			Time (secs)	6,050	1,282	779	463	212	72
99	3	51	Upper Bound	1,000	53	52	51	50	49
			Time (secs)	10,766	3,129	2,100	1,245	572	193
115	4	47	Upper Bound	1000	49	48	47	46	45
			Time (secs)	-	7,983	3,807	1,124	0	0

Table VIII: Runtime for Problems 107, 118 and 121
First Lot of Kolisch *et al.*(1995)
Machine: UNIX-Based RS 6000

Prob No	Runtime of BDP or UBDP in secs				
	$m = inf$	$m = 5$	$m = 4$	$m = 3$	$m = 2$
107	4.7	1,446	-	-	10,546
118	0.03	6,801	-	-	22,927
121	1,237	1,437	1,466	1,899	1,758

means that not more than four activities get processed simultaneously because of other resource constraints even when there is no restriction on the number of processors. As a consequence, the runtime of UBDP changes only slightly as m is decreased from infinity to 4. The additional number of MRSs that get formed as m is decreased causes the slight increase in runtime. For Problems 107 and 118, the resource constraints when m = infinity are far less tight; so these are solved readily by BDP. At most eight activities are processed simultaneously in Problem 107 and at most 11 in Problem 118. As soon as m is decreased to 5, the constraint on the number of processors causes a

sudden spurt in the number of MRSs, making the problems very difficult. When m decreases to 2, the number of MRSs gets reduced for all three problems because the constraint on the number of processors becomes the overriding constraint and other resource constraints play a minor role.

In our experiments on the 480 problems in the First Lot of PROGEN, we were able to solve 465 on the RS 6000 machine for all values of m, i.e., for m = 2, 3, 4, 5, and infinity. We examined the nature of variation with m of the average runtime for these 465 problems. For a specific problem instance, we expected a functional relationship of the form

$$T = A + exp(-\beta\ m).\Phi\ (m)$$

where T is the runtime, A and β are real numbers independent of m, and $\Phi\ (m)$ is a quadratic polynomial in m (see Figure 4). Clearly, T should reach a limiting value when m is large; this explains the exponential damping. Moreover, it was anticipated from the experimental results that T would have a single peak in the neighbourhood of m = 3 for most problems. For some problems such as problem 46 of the First Lot, the runtime was greater for m = 2 than for m = 3, but the number of such problems was very small. So the average runtime T_{avg} for the entire lot of 465 problems was also expected to exhibit a functional dependance with m of the indicated form. Since we had only five data points for T_{avg} we made the assumption that for values of $m \geq 9$, the runtime was the same as for m = 20. This assumption is justified because there are very few PROGEN problems in which, when m is large, more than 8 activities run simultaneously. Performing a regression analysis using the Statistica package, we were able to achieve an excellent fit for the computed values of T_{avg} by taking β = 2 and choosing $\Phi\ (m)$ to be a third degree polynomial of the form $Bm^3 + Cm^2 + Dm + E$, where B, C, D and E are real numbers independent of m. We also obtained a fairly good fit by choosing $\Phi\ (m)$ to be a quadratic polynomial. We preferred to study the variation of the average runtime rather than the individual runtimes of specific problem instances because of the great range of values of the runtime for many of the problems. There were quite a few problem instances for which the runtime for m = infinity was a fraction of a second, while the runtime for m = 3 was more than 1,000 secs. Nevertheless, a good fit was obtained with the above function for many of the individual problems, including problem 46 of the Fitrst Lot. Why the third degree polynomial expression for T_{avg} gave a better fit than the quadratic expression is still a matter of conjecture. It can be claimed however that our expectations regarding the nature of variation of T with m have been largely confirmed. It still remains to give a satisfactory theoretical explanation for the observed functional dependance of T on m.

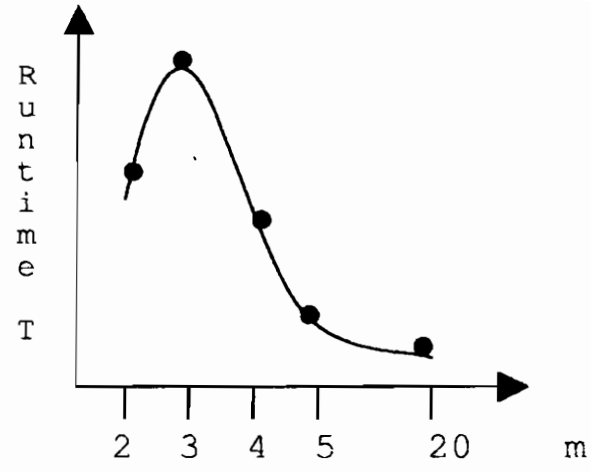


Figure 4: Variation of runtime with the number of processors

How close does the approximate algorithm GET_MEASURE get to the makespan in a given number of iterations? Table IX summarizes the picture when $m = 4$ for the problems in the PROGEN First Lot. This is quite typical; the results are similar for other values of m and for the Second Lot. Of the 480 problems, the makespan is known in 478 cases when $m = 4$, as indicated in Table I; for two problems the makespan could not be determined by any method, so it was not possible to tell whether GET_MEASURE achieved the value or not. Table IX shows that as the number of iterations is increased starting from 1000, more and more problems are solved optimally by GET_MEASURE. It is interesting to note that even when the number of iterations is only 1000, about 63 percent of the problems are solved optimally. As expected, the runtime is linear in the number of iterations and does not change with the problem. If GET_MEASURE is run for a maximum of 2,000,000 iterations until a problem is solved optimally, then the makespan value is not reached for only 40 problems. In practice of course it would be undesirable to run GET_MEASURE for too many iterations, since that would increase the runtime without guaranteeing an optimal solution. Since our interest lay in solving the harder problems using UBDP, we chose to run GET_MEASURE for 20,000 iterations and supplied the resulting GM to UBDP. For m = 4, only in nine cases out of 92 did the GM value exceed the makespan by more than three.

The runtime of GET_MEASURE for 20,000 iterations increases slightly as m is decreased, from 38.9 secs when m = infinity to 48.5 secs when $m = 2$. This runtime changes very slightly with the problem instance. Since the makespan is larger for smaller m, GET_MEASURE at Step 5 has to push activities further to the right when creating a schedule, thereby taking more time. The

Table IX: GET_MEASURE: PROGEN First Lot
Machine: UNIX-Based RS 6000

Number of Problems = 480 Number of Processors = 4

Iterations	1,000	2,000	10,000	20,000	≤ 2,000,000
Avg Time (secs)	2.12	4.18	21.62	42.41	-
Optimal	301	319	365	386	438
Non-optimal	177	159	113	92	40
Not Known	2	2	2	2	2

Table X: GET_MEASURE: PROGEN First Lot
Number of Optimal Solutions for Given Number of Iterations
Machine: UNIX-Based RS 6000
Number of Problems = 480

Iterations	Number of Problems Solved Optimally				
	m = inf	*m = 5*	*m = 4*	*m = 3*	*m = 2*
1 - 19	264	244	191	59	267
20 - 199	57	61	72	75	110
200 - 1,999	32	38	56	115	29
2,000 - 19,999	53	56	67	88	15
20,000 - 199,999	24	29	33	58	18
200,000 - 1,999,999	16	18	19	32	17
Opt Not Found in $2x10^6$ iters	34	34	40	45	24
Optimal Not Known	0	0	2	8	0

Table XI: GET_MEASURE: PROGEN Second Lot
Number of Optimal Solutions for Given Number of Iterations
Number of Problems = 200

Iterations	Number of Problems Solved Optimally				
	m = inf	*m = 5*	*m = 4*	*m = 3*	*m = 2*
1 - 19	104	94	61	29	112
20 - 199	31	31	32	25	48
200 - 1,999	27	25	29	49	8
2,000 - 19,999	18	21	21	19	7
20,000 - 199,999	7	11	17	28	3
200,000 - 1,999,999	10	13	18	14	5
Opt Not Found in $2x10^6$ iters	3	4	14	13	8
Optimal Not Known	0	1	8	23	9

optimal value is reached in 20,000 iterations for 406 problems when m is infinity and 337 problems when $m = 3$. The number increases sharply to 421 when m goes down to 2, a phenomenon that is

yet to be explained satisfactorily. Tables X and XI give the number of iterations needed to reach the optimal value for various values of m.

Does an instance of the PMRCSP get easier to solve if all resource constraints are removed except for the constraint on the number of processors? The answer is definitely *yes* when m is infinite. But for small values of m such as 2 or 3, the problems are typically harder in the absence of other resource constraints owing to the proliferation of MRSs. The change in the runtime of UBDP for three typical problems is shown in Table XII. In each case, the output of GET_MEASURE was fed to UBDP. To emphasize the change in the runtime of UBDP, the time taken by GET_MEASURE has not been taken into account. In the Resource Constraints column, we indicate whether other resource constraints, except for the one on the number of processors, are present or absent. The table shows that when other constraints are removed, the problem is very easy for m = infinity. As m decreases, in many cases the problem suddenly becomes very hard to solve, and when $m \leq 3$, the unconstrained problem is often harder than the constrained problem.

Table XIII gives the average time taken by UBDP for the problems in the First Lot with resource constraints present and with the constraints removed, the average being taken over solved problems only. When $m = 2$, the unconstrained problems frequently take too long to solve, so that value of m is not included in the table. Again, the time taken by GET_MEASURE has been ignored. For the unconstrained problems, the dramatic rise in the runtime as m decreases should be noted.

The A* search algorithm of Chang and Jiang [1994] is the only other method that tries to minimize the makespan of projects on parallel machines. However, this method assumes there are no resource constraints. Unfortunately, this method sometimes takes very long to run even on easy

Table XII: Change of Runtime of UBDP with m for Selected Problems
PROGEN First Lot
Machine: UNIX-Based RS 6000

Prob No	Runtime of UBDP in secs					Resource Constraints
	$m = inf$	$m = 5$	$m = 4$	$m = 3$	$m = 2$	
3	0.1	0.2	1.6	18.5	177.6	Yes
3	0.0	0.2	6.2	69.5	219.7	No
82	8.6	8.6	8.6	10.3	41.7	Yes
82	0.0	0.0	0.5	120.4	851.1	No
156	0.0	20.8	320.2	119.4	2,161.3	Yes
156	0.0	20.8	321.7	118.6	2,144.4	No

Table XIII: UBDP : Runtime for Constrained and Unconstrained Cases
PROGEN First Lot
Number of Problems = 480

Number *m* of Processors	Constraints Present Number Solved	Time (secs)	Constraints Absent Number Solved	Time (secs)
Infinity	480	10.4	480	0.0
5	480	32.0	480	22.8
4	478	42.3	477	51.6
3	471	277.1	469	378.5

problems. We compared the runtimes on the UNIX-based RS 6000 of the three procedures BDP, BSP, and the method of Chang and Jiang (CJ) on Patterson's benchmark set of 110 problem instances. These problems are known to be very easy. CJ was programmed in C like the other two procedures. For BDP and BSP, the resource availability was increased to a very large number so that no resource constraints were imposed when solving the problems. For CJ, resource requirements and availabilities were simply ignored.

Table XIV: Patterson's Problems without Resource Constraints
Number of Problems Solved
Machine: UNIX-Based RS 6000
Total Number of Problems = 110

No	Method	Number of Problems Solved			
		m = 5	*m = 4*	*m = 3*	*m = 2*
1.	BSP	110	110	110	110
2.	BDP	110	110	110	110
3.	CJ	109	108	90	14

Table XV: Patterson's Problems without Resource Constraints
Average Runtimes
Machine: UNIX-Based RS 6000
Total Number of Problems = 110

No	Program/Method	Time to solve in seconds							
		m = 5		*m = 4*		*m = 3*		*m = 2*	
		max	*avg*	*max*	*avg*	*max*	*avg*	*max*	*avg*
1.	BSP	0.01	0.00	0.02	0.01	4.71	0.08	29.25	0.61
2.	BDP	0.15	0.00	3.00	0.03	27.42	0.39	37.07	0.72
3.	CJ	2,505.43	62.18	3,558.62	88.93	2,960.99	125.08	0.72	0.145

As can be seen from Tables XIV and XV, CJ performs very poorly compared to BDP and BSP. The runtime of CJ is orders of magnitudes greater than that of BSP or BDP. CJ fails to solve many of the problems for $m = 2, 3$. For $m = 2$, its performance is particularly weak. It solves only 14 out of 110 problems. Although in Table XV, CJ seems to take less time on the average than BSP or BDP for $m = 2$, this is misleading since CJ solves only the very easy problems.

4. Extensions: Section 3 provides a variety of experimental results on the PMRCSP with makespan as the chosen measure. The measure can be changed to mean flow time by making appropriate alterations in the pruning rules as explained in [Nazareth *et al* 1999]. Up to now we have assumed that all activities are available for consideration by the scheduling algorithm at start time. When minimizing the mean flow time, we might want to allow an activity to become available at a later time. By the *ready time* of an activity, we mean the time at which it is available for consideration by the scheduling algorithm; it is not necessarily ready for processing at that time, since it might have predecessors that are still not processed. The *flow time* of an activity, which is the time it spends in the system, equals (finish time - ready time). The algorithms BDP, BSP and UBDP can all be modified to take account of non-zero ready times of activities. This appears at first to be a simple feature to incorporate. But we should note that when activities have non-zero ready times, there might be instants during the execution of a project scheduling algorithm when no activities are in progress. The procedures have to be adjusted to take care of this possibility [Verma 1998]. Since the activities in the benchmark problems of [Patterson 1984] and [Kolisch *et al* 1995] have zero ready times, when conducting experiments a method needs to be devised for assigning ready times to activities. One way to do this is as follows. Let us assume that the optimal schedules for the original problems have been found for the chosen performance measure. Then the start times of the activities in the original optimal schedules are known. For each activity, we can multiply the start time by a random number lying between 0.75 and 1.25; the resulting value can be assigned as the ready time of the activity. Experiments on Patterson's problems suggest that the assignment of non-zero ready times does not increase the difficulty level of problems [Verma 1998].

For measures such as maximum tardiness or the number of tardy jobs, each activity must be assigned a *due date*. If an activity completes after its due date then it is *tardy*. Again, the scheduling algorithms can be modified to work for the new measures by appropriately reformulating the pruning rules. In experiments, due dates can be assigned to activities in the same way as ready times are assigned, except that finish times of activities, rather than start times, would be have to be

considered [Verma 1998].

In which ways can OPT_MEASURE be improved? One issue that needs further consideration is the computation of the difficulty level of a problem instance. A thumb rule for the m = infinity case has been provided in Section 2.3. While this thumb rule appears to be adequate when m = infinity, it cannot be used profitably when $2 \leq m \leq 5$. No simple procedure for estimating the difficulty level of a problem is known for such values of m. This issue is important because some problems that are very easy when m = infinity suddenly become very hard when m is decreased to 5 or 4.

The performance of the approximate algorithm GET_MEASURE has been found to be fairly satisfactory. But it should be possible to get results as good as those obtained from GET_MEASURE in much less time. GET_MEASURE has one shortcoming, namely, iterations are completely independent of each other. In Simulated Annealing, the notion of the neighbourhood of a trial solution plays an important role in the generation of subsequent trial solutions. Is it possible to improve GET_MEASURE by introducing the concept of neighbourhoods of trial solutions? We have not yet succeeded in doing so.

It would be possible to incorporate GET_MEASURE in a depth-first search scheme. The upper bound obtained from GET_MEASURE can be used for pruning unpromising states until a better solution is discovered by the depth-first search. For example, it would be possible to combine GET_MEASURE with the algorithm of [Demeulemeester and Herroelen 1992]. This aspect has not yet been investigated in detail.

It was anticipated that the average runtime T of a set of problem instances would exhibit a certain functional dependance on the number m of processors. Experimental results seem to confirm that the conjectured functional form is the correct one. But no satisfactory theoretical justification for this phenomenon has yet been found. This appears to be a very interesting topic for further investigation.

References

Chang P C, and Jiang Y S [1994]: A state-space search approach for parallel processor scheduling problems with arbitrary precedence relations, *European Journal of Operational Research*, 77, 208-223

Cho J-H, and Kim Y-D [1997]: A simulated annealing algorithm for resource constrained project scheduling problems, *Journal of the Operational Research Society*, 48, 736-744

Christofides N R, Alvarez-Valdes R and Tamarit J M [1987]: Project scheduling with resource constraints: A branch-and-bound approach, *European Journal of Operational Research*, 29, 262-273

Demeulemeester E, and Herroelen W [1992]: A branch-and-bound procedure for the multiple resource-constrained project scheduling problem, *Management Science*, 38, 1803-1818

Demeulemeester E, and Herroelen W [1997]: New benchmark results for the resource-constrained project scheduling problem, *Management Science*, 43, 1485-1492

French S [1982]: *Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop*, Ellis Horwood

Garey M R and Johnson D S [1979], *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman

Icmeli O and Erenguc S S [1996]: A branch-and-bound procedure for the resource-constrained project scheduling problem with discounted cash flows, *Management Science*, 42, 1395-1408

Icmeli-Tukel O and Rom W [1997]: Ensuring quality in resource constrained project scheduling, *European Journal of Operational Research*, 103, 483-496

Kolisch R, Sprecher A and Drexl A [1995]: Characterization and generation of a general class of resource-constrained project scheduling problems, *Management Science*, 41, 1693-1703

Mori M and Tseng C C [1997]: A genetic algorithm for multi-mode resource constrained project scheduling problem, *European Journal of Operational Research*, 100, 134-141

Nazareth T, Verma S, Bhattacharya S and Bagchi A [1999]: The multiple resource constrained project scheduling problem: A breadth-first approach, *European Journal of Operational Research*, 112, 347-366

Patterson J H [1984]: A comparison of exact approaches for solving the multiple constrained resource project scheduling problem, *Management Science*, 30, 854-867.

Stinson J P, Davis E W and Khumawala B M [1978]: Multiple resource constrained scheduling using branch-and-bound, *AIIE Transactions*, 10, 252-259

Verma Sanjay [1998]: *Improved Methods for Scheduling Partially Ordered Jobs Under Resource Constraints*, Unpublished Fellow Programme Thesis, Indian Institute of Management Calcutta