Overheads in Loosely Coupled Parallel Search

E. Altmann
T. Breitkreutz
and
T.A. Marsland

Computing Science Department
University of Alberta
Edmonton,
Canada T6G 2H1

Technical Report TR87.15

ABSTRACT

The vertex cover problem is identified as a task less complicated than chess that exhibits similar overheads when solved using loosely coupled parallel systems. For both problems, pruning can be applied to the search trees, causing the trees to be skewed. Skewed trees lead to overheads in parallel search because scheduling work to keep all processors productive is difficult. The combined overheads comprise solution time *overrun*, the amount by which a solution time is slower than linear speedup. Here, results from another study of parallel vertex cover solutions are replicated, and additional experiments are done to gain insight specifically into communication and synchronization losses. An improvement to the simple vertex cover algorithm used in the original study is presented, and discussed with respect to its parallel adaptation.

Acknowledgements

Financial support from the Canadian Natural Sciences and Engineering Research Council through Grant A7902 made the experimental work possible.

July 27, 1987

Overheads in Loosely Coupled Parallel Search

E. Altmann
T. Breitkreutz
and
T.A. Marsland

Computing Science Department
University of Alberta
Edmonton,
Canada T6G 2H1

Technical Report TR87.15

1. Introduction

Using a loosely coupled network of processors is a simple way to increase the processing power available to a searching application. Despite their simplicity, such systems can exhibit heavy overheads that undermine their efficiency and limit their effective speed. The overheads fall into three broad categories:

(a) Communication overhead, where processors wait while information that may improve their efficiency is exchanged. Typically this involves the updating or retrieving of data from a global shared table, or time spent sending and receiving messages.

(b) Search overhead, where more nodes are searched in the parallel implementation than in a sequential one. One form of search overhead occurs when processors do redundant calculations. Because work is not being done in the strictly sequential style of a single processor, information must be shared if processors are to detect and avoid the duplication of work; redundant calculations suggest that information is not being adequately shared.

   Another form of search overhead occurs if work is deliberately assigned to processors on a speculative basis, that is, in the absence of anything better for them to do. Here some duplicated effort is acceptable, being a calculated loss.

(c) Synchronization overhead, where processors become idle after completing their assigned work, and cannot continue until some (even all) others finish completely. Clearly, while processors are idle the effective speed of the system is decreased.

Normally there is a trade-off between search, communication, and synchronization overheads. For example, if speculative computing is employed, there will be some increase not only in communication but

possibly also search overhead.

2. Overheads and Effective Power of Parallel Solutions

The power of parallel solutions is often demonstrated via the solution of classical combinatorial problems [10]. Especially popular is the traveling salesman problem, since all combinations may have to be searched and hence nearly linear speed-up is possible [7]. Almost all combinatorial search problems are well-suited to a multiprocessor solution, especially if:

(a) Most work is independent (i.e., negligible data sharing is needed and the calculations may be carried out in any order).

(b) Most work requires a predictable amount of time, so that nearly perfect processor scheduling is possible.

If search tree pruning techniques are present in a sequential solution, then, in parallel adaptations, the information that leads to cutoffs must be shared between processors. Information-sharing entails communication loss; more importantly, synchronization overhead arises in part from the unpredictability of chunks of work, whose size may change dynamically when pruning occurs. Thus pruning techniques, while improving sequential solutions, can reduce the power of a multiprocessor by contributing to overheads.

Note that the effective power of parallel solutions is overestimated if the uniprocessor solutions against which they are compared are not the fastest available. For example, consider a pure minimax search of a uniform tree. Simple tree-splitting will yield close to ideal speedups: there is no search overhead, since no pruning occurs; there is no communication overhead, since no information is shared; assuming constant cost per node, there is no idle time, since each chunk can be made equal-sized. However, when examined in the light of a uniprocessor version enhanced with alpha-beta pruning, the speedups derived from applying tree-splitting to pure minimax search become less appealing.

Given that pruning effects are equally desirable in a parallel solution, it is necessary to devise methods for controlling the overheads that stem from them. Such methods include (a) intelligent use of speculative computing, (b) allowing for dynamic processor configurations (i.e. allowing communication paths to change), so that idle processors can be assigned to those that are still busy, and (c) designing static processor configurations that are well-suited to a given application. Method (c) has the advantage of code simplicity, and was our choice for the experiments reported here.

Our interest is the investigation of synchronization loss, from which computer chess programs suffer most severely. Control of this overhead is made especially difficult in the chess case by the complexity of the program itself and the difficulty in subdividing the work into smaller chunks that can be distributed

more uniformly across all processors. Therefore our purpose here was to find a simpler application that exhibits a similar serious synchronization overhead. One such application is the vertex cover problem: given an undirected graph, find the smallest set of vertices such that every edge in the graph is incident to at least one vertex in the set [3].

3. Vertex Cover

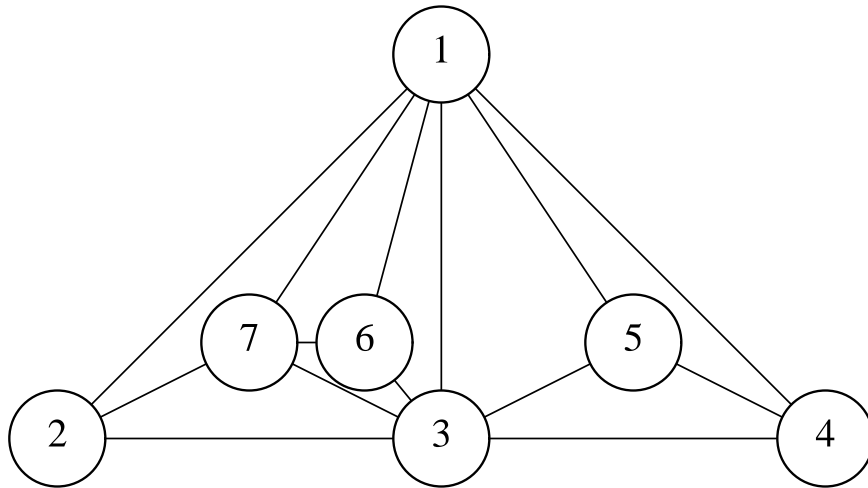
3.1. Background Research

A multiprocessor solution to the vertex cover problem was one of several applications for the MANIP architecture proposed by Wah and Ma [9]. Although vertex cover was only briefly discussed, they presented some salient features of the problem. They showed that for an N-node graph a tree of maximum depth of $N - 1$ must be searched. The search tree is strictly tapered, in that the branching factor decreases by 1 for each successively deeper level. Also they noted that the search tree, when pruning is applied, is skewed to the left (as indeed are game trees). Most usefully, they presented a bounding rule that for sparse graphs provides a good lower bound on the remaining vertices needed to complete a solution (see Appendix A for pseudocode based on Wah and Ma's outline [9]). Since no implementation details or other helpful heuristics to speed the search were provided, there was little basis on which to do a comparative study on the vertex cover problem *per se*.

A follow-up study by Zariffa [11] used a 7-processor Data General based system to gain working experience with some pragmatic aspects of multicomputer systems. In that thesis 15 graphs were searched with 2, 4 and 7 processors under three different processor scheduling schemes. Fixed First Level (FFL) scheduling assigned fixed partitions of the first level tree nodes to each processor; its sole advantage was an absence of communication overhead. Dynamic First Level (DFL) scheduling assigned a first level tree-node to each processor, then dynamically assigned the remaining ones on a first-come, first-served basis. Dynamic First and Second Level (DFSL) scheduling used all processors to expand the most promising first level tree node, then pooled that node's children and the remaining first level tree nodes for dynamic allocation. DFL, both simple and the most effective, is the scheme on which we base our comparison.

Note that Zariffa's parallel solutions use a simple but non-optimal search algorithm. Her search trees branch solely on the choice of which unused vertex to select next; that is, the children of a tree node correspond to vertices that have not been selected previously on the path from the root. But the selection order of the vertices is irrelevant with respect to their inclusion in a solution; therefore, if multiple paths from the root include the same vertices in permuted order, all such paths but one are redundant. Zariffa's search trees contain a path from the root for each permutation of a given set of vertices. Such trees are

easily split because, without pruning, they are symmetric: every search path from the root can have a length up to $N - 1$. This symmetry allows equivalent chunks of work to be assigned to each processor. In contrast, when redundant paths are eliminated by a process we term *duplicate path elimination* (or simply *dpe*), the tree becomes sharply skewed to the left and significantly smaller. The asymmetry of the *dpe* tree renders at least Zariffa's FFL scheduling algorithm clearly ineffective, since partitions of equal numbers of first level tree nodes will vary greatly in total number of nodes to be searched per partition. Furthermore, the decreased size of the *dpe* tree entails a significant decrease in search time for the uniprocessor case, as the results in Table 1a, presented later, confirm. Thus Zariffa's speedups, obtained using non-*dpe* trees, do not necessarily reflect with accuracy the effective power of parallel vertex cover solutions, since they are not calculated using the best available uniprocessor algorithm. In this report, however, we have first replicated Zariffa's results using non-*dpe* trees to provide direct comparison of the two computer systems.



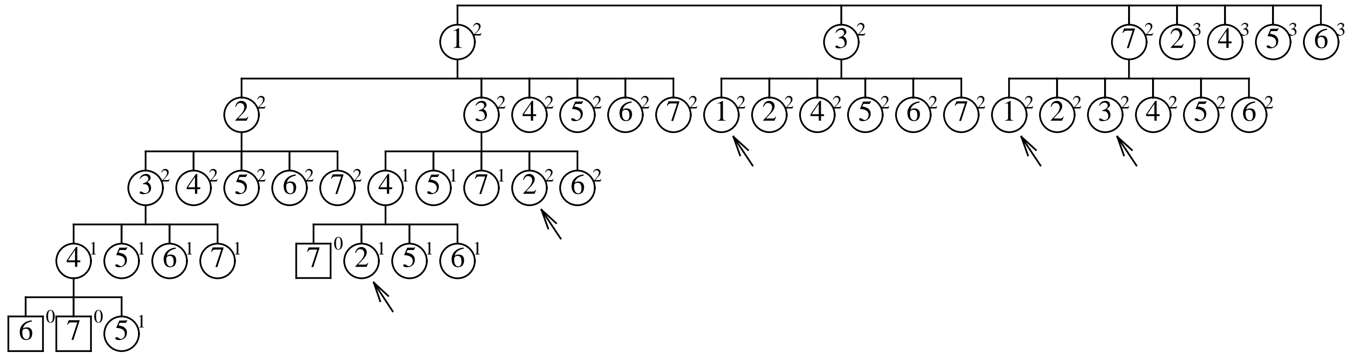


Figure 1: Sample Graph and Duplicate Path Elimination

Consider Figure 1, which illustrates the effect of *dpe*. A sample graph and its vertex cover search tree are presented. In the figure, square nodes represent solutions, and as such terminate search paths. The node superscripts are lower bounds on the size of the solution; a node whose lower bound plus depth is not less than the best solution found so far (in a left-to-right traversal) is terminated. Several duplicate paths appear in the tree; arrows point to nodes eliminated by *dpe*. Note that *dpe* has much greater impact in larger search trees, where eliminated nodes are the roots of large subtrees.

3.2. Replication of Prior Work

3.2.1. Implementation Details

The algorithm for solving the vertex cover problem is given in Appendix A, as is the function for calculating Wah and Ma's lower bound. Here we present implementation factors that bear on the analysis that follows in Section 3.2.2.

3.2.1.1. Non-homogeneous Processor Systems

Zariffa's hardware ([11], p. 32) consisted of Data General Nova 4's and one Nova 3. The system was non-homogeneous, the Nova 3 being roughly 10 percent faster than the others. However, because the Nova 3 provided disk access for the system, it participated in all Zariffa's experiments.

Our system had an analogous feature: six Motorola 68010's, running with limited operating system support, were roughly 20 percent faster than the seventh, which ran under UNIX.† The UNIX-based

† Registered trademark of AT&T in the USA and other countries.

processor (named *sunshine*) provided disk access for the system. To reduce operating system overhead to a minimum, the kernel of the other processors (referred as *standalones*) supported Ethernet I/O only, and did not support multiple application processes. *Sunshine* necessarily participated in all the replication experiments, but usually only as a master data gatherer.

3.2.1.2. Processor Configurations

The configuration of processors (i.e. the communication paths established between them) was implemented differently in our study than in Zariffa's. Constraints imposed by the hardware on which her experiments were conducted dictated that a ring configuration was the simplest and most effective. In her form of ring, a record of control information (e.g. a list of completed chunks of work) is maintained by each processor. When the record changes (e.g. upon the selection of a new chunk by a processor), the updated record is passed around the ring.

Because P copies of the control information are maintained, updating the information requires that a packet travel $P-1$ successive hops before the update is complete. In Zariffa's implementation, an update was delayed at least 10 milliseconds before reaching the last processor. This latency made it possible for nodes to be expanded redundantly, given that processors referred to their own copies of control information when looking for more work. That is, if two or more processors claimed new chunks of work within the period of communication latency, then the same chunk would be claimed by each, because the updates would arrive too late. Apparently this form of redundant search overhead was encountered by Zariffa in her experiments ([11], p. 78).

The availability of a broadcast bus (Ethernet) allowed us greater flexibility in designing a processor configuration (also referred to as a *network architecture*). Our experiments were conducted using a single-level processor tree, in which slave processors executed the search, and spoke only with a master processor. The slave processes resided on the standalones, and the master, responsible for file I/O, resided on *sunshine*. The master also coordinated the search, maintaining the unique copy of the control record. The advantages of such a configuration are twofold:

(a) Faster broadcasting of updates. While information must travel P hops for an update (1 slave-to-master message and $P-1$ master-to-slave messages), there is no time dependence between messages, as there is in a ring. The master, executing non-blocking *send* operations (terms explained in Section 3.2.1.3), queues all outgoing messages in a tight loop, without waiting for any particular one to be sent. This time independence of messages allows the broadcast of $P-1$ messages to occur nearly simultaneously.

(b) No possibility of duplicating work on a chunk, because all references to search control information go through the master, which is a single, sequential process.

As Zariffa notes ([11], p. 35), a master/slave configuration is susceptible to message-processing bottlenecks in the master, especially at higher parallelisms. It was found that our systems did not exhibit significant communication overhead, implying that such a bottleneck is not a factor up to parallelism 7. Our findings on communication overhead are discussed in Section 3.2.2.3.

Another issue that arises from employing a master process is the selection of a physical processor on which to place it. In our system the master, given responsibility for file I/O, was placed on *sunshine* for all experiments. In our 2- and 4-processor systems a standalone processor was dedicated to each slave process. However, because of limited hardware available when our experimental work began, it was initially convenient in our 7-processor system to double up the master with a slave on *sunshine*. Later the effects of a doubled versus an independent master were explored, and are discussed in Section 3.2.2.4; for experiments involving an independent master and 7 slaves, a seventh standalone processor was employed.

3.2.1.3. Message Passing Operations

Three message passing operations occur in our systems: *polls*, *sends*, and *receives*. When a slave has finished a chunk of work, it *sends* the results to the master, then waits for new work to arrive, at which point the work is *received*. Waiting for new work occurs during a *blocking* poll executed by the slave. *Non*-blocking polls, used to check for new bounds before node expansion, involve no waiting. Sends and receives, which copy information into and out of system buffers, do not block, and hence take small and constant amounts of time.

3.2.2. Results

Zariffa's results, in terms of node counts for sequential solutions, were successfully replicated for 12 out of the 15 problem sets; problems 6, 7, and 13 yielded inconsistent results. Our solution sets were identical except for Problem 15, for which the sets differ in one node; we attribute the difference to a labeling inconsistency in Zariffa's graph representation. Zariffa's problem set and her solutions are presented verbatim in Appendix B. Table 1a gives a summary of our solution sets found using DFL, and for comparison includes both sets of measured solution times. Interestingly, our sequential solution times are roughly 20 times faster. When *dpe* is included another 15-fold improvement occurs, demonstrating the drastic reduction in tree size brought about by *dpe*.

Prob. #	Non-*dpe* Graph Size	Solution Sets	Gen.	Data General Times	Motorola Times	*dpe* Gen.	Exp.	Gen. at Soln	Motorola Times
1	13	1 2 5 7 9 11 13	4517	07:08	0:23.1	712	138	167	0.2
2	13	1 3 6 7 8 9 11	4536	07:24	0:23.3	679	130	172	0.2
3	20	2 5 8 9 12 15 16 18	5538	17:02	0:52.8	783	59	374	0.4
4	16	2 4 6 7 8 10 14	18387	40:43	2:13.8	2012	341	929	0.8
5	12	1 2 4 6 7 8 10	7249	10:59	0:33.8	736	188	112	0.2
6	15	1 3 4 5 7 9 12 14	3512	07:10	0:35.9	809	100	130	0.3
7	13	1 2 3 6 9 10 12	4756	07:35	1:08.5	917	187	68	0.4
8	12	1 2 4 5 6 7 10	15778	25:54	1:17.4	866	245	75	0.3
9	14	1 2 5 6 7 10 11 13	44707	1:33:32	4:18.5	1583	373	123	0.7
10	15	1 2 6 7 9 10 13 14	13641	28:16	1:26.4	1292	212	263	0.5
11	14	1 2 3 4 7 9 10 11	95054	3:26:56	9:00.8	2341	515	81	1.1
12	14	2 3 5 6 8 9 10 14	55665	1:55:25	5:27.0	2067	503	347	0.9
13	15	1 3 4 5 8 9 10 11	58289	2:49:08	7:36.8	2203	452	444	1.0
14	13	1 2 3 4 5 7 8 10	110763	3:44:00	9:29.6	2033	644	73	0.9
15	14	1 2 6 7 8 9 11 14	73587	2:33:27	7:07.0	2254	578	244	1.0
Notes:				*[h:]mm:ss*	*mm:ss.s*				*s.s*

Gen. is the count of nodes generated (Zariffa's measure).
Exp. is the count of nodes expanded.
at Soln is the count when the optimal solution was found.

Table 1a: Summary of Lower Bound Method (Non-*dpe* and *dpe*)

Also presented in Table 1a are various sequential node count measures for both the non-*dpe* and *dpe* cases. The method used by Zariffa was to count all nodes *generated* before pruning; an alternate method is to count only those nodes actually *expanded* in the search. One metric may be more appropriate than the other, depending on the relative costs of generating and expanding nodes. Since the count of expanded nodes is more closely related to the number of procedure calls made during search, it may better reflect cost in terms of cpu time. Both metrics are included in Table 1a. The *Gen. at Soln* column in the table is the count of nodes *generated* that was current when the optimal solution was discovered. These numbers show that the final solution is found early in the search; the majority of search is devoted to proving that the best solution cannot be improved upon. This observation has implications for the design of approximation algorithms for the vertex cover problem.

Table 1b presents our results from an alternate vertex cover algorithm, one that does without the lower bound used by both Wah & Ma [9] and Zariffa [11] for pruning the search tree. Vertices are sorted on outdegree once at the beginning of the search (using a stable sort), and chosen in order of highest outdegree first. Search along a particular path is stopped only when the path length is longer than the best solution found so far. The outdegree sorting method generates (and expands) more nodes, but performs comparably in terms of solution times, because the expensive lower bound calculation is omitted. Were

the costs of node expansions or terminal node evaluations significant, as they are in the chess case, then the lower bound method is likely to prove the better alternative.

Note that with the outdegree sorting method the solution sets are different than in the lower bound case. In most instances of the vertex cover problem several optimal solution sets exist; the one discovered first depends on the search algorithm used, because the algorithm determines the order in which search tree nodes are examined. The solution set also changes with the labeling of the problem graph. Because we were able to reproduce Zariffa's node counts in all but 3 cases we are confident that we have correctly duplicated both her method and, in most cases, her graph labelings. Nevertheless, through replicating Zariffa's work we became aware that the vertex cover problem is subject to diverse methods of solution.

Prob. #	Solution Sets	Times *ss.s*	Gen.	Exp.	Gen. at Soln
1	2 4 5 6 8 10 12	0.5	10183	4202	478
2	1 3 5 7 8 9 11	0.6	10559	4380	1329
3	2 5 8 9 12 15 16 18	15.2	401966	137986	177
4	2 4 6 7 8 10 16	1.5	41225	14894	98
5	1 3 4 6 7 9 10	0.3	5880	2536	154
6	1 3 4 5 7 9 12 14	1.7	41299	17311	4653
7	1 3 5 6 10 12 13	0.5	10023	4133	206
8	1 2 4 6 7 8 9	0.3	5867	2533	160
9	1 2 6 7 9 10 11 13	1.0	23407	10167	1272
10	1 2 6 7 9 10 13 14	1.5	39369	16432	320
11	1 2 3 4 7 10 11 12	1.0	22944	9951	267
12	2 3 5 6 8 10 11 14	1.0	22825	9911	99
13	1 3 4 5 8 9 10 11	2.0	39718	16567	887
14	1 2 3 4 5 7 8 10	0.6	12916	5815	90
15	2 3 4 6 8 9 11 14	1.3	23134	10032	623

Table 1b: Summary of Outdegree Sorting Method (using *dpe*)

3.2.2.1. Node Count Comparison

Table 2a presents a summary of Zariffa's (generated) node counts, including ratios that help quantify search overheads. Table 2b presents the corresponding node count and search overhead data from our replication. A point to note in Table 2a is that for the 2-processor case there is a slight degree of search overhead for all problems (that is, the *Ratio of Nodes Generated* is strictly less than 1). This is to be expected, but later Table 3a presents contradictory data from Zariffa's thesis, where solution time speedups greater than 2 were often reported for a 2-processor system.

Prob. #	Nodes Generated: Number of Processors 1	2	4	7	Ratio of Nodes Generated 1 : 2	1 : 4	1 : 7
1	4517	4714	4651	4980	0.96	0.97	0.91
2	4536	4568	4798	5208	0.99	0.95	0.87
3	5538	5881	6567	7792	0.94	0.84	0.71
4	18387	20595	20492	20968	0.89	0.90	0.88
5	7249	7346	7405	7546	0.99	0.98	0.96
6	3512	3627	3756	4482	0.97	0.94	0.78
7	4756	4825	5977	6390	0.99	0.80	0.74
8	15778	15822	15910	16042	1.00	0.99	0.98
9	44707	44784	44975	45226	1.00	0.99	0.99
10	13641	13934	14335	16011	0.98	0.95	0.85
11	95054	95084	95144	95234	1.00	1.00	1.00
12	55665	56647	58515	61431	0.98	0.95	0.91
13	58289	74808	71101	71374	0.78	0.82	0.82
14	110763	110790	110852	110949	1.00	1.00	1.00
15	73587	74037	74953	73950	0.99	0.98	1.00
mean (calculated down)					0.96	0.94	0.89

Table 2a: Data General Node Counts and Search Overheads†
† -- Data from Zariffa, p. 77, Table 4.10 [11].

Prob. #	Nodes Generated: Number of Processors 1	2	4	7	Ratio of Nodes Generated 1 : 2	1 : 4	1 : 7
1	4517	4701	4671	4837	0.96	0.97	0.96
2	4536	4647	4820	5014	0.98	0.94	0.90
3	5538	5861	6507	7404	0.94	0.85	0.73
4	18387	20728	18259	19616	0.89	1.01	0.93
5	7249	7334	7410	7561	0.99	0.98	0.96
6*	5650	5651	5721	5783	1.00	0.99	0.96
7*	13815	13831	13863	13876	1.00	1.00	0.99
8	15778	15810	15868	15953	1.00	0.99	0.99
9	44707	44770	44918	45117	1.00	1.00	0.99
10	13641	13913	14315	15079	0.98	0.95	0.91
11	95054	95070	95102	95126	1.00	1.00	1.00
12	55665	56642	58641	61138	0.98	0.95	0.90
13*	72783	74837	71109	71188	0.97	1.02	1.02
14	110763	110777	110818	110823	1.00	1.00	1.00
15	73587	74024	74916	74022	0.99	0.98	1.00
mean (calculated down)					0.98	0.98	0.95
mean (calculated across)					0.99	0.99	0.98

Table 2b: Motorola Node Counts and Search Overheads

Small inconsistencies arose in the node counts for the three problems starred in Table 2b; in each case Zariffa's sequential node counts are lower. Enumerating possible causes of these inconsistencies, we have considered the following:

(a) An undisclosed heuristic employed by Zariffa that results in more cutoffs in special cases. This is an unlikely cause, because of the high overall degree of consistency in results.

(b) Erroneous numbers in Zariffa's Table 4.10 ([11], p. 77). Such mechanical errors are possible if the table in question is produced by hand. To eliminate this potential hazard, our tables were machine-generated.

(c) Errors in the representation of the problem graphs. A separate communication [8] confirmed that Zariffa presented the wrong graph for Problem 7. In the other two cases arcs or vertices may have been omitted, or extra ones included; in Problem 6 there is at least one arc that could be removed without altering the vertex cover, although the work done might change.

Because there are small typographical errors in the thesis, and for lack of better information, we assume that (c) is the cause of the inconsistencies. These apparent inconsistencies are unimportant and question neither Zariffa's conclusions nor our own techniques.

Another notable difference between Table 2a and Table 2b is that our search overheads, presented in the *Ratio of Nodes Generated* column of the latter, are consistently lower and less variable than Zariffa's. For search overheads to be lower given identical splitting algorithms, new bounds must have been communicated to processors more quickly, so that uninteresting subtrees were recognized sooner. Faster I/O over our 10 Mbit/second Ethernet is one possible source of speed differential. Another source is the network architecture of processors (discussed in Section 3.2.1), illustrating the advantages of a broadcast bus over a loop.

3.2.2.2. Speedup Comparison

Table 3a presents timing and speedup figures for Zariffa's study, and Table 3b presents the corresponding data from our work. Note the large discrepancy between Zariffa's solution times and our own; we attribute this to the use of non-optimized, possibly interpreted, Fortran software on the Data General systems. In contrast, our algorithms were programmed in C and benefited from the use of an optimizing compiler.

Prob. #	Times (*[h:]mm:ss*) for N processors. Number of Processors 1	2	4	7	Speedup with N processors. 2	4	7
1	07:08	03:40	01:59	01:23	1.95	3.60	5.16
2	07:24	03:41	02:23	01:50	2.01	3.10	4.04
3	17:02	10:00	07:09	04:24	1.70	2.38	3.87
4	40:43	23:14	11:09	06:51	1.75	3.65	5.94
5	10:59	05:25	02:58	01:54	2.03	3.70	5.78
6	07:10	03:32	02:13	01:37	2.03	3.23	4.43
7	07:35	03:46	02:41	01:33	2.01	2.83	4.89
8	25:54	12:58	07:24	05:28	2.00	3.50	4.74
9	1:33:32	46:47	28:37	19:18	2.00	3.27	4.85
10	28:16	14:28	08:59	05:45	1.95	3.15	4.92
11	3:26:56	1:41:27	54:58	34:49	2.04	3.76	5.94
12	1:55:25	1:01:08	36:30	24:33	1.89	3.16	4.70
13	2:49:08	1:25:03	42:39	28:23	1.99	3.97	5.96
14	3:44:00	1:47:53	57:25	39:59	2.08	3.90	5.60
15	2:33:27	1:14:22	41:03	25:33	2.06	3.74	6.01
mean (calculated down)					1.97	3.40	5.12

Table 3a: Data General Timing and Speedup Results†

† -- Data from Zariffa, p. 67, Table 4.4 [11].

Prob. #	Times (*ss.ss*) for N processors. Number of Processors 1	2	4	7	Speedup with N processors. 2	4	7
1	23.14	13.70	8.47	6.60	1.69	2.73	3.51
2	23.26	13.54	9.35	7.57	1.72	2.49	3.07
3	52.76	34.73	24.63	15.71	1.52	2.14	3.36
4	133.76	83.66	37.56	25.44	1.60	3.56	5.26
5	33.80	18.92	10.74	7.72	1.79	3.15	4.38
6	35.94	20.28	11.26	11.78	1.77	3.19	3.05
7	68.54	36.88	22.02	18.86	1.86	3.11	3.63
8	77.37	40.30	22.86	17.48	1.92	3.38	4.43
9	258.50	147.95	90.75	58.63	1.75	2.85	4.41
10	86.36	48.62	29.78	24.16	1.78	2.90	3.57
11	540.76	288.64	158.59	104.76	1.87	3.41	5.16
12	327.00	188.60	109.39	74.94	1.73	2.99	4.36
13	456.82	251.36	127.61	97.11	1.82	3.58	4.70
14	569.58	310.72	163.50	111.86	1.83	3.48	5.09
15	426.97	226.05	128.94	81.56	1.89	3.31	5.24
mean (calculated down)					1.77	3.08	4.21
mean (calculated across)					1.81	3.26	4.69

Table 3b: Motorola Timing and Speedup Results

In comparing average speedups in these two tables, ours are seen to be uniformly less than Zariffa's. We have identified three possible sources of this reduction:

(a) Magnification of overheads in our implementation. Search overheads, however, clearly do not slow down our system. In fact, Tables 2a and 2b show from the node count ratios that the search overhead is lower than Zariffa's, thus tending to offset, rather than enhance, the discrepancy. Communication overheads are equally unlikely to contribute to a slowdown, in part because their magnitude is negligible (as discussed in Section 3.2.2.3), and in part because of the relative efficiency of our Ethernet and network architecture. This leaves synchronization overhead, but this loss is primarily a function of the splitting algorithm and the problem instance at hand, both of which are (for practical purposes) identical with Zariffa's. It is possible, but considered unlikely, that some of the various technical differences between our system and Zariffa's (such as the 20-fold increase in serial solution speed) interact in some unexplained fashion to increase synchronization overhead.

(b) Effects of start-up overhead and different positioning of timing checkpoints. If these were the sources of apparent slowdown, then larger problems would produce higher speedups, as these constant factors become overwhelmed. There is indeed a correlation between problem size and speedup, particularly in the 7-processor case. However, this correlation exists in Zariffa's results as well, and even to a more noticeable degree.

(c) Calculation or other mechanical error. Inaccurate estimates of sequential times would account for the discrepancies across parallelisms evident in the data. Zariffa's quoted sequential times are derived from a calculation involving averaging of results from two processors with different speeds. This derivation is inherently suspect, relative to direct measurement of times.

It is our inference that (c) is the most likely source of speedup differences; that is, Zariffa's sequential times may embody a margin of error large enough to account for the observed discrepancies. This inference is supported by an apparent contradiction in Zariffa's results. For the 2-processor case she reports acceleration anomalies (speedups greater than degree of parallelism); this data appears in Table 3a. Such anomalies are not rare and have been reported frequently [2, 4, 5]; however, there is no supporting evidence of reduced search overhead from Table 2a. For an acceleration anomaly to occur in the presence of positive search overhead, the average cost of expanding a node must decrease in the parallel case, a phenomenon difficult to account for. Note that an error of only 3 to 5 percent in the measurment of a single processor time would account for the contradictions in the data.

We observe also that the correlation between problem size and speedup, mentioned in (b) above, points to a need to experiment with larger graphs: synchronization losses may decrease with increasing problem size. Furthermore, larger problems can reduce the effect of measurement error. Our companion

study deals with the generation of larger graphs having common statistical properties, and explores a variety of binary-tree processor configurations for solving such problems [1].

3.2.2.3. Estimating Communication Overhead

Tables 4 and 5 show a break-down by processor of idle times and nodes searched, derived from our experiments. The individual processors are labeled A, B, ..., F, G. Idle times are measured by clocking from when a slave completes one chunk to when it gets another; specifically, the interval timed is that through which a slave waits on a blocking poll to the master. As such, this idle time measure includes the cost of communication overhead; send operations to the master do not block, so the time spent by a slave blocked on the poll incorporates all message processing time for slave-to-master messages. It also incorporates processing time for messages in the reverse direction, because the poll does not return until the slave's message buffer has new contents. Note that in Table 4 (the 4-processor case) all processors are dedicated to slaves, while in Table 5 (the 7-processor case) Processor G also executes the master process. Thus it is reasonable that Processor G is capable of less work as a slave; this is borne out by its lower-than-average node counts.

	Proc. A		Proc. B		Proc. C		Proc. D	
#	Gen. Nodes	Idle *ss.ss*	Gen. Nodes	Idle *ss.ss*	Gen. Nodes	Idle *ss.ss*	Gen. Nodes	Idle *ss.ss*
1	1328	2.55	1237	2.92	924	2.46	1182	**1.18**
2	1206	1.86	1039	2.81	1473	**0.33**	1102	2.33
3	2403	**0.29**	1371	10.59	1367	10.66	1366	10.66
4	4274	6.73	4293	6.52	5161	**0.30**	4531	4.71
5	1971	0.81	1564	2.76	1839	1.39	2036	**0.35**
6	1425	1.70	1630	**0.34**	1142	3.56	1524	0.95
7	2788	7.55	3015	6.44	4105	**0.36**	3955	1.12
8	3868	4.11	3582	5.54	3802	4.26	4616	**0.33**
9	10215	24.20	9545	27.96	14071	**0.35**	11087	17.22
10	3191	8.67	3579	5.79	4399	**0.34**	3146	8.79
11	26630	**0.37**	24522	12.67	22372	25.82	21578	31.02
12	13829	23.75	17725	**0.32**	13344	26.55	13743	24.02
13	16319	20.19	18104	8.48	19307	**0.36**	17379	12.96
14	28270	9.59	30213	**0.33**	26361	20.99	25974	23.16
15	20119	2.84	17183	20.62	20422	**0.41**	17192	20.68
av.	9189	7.68	9240	7.61	9339	6.54	8694	10.63

Table 4: Node Counts and Idle Times for 4 Processors

#	Proc. A Gen. Nodes	Proc. A Idle *ss.ss*	Proc. B Gen. Nodes	Proc. B Idle *ss.ss*	Proc. C Gen. Nodes	Proc. C Idle *ss.ss*	Proc. D Gen. Nodes	Proc. D Idle *ss.ss*	Proc. E Gen. Nodes	Proc. E Idle *ss.ss*	Proc. F Gen. Nodes	Proc. F Idle *ss.ss*	Proc. G Gen. Nodes	Proc. G Idle *ss.ss*
1	823	3.23	980	2.17	611	4.11	832	2.81	711	3.35	491	4.21	389	**1.52**
2	601	4.29	815	3.08	725	3.38	766	3.09	1112	**1.09**	611	3.41	384	2.79
3	1363	1.30	1355	1.29	1355	1.18	1355	1.21	1346	**1.09**	393	10.18	237	9.88
4	2892	5.28	2865	5.40	2832	5.23	2832	4.99	2846	4.91	3323	**0.96**	2026	3.72
5	1127	2.75	1103	2.64	929	3.52	1212	1.93	1080	2.21	1256	1.26	854	**0.00**
6	815	7.10	892	6.47	850	6.78	1086	5.01	525	8.39	535	8.19	1080	**0.00**
7	1807	10.19	1679	10.83	2084	8.34	3123	2.51	1506	11.17	1432	11.51	2245	**0.00**
8	2318	6.88	2080	7.93	1804	9.25	2640	4.58	1922	8.10	3408	**0.76**	1781	3.44
9	4989	29.84	6468	19.91	9574	**1.24**	7347	14.40	4832	29.55	5294	27.18	6613	1.56
10	2112	8.74	1944	9.54	2378	6.58	1439	12.83	2280	6.77	2758	3.36	2168	**0.02**
11	16460	6.51	13271	26.55	12323	31.89	17337	**0.80**	14389	18.68	12691	28.66	8655	29.00
12	8167	27.80	12482	**1.28**	7611	30.89	7402	31.53	9485	18.52	9168	20.55	6823	15.62
13	10157	26.68	12508	11.10	9327	31.77	8698	35.69	7122	45.13	13323	5.14	10053	**0.00**
14	13154	42.96	20948	**1.21**	19183	10.67	15733	28.52	17635	18.98	14904	33.44	9266	38.16
15	11613	8.52	10058	17.56	12602	**1.96**	12465	3.21	9544	20.63	10905	12.08	6835	18.99
av.	5226	12.80	5963	8.46	5612	10.45	5617	10.21	5089	13.24	5366	11.39	3960	8.31

Table 5: Node Counts and Idle Times for 7 Processors

With the composition of the idle time measure in mind, Table 4 demonstrates some aspects of synchronization overheads that agree well with independent predictions. The salient feature of the data is that, for each problem, at least one processor has an idle time close to zero (roughly between 30 and 40 milliseconds, with one exception). That is, at least one processor suffers only a negligible loss to overheads. If we assume that communication overhead is equally distributed across processors, it follows that communication overhead is negligible for all processors. This assumption seems intuitively rational, there being no apparent reason to suspect that the "busy" processor is biased toward a lesser degree of communication. Lastly, we note that the processor to finish last (identified in the data by emboldened idle times) appears to be randomly determined.

Table 5 exhibits an interesting anomaly in the recorded times of the last processor to finish. When Processor G is last to finish, it usually has an idle time of 0. Each of the other processors, when last to finish, shows a discrepancy of roughly 1 second from the range of minimum times found in Table 4. This inconsistency may have its source in the doubled-up processor configuration (noted in Section 3.2.1); other than the number of processors, the doubling of master and slave is the only difference we have identified between the 4- and 7-processor cases. The occurrence of this anomaly reinforces the need for bigger problems to experiment with: solution times should be large enough to overwhelm any small and constant overheads that cannot be identified.

3.2.2.4. Placement of Master Process

Table 6 explores the effect of doubling a master and a slave on one machine, versus retaining an independent master. Five of the computationally more expensive problems were chosen. The goal in this experiment was to determine the extent to which doubling decreased the effective power of the system. More generally, this experiment also tests the influence of introducing a non-standard processor (here an effectively slower one) to a homogeneous set. Note that in the 7-processor case a seventh standalone processor was made available and used for the *Master Alone* tests.

	Times (*sss.ss*)								
	2 Processors			4 Processors			7 Processors		
Prob. #	Master Doubled	Master Alone	Diff.	Master Doubled	Master Alone	Diff.	Master Doubled	Master Alone	Diff.
11	337.63	288.64	48.99	176.50	158.59	17.91	104.76	106.32	-1.56
12	214.12	188.60	25.52	121.78	109.39	12.39	74.94	75.00	-0.06
13	286.37	251.36	35.01	141.72	127.61	14.11	97.11	86.33	10.78
14	356.58	310.72	45.86	197.21	163.50	33.71	111.86	107.56	4.30
15	277.65	226.05	51.60	131.87	128.94	2.93	81.56	76.61	4.95
av.	294.47	253.07	41.40	153.82	137.61	16.21	94.05	90.36	3.68

Table 6: Timing Effects of Doubling Master and Slave

The data, which is for the last five problems of Zariffa's set, shows that for 2- and 4-processor systems an independent master increases performance by a non-trivial amount. However, with 7 processors an independent master causes slower solution times in some cases (Problems 11 and 12). For these instances the increase in processing power is small enough to be offset by random, detrimental changes in the order of work allocation. Our conclusion is that, as the degree of parallelism increases, the benefit from an independent master tends to zero.

An inference that may be drawn from this conclusion is that having one slightly *faster* processor will have just as little effect at parallelism 7 as having a slightly *weaker* one. This justifies the direct comparison between Zariffa's results and our own for 7 processors. On the other hand, the 4-processor results suggest that our solution times for 4 processors may overestimate the power of our system, since the influence of a fifth processor is non-trivial. The general conclusion drawn from the data is that the greater the parallelism, the less significant the effect of having a processor of different capability.

3.2.2.5. Estimating Synchronization Overhead

Table 7 compares methods for estimating synchronization loss in a 4-processor system. Values appearing in the *Overrun* column were calculated by first dividing sequential solution times by the number of processors to yield the *Linear Speedup* column. Linear speedups were subtracted from the *Measured* column to yield the *Overrun* column, an estimate of the overrun of the measured parallel solution times relative to linear (ideal) speedup. Times appearing in the *Average Idle* columns were arrived at using two different empirical estimators: clocking around polls from the slave to the master (this technique was described in Section 3.2.2.3), and counting the nodes searched per processor. In the latter method, a *cost per node* value is derived for a given search tree by dividing the largest individual processor node count into the measured parallel solution time. Individual processor working times are estimated by multiplying their respective node counts and the *cost per node*; idle time estimates follow after subtracting working times from measured solution times.

	Times (*sss.ss*)			Synch. Loss Estimators			
				Clocking		Node Counts	
Prob. #	Measured	Linear Speedup	Overrun	Average Idle	Clocking Error (%)	Average Idle	Node Error (%)
1	8.47	5.79	2.69	2.28	15.24	1.02	62.17
2	9.35	5.82	3.53	1.83	48.16	1.68	52.34
3	24.63	13.19	11.44	8.05	29.63	7.94	30.57
4	37.56	33.44	4.12	4.57	-10.68	4.32	-4.85
5	10.74	8.45	2.29	1.33	41.92	0.97	57.75
6	11.26	8.98	2.28	1.64	28.07	1.41	38.27
7	22.02	17.14	4.88	3.87	20.90	3.47	28.94
8	22.86	19.34	3.52	3.56	-1.14	3.20	9.09
9	90.75	64.63	26.13	17.43	33.26	18.15	30.54
10	29.78	21.59	8.19	5.90	27.96	5.51	32.72
11	158.59	135.19	23.40	17.47	25.34	17.05	27.15
12	109.39	81.75	27.64	18.66	32.49	18.87	31.73
13	127.61	114.20	13.41	10.50	21.70	9.89	26.25
14	163.50	142.40	21.10	13.52	35.97	13.49	36.08
15	128.94	106.74	22.20	11.14	49.82	10.96	50.63
mean (calculated down)				8.12	26.58	7.86	33.96

Table 7: Synchronization Overhead Calculations for 4 Processors

The overrun and the empirical estimators were analyzed with the following premises in mind:

(a) Time cost per node is randomly distributed about a mean, which entails an expected time cost for an arbitrary node. As a result, in a given amount of time one processor is unlikely to search (significantly) more nodes than another. Put another way, node counts are in fixed proportion to

working (non-idle) time.

(b) Communication overhead is negligible, so that the idle time measurements reflect primarily synchronization overheads.

(c) At least one processor is working for the duration of the solution time, so that the number of nodes searched by that processor can be used to find a *cost per node* value for that tree, without correcting for time spent idle.

(d) (a) and (c) together imply that the idle time of a processor can be estimated from the *cost per node* for the tree and the node count of that processor.

Note that (b) and (c) derive from the discussion of idle times in Section 3.2.2.3; thus we assume here that clocking around polls is a valid method of measuring synchronization losses. Now, comparing *Average Idle* columns, the two empirical estimators appear to correlate well, especially for larger problems. This correlation lends support to the node count method, based on premise (d), and hence supports premise (a) as well. But then the overrun calculation is also justified, since it assumes (a); it also assumes insignificant search overhead, which is confirmed by Table 2b.

Thus both the overrun calculation and the empirical estimators appear valid as measures of synchronization overhead. Unfortuneately, although the empirical estimators agree well with each other, when compared to the overrun there is a non-trival discrepancy. The differences between clocking estimator and overrun and between node count estimator and overrun are presented as percentages of the overrun in Table 7. We note that the percentages cluster about the mean; that is, the raw differences appear to vary in proportion with the overrun. Several hypotheses, dealing primarily with overheads not accounted for in the empirical estimators, have been suggested to explain the difference, but so far none have been found tenable. For instance, start-up overheads, since they are constant, are probably not responsible for a discrepancy that varies between problems. Also, as mentioned above, search overhead is insignificant, and therefore not responsible.

3.2.2.6. Variance in Solution Times

Table 8 addresses the issue of variance in solution times. To induce variance, a period of daytime Ethernet traffic was chosen, and ten repetitions of Problem #14 were executed with the UNIX-based processor lightly loaded with various non-cpu-intensive processes. There is a slight variation in node counts between runs, confirming the evidence of the 7-processor data in Table 6, namely that parallel search can be affected by small changes in the timing of inter-processor communication. However, the differences are not significant, and the variance in solution times is negligible.

Run #	4 Processors Nodes	4 Processors Time	7 Processors Nodes	7 Processors Time
1	110818	180.36	110844	120.20
2	110818	180.36	110860	119.90
3	110818	180.38	110860	120.22
4	110818	180.34	110818	120.06
5	110818	180.32	110810	120.04
6	110818	180.36	110841	120.02
7	110818	180.36	110849	120.16
8	110818	180.36	110769	119.96
9	110818	180.52	110855	120.24
10	110792	180.18	110789	120.24

Table 8: Repeated Runs of Problem #14
Under Daytime Ethernet Traffic

4. Summary and Conclusions

The purpose of the experiments and analysis reported here is to provide a greater understanding of synchronization overheads in loosely coupled parallel search. Results from a study by Zariffa ([11]), which examined parallel solutions to the vertex cover problem, were replicated and compared. In almost every respect we successfully deduced and implemented the algorithms and replicated the given solutions.

In our experiments we found not only dramatically reduced solution times with our hardware and software, but also discovered a simple-to-implement improvement to the basic search algorithm, an improvement that we termed *duplicate path elimination*. Because *dpe* significantly reduces uniprocessor search times, speedups reported in [11] were not relative to the fastest available sequential algorithm, and as such may not accurately reflect the effective power of parallel solutions to the vertex cover problem. However, realizing an efficient parallel solution for use with *dpe* is made difficult by the highly skewed shape of *dpe* search trees.

In our attempts to quantify synchronization losses in our systems, we used two empirical estimators and compared them to solution time overrun. The empirical measures correlated well with each other, but consistently failed to account for all of the overrun; this discrepancy is a matter for further research.

Two factors pointed to a need to experiment with larger problem graphs than those presented by Zariffa: the occasional occurrence of non-negligible idle times for the last processor to finish, and the increased speed of solutions when *dpe* is in place. Ideally, solution times should be great enough to overwhelm all small, unidentifiable overheads.

One benefit of this study is that our parallel solution for the vertex cover problem exhibits synchronization losses comparable to those found in chess programs [5, 6]. Thus, in terms of overheads, we have an simple analogy for the chess case, so any technique that reduces the synchronization overhead in one will probably be effective in the other.

References

1. T. Breitkreutz, T.A. Marsland and E. Altmann, Parallel Search of Skewed Trees, TR87.16, Comp. Sci. Dept., Univ. of Alberta, Edmonton, In preparation.
2. Ten-Hwang Lai and Sartaj Sahni, Anomalies of Parallel Branch-and-Bound Algorithms, *Communications of the ACM 27(6)*, (June 1984), 594-602.
3. Eugene L. Lawler, Covering Problems: Duality Relations and a New Method of Solution, *SIAM Journal on Applied Mathematics 14(5)*, (Sept 1966), 1115-1132.
4. Guo-jie Li and Benjamin W. Wah, Coping with Anomalies in Parallel Branch-and-Bound Algorithms, *IEEE Trans. on Computers 35(6)*, (June 1986), 568-573.
5. T.A. Marsland and F. Popowich, Parallel Game-Tree Search, *IEEE Trans. on PAMI 7(4)*, (July 1985), 442-452.
6. T.A. Marsland, M. Olafsson and J. Schaeffer, Multiprocessor Tree-Searching Experiments, in *Advances in Computer Chess 4*, D. Beal (ed.), Pergamon Press, 1985, 37-51.
7. J. Mohan, A Study of Parallel Computation - The Travelling Salesman Problem, Tech. Rep. CMU-CS-82-136, Comp. Sci. Dept., Carnegie Mellon Univ., Pittsburg, Aug. 1982.
8. M. Newborn, *Private Communication*, McGill University, Montreal, July 1986.
9. Benjamin W. Wah and Y. W. Eva Ma, MANIP -- A Multicomputer Architecture for Solving Combinatorial Extremum-Search Problems, *IEEE Trans. on Computers C-33(5)*, (May 1984), 377-390.
10. Benjamin W. Wah, Guo-jie Li and Chi Fen Yu, Multiprocessing of Combinatorial Search Problems, *IEEE Computer*, June 1985, 93-108.
11. N. Zariffa, Implementation and Analysis of Three Parallel Branch-and-Bound Algorithms for the Vertex Covering Problem, M.Sc. Thesis, School of Computer Science, McGill University, Montreal, March 1986.

Appendix A

The following pseudocode gives, in C-like syntax, the lower bound vertex cover algorithm used to replicate Zariffa's results. The algorithm is executed by the slave processors in parallel search. Function `vcover` controls the depth-first search, and `findbounds` calculates lower bounds.

```
vcover( vertices, selected, nextvertex, depth, bound )
{
    nodes++;            /* generated node count */

    /* add nextvertex (locally) to those already selected
     */
    select( selected, nextvertex );
    if ( SolutionFound() )
        return( depth );

    /* find lower bounds of vertices, given those selected:
     */
    findbounds( bounds, vertices, selected );

    /* stable sort on bounds of vertices, done on local copy:
     */
    sort( bounds, vertices );

    /* set sons to be the set of unselected vertices:
     */
    sons = subtract( vertices, selected );

    /* do a depth-first search of sons:
     */
    for (i = 0; i < size(sons); i++) {
        if ( NewBoundArrived() )   /* non-blocking poll */
            bound = GetNewBound();

        /* if current bound causes a cutoff, skip
         * recursive call but count generated node:
         */
        if ( cutoff(bound, bounds[i]) ) {
            nodes++;
            continue;
        }
        bound = vcover( vertices, selected,
                sons[i], depth+1, bound );
    }
    return( bound );
}
```

```
findbounds( bounds, vertices, selected )
{
    sons = subtract( vertices, selected );

    /* find bounds for each son in turn:
     */
    for (i = 0; i < size(sons); i++) {

        /* select ith vertex (locally), then find
         * updated outdegrees of unselected vertices:
         */
        select( selected, i );
        findoutdegree( vertices, selected );

        /* count vertices until the sum of their
         * outdegrees is ≥ the number of edges
         * remaining to be covered:
         */
        sum = 0; /* sum of covered edges */
        for (j = 0; ; j++) {
            if (sum ≥ remaining)
                break;

            /* find and include vertex with highest
             * outdegree (max):
             */
            max.outdegree = 0;
            for (k = 0; k < GraphSize; k++)
                if (sons[k].outdegree > max.outdegree)
                    max = remember( k, sons );
            sum = sum + max.outdegree;

            /* max is not considered again in finding
             * the lower bound for ith vertex:
             */
            sons[max.index].outdegree = 0;
        }
        sons[i].bound = j;   /* lower bound for vertex i */

        /* reset i to be an unselected vertex,
         * for next iteration:
         */
        deselect( selected, i );
    }
}
```