International Journal of Foundations of Computer Science

DETERMINISTIC BRANCH-AND-BOUND ON DISTRIBUTED MEMORY MACHINES

KIERAN T. HERLEY
Department of Computer Science, University College Cork, Cork, Ireland.
k.herley@cs.ucc.ie.

and

ANDREA PIETRACAPRINA and GEPPINO PUCCI
Dipartimento di Elettronica e Informatica, Università di Padova, I-35131 Padova, Italy.
{andrea,geppo}@artemide.dei.unipd.it

Received
Revised
Communicated by

ABSTRACT

The *branch-and-bound* problem involves determining the leaf of minimum cost in a cost-labelled, heap-ordered tree, subject to the constraint that only the root is known initially and that the children of a node are revealed only by visiting their parent. We present the first efficient deterministic algorithm to solve the branch-and-bound problem for a tree T of constant degree on a p-processor distributed-memory Optically Connected Parallel Computer (OCPC). Let c^* be the cost of the minimum-cost leaf in T, and let n and h be the number of nodes and the height, respectively, of the subtree $T^* \subseteq T$ of nodes whose cost is at most c^*. When accounting for both computation and communication costs, our algorithm runs in time $O\left(n/p + h(\max\{p, \log n \log p\})^2\right)$ for general values of n, and can be made to run in time $O\left(\left(n/p + h\log^4 p\right)\log\log p\right)$ for n polynomial in p. For large ranges of the relevant parameters, our algorithm is provably optimal and improves asymptotically upon the well-known randomized strategy by Karp and Zhang.

Keywords: parallel branch-and-bound; distributed-memory machines; OCPC; load balancing; information dispersal.

1. Introduction

Branch-and-bound is a widely used and effective technique for solving hard optimization problems. It determines the optimum-cost solution of a problem through a selective exploration of a solution tree, whose internal nodes correspond to different relaxations of the problem and whose leaves correspond to feasible solutions. The shape of the tree is generally not known in advance, since the subproblems associated with the nodes are generated dynamically in an irregular and unpredictable fashion. A suitable abstract framework for studying the balancing and communication

issues involved in the parallel implementation of branch-and-bound is provided by the *branch-and-bound problem*, introduced in [8], which can be specified as follows. Let T be an arbitrary tree of finite size. Initially, a pointer to the root is available, while pointers to children are revealed only after their parent is *visited*. A node can be visited only if a pointer to it is available, and it is assumed that the visit takes constant time. All nodes of T are labeled with distinct integer-valued *costs*, the cost of each node being strictly less than the cost of its children (heap property). The branch-and-bound problem involves determining the cost c^* of the minimum-cost leaf. Note that any correct algorithm for the branch-and-bound problem must visit all those nodes whose costs are less than or equal to c^*. These nodes form a subtree T^* of T. Throughout the paper, n and h will denote, respectively, the size and the height of T^*.

The efficiency of any parallel branch-and-bound algorithm crucially relies on a balanced on-line redistribution of the computational load (tree-node visits) among the processors. Clearly, the cost of balancing must not be much larger than the cost of the tree-visiting performed. Furthermore, since c^* is not known in advance, one cannot immediately distinguish nodes in T^* (all of which must be visited) from nodes in $T - T^*$ (whose visits represent wasted work). Ensuring that the algorithm visits few superfluous nodes is nontrivial in a parallel setting as it requires considerable coordination between processors.

In this paper, we devise an efficient deterministic parallel algorithm for the branch-and-bound problem on a *Distributed-Memory Machine* (DMM) consisting of a collection of processor/memory pairs communicating through a complete network. The model assumes that in one time step, each processor can perform $O(1)$ operations on locally stored data and send/receive one message to/from an arbitrary processor. We consider the weakest DMM variant, also known as *Optically Connected Parallel Computer* (OCPC) in the literature [3], where concurrent transmissions to the same processor are heavily penalized. Specifically, in the event that two or more processors simultaneously attempt to transmit to the same destination, the processors involved are informed of the collision but no message is received by the destination.

1.1. Related Work and New Results

A simple sequential algorithm for the branch-and-bound problem is based on the *best-first* strategy, where available (but not yet visited) nodes are stored in a priority queue and visited in increasing order of their cost. The $O(n \log n)$ running time of this simple strategy is dominated by the cost of the $O(n)$ queue operations. In [2], Frederickson devised a clever sequential algorithm to select the k-th smallest item in an infinite heap in $O(k)$ time. The algorithm can be easily adapted to yield an optimal $O(n)$ sequential algorithm for branch-and-bound.

In parallel computation, the branch-and-bound problem has been studied on a variety of machine models. In [8], Karp and Zhang show, by a simple work/diameter argument, that any algorithm for the problem requires at least $\Omega(n/p + h)$ time on any p-processor machine, and devise a general randomized algorithm, running

in $O(n/p + h)$ steps, with high probability. Each step of the algorithm entails a constant number of operations on local priority queues per processor, and the routing of a global communication pattern where a processor can be the recipient of $\Theta(\log p/\log\log p)$ messages. A straightforward implementation of this algorithm on any DMM would require $\Omega(\log(n/p) + \log p/\log\log p)$ time per step, with high probability, if both priority queue and contention costs are to be fully accounted for. The resulting algorithm is nonoptimal for all values of the parameters n, h and p.

Kaklamanis and Persiano [7] present a deterministic branch-and-bound algorithm that runs in $O\left(\sqrt{nh}\log n\right)$ time on an n-node mesh. The mesh-specific techniques employed by the algorithm, coupled with their assumption that the mesh size and problem size are comparable, limit the applicability of their scheme to other parallel architectures.

A deterministic algorithm for the shared-memory EREW PRAM, based on a parallelization of the heap-selection algorithm of [2] appears in [6]. The main result of this paper extends the approach of [6] to the distributed-memory OCPC with the performance stated in the following theorem.

Theorem 1 *There is a deterministic algorithm running on a p-processor OCPC that solves the branch-and-bound problem for any tree T of constant-degree in time*

$$O\left(\frac{n}{p} + h(\max\{p, \log n \log p\})^2\right).$$

When n is polynomial in p, the algorithm can be also implemented to run in time

$$O\left(\left(\frac{n}{p} + h\log^4 p\right)\log\log p\right).$$

Note that when $n/\max\{p^3, p(\log n \log p)^2\}$ is large with respect to h, a typical scenario in real applications, our algorithm achieves optimal $\Theta(n/p)$ running time. In this case, the implementation of the algorithm is very simple and the big-oh in the running time does not hide any large constant. In contrast, the second implementation is asymptotically better, although more complex, when the values of n/p and h are close, and it is indeed within a mere $O(\log\log p)$ factor of optimal as long as $h = O\left(n/(p\log^4 p)\right)$, which is a weak balance requirement on the solution tree.

The rest of the paper is organized as follows. Section 2 reviews the generic branch-and-bound strategy of [6], while Section 3 describes the two OCPC implementations of the generic algorithm, whose running times are given in Theorem 1. Finally, Section 4 provides some concluding remarks and directions for future research.

2. A Machine-Independent Algorithm

In this section, we briefly review the machine independent, parallel branch-and-bound strategy introduced in [6], which is at the base of the OCPC algorithms described in this paper.

Consider a branch-and-bound tree T and let s be an integer parameter which will be specified later. For simplicity, we assume that T is binary, although all our results immediately extend to trees of any constant degree. We begin by summarizing some terminology introduced in [2, 6]. For a set of tree nodes N, let $Best(N)$ denote the set containing the (at most) s nodes with smallest cost among those in N and their descendants. A set of nodes of the form $Best(N)$ is called a *clan*, and nodes themselves are referred to as the clan's *members*. The nodes of T can be organized in a binary *tree of clans* $\mathcal{TC}$ as follows. Let r denote the root of T. The root of $\mathcal{TC}$ is the clan $R = Best(\{r\})$. Let C be a clan of $\mathcal{TC}$, and suppose that $C = Best(N)$ for some set N of nodes of T. Define $Off(C)$ (*offspring*) as the set of tree nodes which are children of members of C but are not themselves members of C, and define $PR(C)$ (*poor relations*) to be the set $N - Best(N)$. Then, clan C has two (possibly empty) child clans C' and C'' in $\mathcal{TC}$, namely $C' = Best(Off(C))$ and $C'' = Best(PR(C))$. Since T is binary, we have $|Off(C)|, |PR(C)| \leq 2s$, for every clan $C \in \mathcal{TC}$. If a clan C has exactly s members, its *cost*, denoted by $cost(C)$, is defined as the maximum cost of any of its members; if C has less than s members, then we set $cost(C)$ to be ∞ (note that in this last case, C is a leaf of $\mathcal{TC}$). Since every node in a clan C costs less than every node in $Off(C) \cup PR(C)$, it is easy to show that both $cost(C')$ and $cost(C'')$ are strictly greater than $cost(C)$. As a consequence, $\mathcal{TC}$ is heap-ordered with respect to clan cost. An example of clan creation is illustrated in Fig. 1.

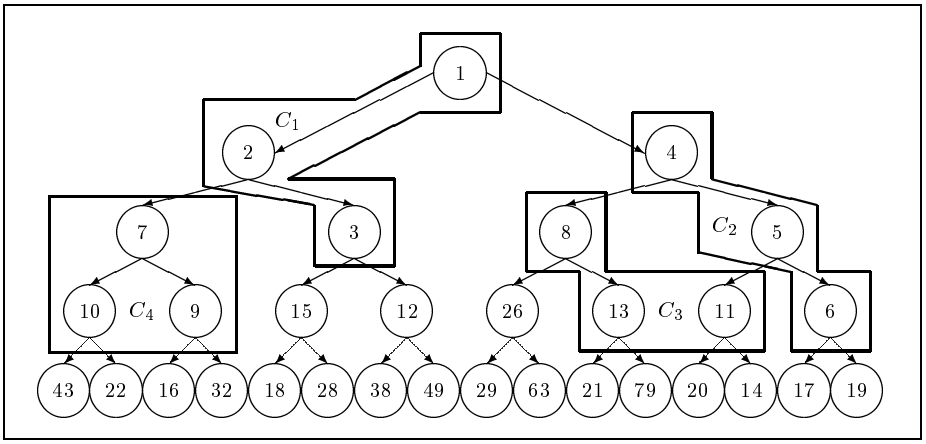

Fig. 1. The clan creation process ($s = 3$). In the picture, $C_1 = Best(\{1\})$ has $Off(C_1) = \{4, 7, 12, 15\}$, $PR(C_1) = \emptyset$, and $cost(C_1) = 3$. Clan $C_2 = Best(Off(C_1))$ has $Off(C_2) = \{8, 11, 17, 19\}$, $PR(C_2) = \{7, 12, 15\}$, and $cost(C_2) = 6$. Finally, $C_3 = Best(Off(C_2))$, with $cost(C_3) = 13$, and $C_4 = Best(PR(C_2))$, with $cost(C_4) = 10$. In $\mathcal{TC}$, C_1 is the root, and C_2 is its sole child. C_3 and C_4 are the children of C_2.

In [2], Frederickson shows that the k-th smallest node of T is a member of one of the $2\lceil k/s \rceil$ clans of minimum cost. Based on this property, he develops a sequential algorithm that finds the k-th smallest node in T in linear time by performing a clever exploration of $\mathcal{TC}$ in increasing order of clan cost. Note that once such a

node is found, the k nodes of smallest cost in T can be enumerated in linear time as well. By repeatedly applying this strategy for exponentially increasing values of k until the smallest-cost leaf is found, the branch-and-bound problem can be solved for T in time proportional to the size of T^*.

The parallel branch-and-bound strategy proposed in [6] can be seen as a *best-first* parallel exploration of $\mathcal{TC}$, where each clan is created in $O(s)$ time by a single processor using a straightforward adaptation of the heap-selection algorithm of [2]. Such a strategy is realized by the generic algorithm BB of Fig. 2, which applies to any p-processor machine. In the next section, we will show how to implement algorithm BB efficiently on the OCPC.

Let P_i denote the i-th processor of the machine, for $1 \leq i \leq p$. P_i maintains a local variable ℓ_i, which is initialized to ∞. Throughout the algorithm, variable ℓ_i stores the cost of the cheapest leaf visited by P_i so far. Also, a global variable ℓ is maintained, which stores the minimum of the ℓ_i's. At the core of the algorithm is a *Parallel Priority Queue* (PPQ), a parallel data structure containing items labeled with an integer-valued key [11]. Two main operations are provided by a PPQ: *Insert*, that adds a p-tuple of new items into the queue; and *Deletemin*, that extracts the p items with the smallest keys from the queue. The branch-and-bound algorithm employs a PPQ Q to store clans, using their costs as keys. Together with Q, a global variable q is maintained, denoting the minimum key currently in Q. Initially, Q is empty and a pointer to the root r of T is available.

Algorithm BB:

1. P_1 produces clan $R = Best(\{r\})$ and sets ℓ_1 to the cost of the minimum leaf in R, if any exists. Then, R is inserted into Q, and q and ℓ are set to the cost of R and to ℓ_1, respectively.
2. The following substeps are iterated until $\ell < q$.
   (a) *Deletemin* is invoked to extract the $k = \min\{p, |Q|\}$ clans $C_1, C_2, \ldots, C_k$ of smallest cost from Q. For $1 \leq i \leq k$, clan C_i is assigned to P_i.
   (b) For $1 \leq i \leq k$, P_i produces the two children of C_i, namely C_i' and C_i'', and updates ℓ_i accordingly.
   (c) *Insert* is invoked (at most twice) to store the newly produced clans into Q. The values ℓ and q are then updated accordingly.
3. The value ℓ is returned.

Fig. 2. The generic parallel branch-and-bound algorithm.

The following lemma was proved in [6].

Lemma 1 *Algorithm BB is correct. Moreover, the number of iterations of Step 2 required to reach the termination condition is* $O(n/(ps) + hs)$.

3. OCPC Implementation

In this section, we show how algorithm BB can be efficiently implemented on the OCPC. The implementation crucially relies on the availability of fast PPQ operations, which is guaranteed by the following lemma.

Lemma 2 *A PPQ Q storing items of constant size can be implemented on a p-processor OCPC so that both Insert and Deletemin take $O(\log(|Q|/p)\log p)$ time.*

Proof. We make use of the *p-Bandwidth Heap* (p-BH) of [11], a binary heap with large nodes, each maintaining p items sorted by their keys. On the OCPC, we distribute the p items held by each p-BH node among the p memory modules. Based on this allocation, the PRAM algorithms for Insert and Deletemin described in [11] can be ported immediately to the OCPC by replacing PRAM sorting and merging with bitonic sorting and merging on the OCPC [9], yielding the stated time bound. □

If we adopted the naive approach of viewing each whole clan (with its $\Theta(s)$ members, offspring and poor relations) as a PPQ item, the complexity of the above operations would increase by a factor of $\Theta(s)$ – a time penalty which is too large for our purposes – since each elementary step would entail the actual migration of the clans involved among the processors. To overcome this problem, we store each clan in a distinct *cell* of a *virtual shared memory* and represent the clan in the PPQ using a constant-size record that includes its cost and its virtual cell's address. Virtual cells are suitably mapped onto the processors' memories, in such a way that the contents of p arbitrary cells can be efficiently retrieved. The mapping must guard against the possibility that the p cells to be accessed might be concentrated in a small number of memory modules, which would render the access unacceptably expensive due to memory contention. Based on these ideas, we propose two implementations, distinguished by the choice for s, and whose efficiency depends on the relative values of n/p and h.

3.1. Implementation 1

Let $s = a\max\{p, \log n \log p\}$, for some constant $a > 0$, and suppose that the $\Theta(s)$ data stored in any virtual cell (i.e., members, offspring and poor relations of some clan) are evenly distributed among the p memory modules, with $O(s/p)$ data per module. Consider an iteration of Step 2 of algorithm BB. Since each clan is represented in the PPQ by a constant-size record, the Deletemin and Insert operations required in Substeps 2.a and 2.c take $O(\log n \log p)$ time by Lemma 2 (note that Lemma 1 implies that at any time during the algorithm $|Q| = O(nsp)$). The generation of the child clans in Substep 2.b can be performed locally at each processor using Frederickson's algorithm in $O(s)$ time. Also, all data movements involved in Step 2 can be easily arranged as a set of $O(s)$ fixed permutations that take $O(s)$ time. Thus, each iteration of Step 2 requires time $O(s)$, which yields an $O(n/p + hs^2)$ running time for the entire algorithm. The first part of Theorem 1 follows by plugging in the chosen value for s.

It has to be remarked that our choice of s requires that the algorithm know

the value $n = |T^*|$ in advance, which is clearly an unrealistic assumption in most real scenarios. In order to remove such an assumption while maintaining the same running time (up to constant factors), we may proceed as follows. Let $n_1 = p^2$, and, for $i > 1$ let $n_i = (n_{i-1})^2$. We start running the algorithm guessing the value n_1 for $|T^*|$. If the algorithm does not terminate within time n_1/p, we abort the execution and run the algorithm again, guessing $|T^*| = n_2 = (n_1)^2$. In general, on the i-th iteration, we run the algorithm, guessing $|T^*| = n_i$, and abort execution if it does not terminate within time n_i/p. Let $\hat{\imath}$ be the least positive index such that $n_{i-1} < n < n_i$. Then, it is easy to see that, depending on the actual values of n and h, the iterated algorithm will terminate either at guess $\hat{\imath}$, or $\hat{\imath} + 1$, or $\hat{\imath} + 2$. Since the guessed values for n increase geometrically, the running time of the algorithm is determined by the running time of the last iteration, which is $O\left(n/p + h \max^2\{p, \log n \log p\}\right)$, since $\log n_{i+j} = \Theta(\log n)$, for $0 \leq j \leq 2$.

3.2. Implementation 2

Note that Implementation 1 achieves optimal $\Theta(n/p)$ running time when n/p is rather larger than h; however, it becomes progressively less profitable for more unbalanced trees. In the latter scenario, it is convenient to choose a much smaller value for s which, however, requires a more sophisticated mechanism to avoid memory contention. In particular, when $s \ll p$, it becomes necessary to introduce some redundancy in the representation of virtual cells and to carefully select, for each virtual cell, a subset of processors that store its (replicated) contents, so that any p cells can be retrieved by the processors with low contention at the memory modules. These ideas are explained in greater detail below.

In the following, we assume that n is polynomial in p, that the machine word contains $\Theta(\log p)$ bits, and that each node of the branch-and-bound tree T is represented using a constant number of words. In this fashion, clans, as well as virtual cells, can be regarded as strings of $\Theta(s \log p)$ bits. Each cell is assigned a set of $d = \Theta(\log p)$ distinct memory modules as specified by a suitable *memory map* modeled by means of a bipartite graph $G = (U, V, E)$ with $|U|$ inputs, corresponding to the virtual cells, $|V| = p$ outputs, corresponding to the OCPC memory modules, and d edges connecting each virtual cell to the d modules assigned to it. The quantity $|U| = p^{O(1)}$ is chosen as an upper bound to the total number of virtual cells ever needed by the algorithm. We call d the *degree* of the memory map.

Consider a set of p newly created clans, $C_1, C_2, \ldots, C_p$, to be stored in the memory modules, and let processor P_i be in charge of clan C_i. First, a distinct unused cell u_i is chosen for each clan C_i. Then, P_i recodes u_i into a longer string of size $3|u_i|$ and splits it into $k = \epsilon \log p$ *components* ($\epsilon < 1$), each of $3|u_i|/k = \Theta(s)$ bits (stored in $\Theta(s/k)$ words), by using an Information Dispersal Algorithm (IDA) [12, 13], so that any $k/3$ components suffice to recreate the original contents of u_i. The following lemma, proved in Subsection 3.2.1, establishes the complexity of these encoding/decoding operations.

Lemma 3 *A processor can transform a cell u into k components of $\Theta(s)$ bits each, in $O(s \log k) = O(s \log \log p)$ time, so that u can be recreated from any $k/3$ compo-*

nents within the same time bound.

After the encoding, P_i replicates each component of u_i into $a = d/k = O(1)$ copies, referred to as *component copies*, and attempts to store the resulting d component copies of u_i into the d modules assigned to the cell by the memory map G, in parallel for every $1 \leq i \leq p$. The operation terminates as soon as all processors effectively store at least $2d/3$ component copies each.

Consider now the case when the processors need to fetch the contents of p cells from the memory modules. Each processor attempts to access the d modules that potentially store the component copies of the cell and stops as soon as *any* $2d/3$ modules are accessed. Although not all accessed modules may effectively contain component copies of the cell, we are guaranteed that at least $d/3$ component copies, hence $k/3$ distinct components, will be retrieved. This is sufficient for each processor to reconstruct the entire cell. The following lemma will be proved in Subsection 3.2.2.

Lemma 4 *For a suitable constant $a > 1$, there exists a memory map $G = (U, V, E)$ with $|U| = p^{O(1)}$, $|V| = p$ and with each node in U of degree $d = a\epsilon \log p$, under which any set S of p virtual cells can be stored/retrieved in the OCPC memory modules in $O\left(s + \log^2 p \log\log p\right)$ time.*

Putting it all together, the extraction of the p clans of minimum cost among those represented in the PPQ Q can be accomplished as follows. First, the addresses of the corresponding cells are extracted from Q in time $O\left(\log(|Q|/p) \log p\right) = O\left(\log^2 p\right)$ and distributed one per processor. Then, by Lemma 4, $k/3$ components for each cell are retrieved in $O\left(s + \log^2 p \log\log p\right)$ time. Finally, the contents of each clan C are reconstructed in $O\left(|C| \log k\right) = O\left(s \log\log p\right)$ time, by Lemma 3. By combining all of the above contributions we obtain a running time of $O\left(s \log\log p + \log^2 p \log\log p\right)$ for parallel clan extraction. In a similar fashion, one can show that the insertion of p new clans in the queue can be accomplished within the same time bound.

The complexity of algorithm BB is dominated by that of Step 2. Each iteration of this step entails one extraction and at most two insertions of p clans ($O\left(s \log\log p + \log^2 p \log\log p\right)$ time), the generation of at most two new clans per processor ($O(s)$ time), and a number of other simple operations all executable in $O(\log p)$ time. Hence, an iteration requires $O\left(s \log\log p + \log^2 p \log\log p\right)$ time. Since, by Lemma 1, there are $O(n/(ps) + hs)$ iterations, the second part of Theorem 1 follows by choosing $s = \log^2 p$. Note that, unlike Implementation 1, this choice of s does not require advance knowledge of the shape of T^*, hence no guessing is needed.

3.2.1. Proof of Lemma 3

We will only describe the encoding procedure, since the reconstruction procedure is based on the same ideas and exhibits the same running time. Consider a virtual cell u, and view it as a rectangular $\Theta(s) \times (k/3)$ bit-array A_u, with every row stored in a separate word. By using IDA [12, 13], each row can be independently recoded into a string of k bits, so that any $k/3$ such bits are sufficient to reconstruct the

entire row. The resulting $\Theta(s) \times k$ bit-array A'_u is then "repackaged" so that each of its k $\Theta(s)$-bit columns can be stored as a sequence of $\Theta(s/k)$ words. Each of these $\Theta(s/k)$-word sequences constitutes a distinct component. A pictorial representation of the encoding procedure is given in Fig. 3.

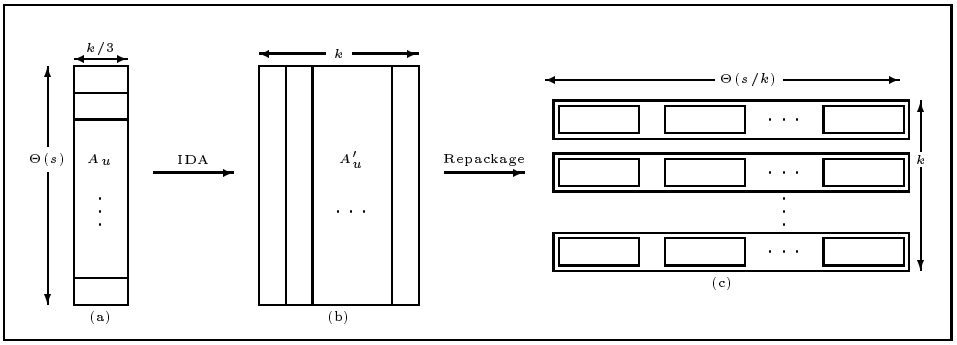

Fig. 3. Encoding of a clan into components: (a) clan contents are arranged into $\Theta(s)$ strings of $k/3$ bits each; (b) each row is encoded into a k-bit string through IDA; (c) each column is packaged into $\Theta(s/k)$ words, thus making a component.

Rather than actually running the information dispersal algorithm, we can use a precomputed look-up table of $2^{k/3} = p^{\epsilon/3}$ entries accessible in constant time. The i-th entry of the table, holds the k-bit encoding of the $(k/3)$-bit binary string corresponding to integer i. Note that this table need only be computed once and made available to each node of the machine. (A similar table with $2^k = p^\epsilon$ entries will be needed for decoding.)

The repackaging of the columns of A'_u into sequences of words can be achieved by sequentially transposing each of the $\Theta(s/k)$ $k \times k$-blocks of A'_u (k consecutive rows). Specifically, the transposition of a $k \times k$ block is performed by first swapping its northeast and southwest quadrants and then recursively and independently transposing the four $k/2 \times k/2$ quadrants in place (see Fig. 4 below).

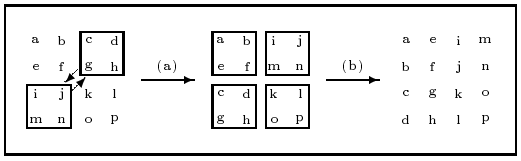

Fig. 4. Transposing a 4×4 block: (a) NE and SW quadrants are swapped; (b) each quadrant is independently and recursively transposed.

Note that the swaps implied by the above algorithm can be realized through a sequence of fixed bit-wise logical operations using masks. Since each of the $\log k$ levels of recursion requires k such operations and a constant number of masks, a block can be transposed in $O(k \log k)$ time using $O(\log k)$ masks.

The running time of the encoding procedure is dominated by the repackaging cost, which is $O(s \log k) = O(s \log \log p)$. Note that the encoding/decoding activities are performed locally at each processor and require no external communication.

3.2.2. Proof of Lemma 4

Consider a memory map $G = (U, V, E)$ and a set $S \subseteq U$ of p cells, and let $E(S)$ denote the set of edges incident on nodes of S. A *c-bundle* for S of congestion q is a subset $B \subseteq E(S)$ where each $u \in S$ has degree c and any $v \in V$ has degree at most q with respect to the edges in B. The following claim demonstrates that there exists a suitable G which guarantees that a $(2d/3)$-bundle of low congestion exists for every set S of p (or fewer) cells, and that such a bundle may be determined efficiently.

Claim 1 *For $d = a\epsilon \log p$, for a sufficiently large positive constant a, there exists a memory map G of degree d such that, for any given set S of p cells, there is a $(2d/3)$-bundle of congestion $q = O(d)$. Moreover, one such bundle can be determined in $O(d \log p) = O(\log^2 p)$ time on the OCPC.*

Proof. Choose $d = a\epsilon \log p$ to be an odd multiple of 3. We map the cell components to the memory modules according to a $(\lambda, d, d/3, 1/d)$-*generalized expander*, a bipartite graph $G = (V, U, E)$ whose existence for $|U| = p^{O(1)}$, $|V| = p$, and suitable constants $a > 1$ and $\lambda < 1$ was proved in [4, Theorem 4]. Such a map has the following expansion property: For any set S' of at most p/d cells in U, and any k-bundle B' for S', with $k \geq d/3$, the set $\Gamma_{B'}(S')$ of memory modules containing those components of cells in S' corresponding to edges in B' has size $|\Gamma(S')| \geq \lambda k |S'|$.

Based on this property, we can determine a low-congestion $2d/3$-bundle for any given set S of p cells by applying a simple variant of the *whittling protocol* of [5], which can be shortly described as follows. Starting from $E(S)$, each step of the protocol prunes the set of edges by selecting $2d/3$ edges each for some of the cells in S, and discarding the remaining $d/3$. The prunings are performed in such a way that the congestion of the final $2d/3$ bundle for S does not exceed a given congestion threshold $q = O(d)$. More specifically, at the beginning of each step, a cell in S is said to be *alive* if the $2d/3$ edges for the cell have not been selected yet, and *dead* otherwise. Let S_i be the set of live variables at the beginning of the i-th step, for $i \geq 1$. During the step, the processors first identify the set $V_i \subseteq V$ of "heavily loaded" modules, i.e., those modules storing more than q components of cells in S_i, and then select arbitrary sets of $2d/3$ edges incident on modules in $V - V_i$ for all those cells which have at least that many components stored outside V_i. Any such cell is marked dead, and its $d/3$ unselected components are discarded.

From a minor variation of [5, Lemma 8] it follows that $|S_i| \leq p/d^{i-1}$, for $i \geq 1$. As a consequence, $1 + \log_d p = O(\log p / \log \log p)$ steps suffice to determine a $2d/3$-bundle for S, which, by construction, has congestion at most $q = O(d)$.

On the OCPC, the i-th step of the whittling protocol, for $i \geq 1$, can be implemented using a constant number of sorting and prefix computations on a set of $d|S_i| \leq p/d^{i-2}$ items. As a consequence, the overall running time of the protocol is dominated by the time of the first step, which is $O(d \log p) = O(\log^2 p)$. □

Let us now consider the problem of writing a set S of p cells, $u_1, u_2, \ldots, u_p$ (the problem of reading p cells is similar). First, each processor P_i encodes the cell u_i it wishes to write into d component copies using the IDA-based techniques described before, and then all processors cooperate to construct a $(2d/3)$-bundle

B of congestion $O(d)$ for S using the whittling protocol described in the proof of Claim 1. At the end of the protocol, we assume that P_i knows the edges in B that are incident on cell u_i and on the i-th memory module v_i. It is important to note that the whittling protocol merely indicates to each processor P_i the locations to which the $2d/3$ component copies of its cell u_i should be written: the actual physical movement of the component copies of u_i, each consisting of $\Theta(s/k)$ words, must be implemented as a separate step. This is a nontrivial task, since each processor P_i must dispatch (resp., receive) one component copy for each edge in B incident on u_i (resp., v_i); hence, the processor must dispatch $2d/3$ component copies and receive $O(d)$ of them. In particular, the transmission of copies must be coordinated so as to avoid "collisions" at the receiving processors.

Standard routing techniques based on sorting [9] cannot be applied in our context, since they would yield an $\Omega(ds \log p/k) = \Omega(s \log p)$ component routing time that is too slow for our purposes. Hence, we resort to a more sophisticated approach based on edge coloring. More specifically, let $\Delta = \Theta(d)$ denote the maximum degree of a node in $U \cup V$ with respect to the edges in B. Then, any $O(\Delta)$-edge coloring of B would allow us to decompose component routing into $O(\Delta)$ stages, each requiring $\Theta(s/k)$ time, for an overall $O(\Delta s/k) = O(ds/k) = O(s)$ time. It remains to show that such an edge coloring for B can be computed efficiently by the OCPC processors. We establish the following result.

Claim 2 *A $(2\Delta - 1)$-edge coloring for B may be computed in $O(\Delta^2 \log \Delta)$ time on the OCPC.*

Proof. It is easy to extend the whittling protocol of Claim 1, so that, together with B, it produces a $O(\Delta^2)$-edge coloring for B within the same time bound. In fact, the extra work needed amounts to the maintenance of one color-tag for each edge, which is suitably updated during the sorting and prefix steps. We transform this initial $O(\Delta^2)$-coloring into a $(2\Delta - 1)$-coloring in two stages: in the first stage, we produce an intermediate $O(\Delta \log \Delta)$-coloring, which is then refined in the second stage to yield the desired $(2\Delta - 1)$-coloring.

Stage 1 In the first stage, we make use of a $(\gamma, 2r - 1, r, \Theta(1/r))$-generalized expander $G = (U', V', E')$, with $|U'| = \Theta(\Delta^2)$, $V' = \Delta \log \Delta$, $\gamma = O(1)$ and $r = \Theta(\log \Delta)$, akin to the one underlying Claim 1, to map each of the $O(\Delta^2)$ colors into $2r - 1$ values in $[1..\Delta \log \Delta]$. This mapping has the property that for any set of colors $D \subset [1..\Theta(\Delta^2)]$, with $|D| \leq \Delta$, we may select, for each $x \in D$, a *palette* of r values in $[1..\Delta \log \Delta]$ such that the same value occurs in at most $O(1)$ palettes. The selection of these subsets is accomplished by running the whittling protocol akin to that of Claim 1, with $q = O(1)$, to determine an r-bundle of congestion q for D. Following the proof of [5, Lemma 8] one can easily show that $1 + \log \Delta$ whittling steps suffice.

By repeating the r-bundle selection procedure for each set of at most Δ colors associated with edges of B incident on either one of the p cells u or a memory module v, we determine, for each edge $(u, v) \in B$, two palettes $\text{Left}_{u,v}, \text{Right}_{u,v} \subset [1..\Delta \log \Delta]$ such that $|\text{Left}_{u,v}| = |\text{Right}_{u,v}| = r$. Since the two palettes were chosen out of a set of $2r - 1$ values, they must have nonempty intersection. We pick

one of the values in the intersection as the new color of (u, v).

Although the above procedure successfully reduces the number of colors to $\Delta \log \Delta$, the resulting $(\Delta \log \Delta)$-coloring may not be legal, since up to q edges of B of the same color may be incident on the same vertex $u \in U$ or $v \in V$. However, we can now easily produce a valid coloring by increasing the number of colors by a factor of $q^2 = O(1)$ as follows. We first assign consecutive ranks to all same-colored edges in B incident on the same node $u \in U$ or $v \in V$ (note that each edge is thus ranked twice). Let now edge (u, v) of color x be the i-th x-colored edge outgoing from node u and the j-th x-colored edge incoming into node v, with $1 \leq i, j \leq q$. We recolor (u, v) with color (x, i, j). The proof that the new coloring is a valid $O\left(q^2 \Delta \log \Delta\right)$-coloring is straightforward.

Processor P_i computes $O(\Delta)$ palettes sequentially, namely, $\text{Left}_{u_i,v}$ for each $(u_i, v) \in B$ and Right_{u,v_i} for each $(u, v_i) \in B$. Using sequential integer sorting and prefix to implement the whittling procedure, this task is accomplished in $O(\Delta)$ time per palette, for a total of $O\left(\Delta^2\right)$ local computation time. Then, the sets Left_{u_i,v_j} and Right_{u_i,v_j} are exchanged between P_i and P_j, so that both processors may select the same color $x \in [1..\Delta \log \Delta]$ for (u_i, v_j). Subsequently, processor P_i performs the prescribed edge-ranking for the edges outgoing from u_i and the edges incoming into v_i in $O(\Delta)$ time and, finally, P_j sends the rank information for (u_i, v_j) to P_i so that the latter processor is able to compute the final, legitimate color of (u_i, v_j). Based on the initial $O\left(\Delta^2\right)$-coloring of B, all the communications needed in Stage 1 can be completed in $O\left(\Delta^2 \log \Delta\right)$ time, which is also the total time required by the stage.

Stage 2 In order to refine the intermediate $O(\Delta \log \Delta)$-coloring to produce a $(2\Delta - 1)$-coloring, we use the following standard, greedy technique. For each intermediate color in $[1..O(\Delta \log \Delta)]$ in turn, the processors examine the edges with that color and recolor each such edge (u, v) with the smallest color in $[1..2\Delta - 1]$ which has not already been assigned to any other edge incident on either u or v. The correctness of the above protocol follows immediately from the observations that in a valid edge coloring, edges sharing the same color form a partial matching, and that each edge of the bundle is adjacent to at most $2\Delta - 2$ other edges.

For each intermediate color in $[1..O(\Delta \log \Delta)]$, the OCPC processors perform the prescribed recoloring of the corresponding edges in $O(\Delta)$ time. As a consequence, the overall running time of Stage 2 is $O\left(\Delta^2 \log \Delta\right)$, which is also the final running time for the entire coloring procedure. □

Lemma 4 follows from Claims 1 and 2 by adding up the times needed to determine the bundle B, to compute the $(2\Delta - 1)$-coloring, and to route the component copies corresponding to the edges in B.

It has to be observed that the edge coloring procedure may be considerably simplified by skipping Stage 1 altogether and by refining the initial $O\left(\Delta^2\right)$ coloring directly into a $(2\Delta - 1)$-coloring, by means of the iterative procedure of Stage 2 with $O\left(\Delta^2\right)$ iterations. However, the overall time requirement of the coloring would increase to $\Theta\left(\Delta^3\right) = \Theta\left(\log^3 p\right)$, which would then force us to choose $s = \Theta\left(\log^3 p / \log\log p\right)$ (rather than $s = \Theta\left(\log^2 p\right)$). In turn, this higher value of

s would yield a larger running time of $O\left(\left((n/p)\log\log p + h\log^6 p\right)\right)$ for the entire branch-and-bound algorithm.

4. Conclusions

In this paper, we have devised two distinct implementations of the generic, deterministic parallel algorithm for branch-and-bound of [6] for the distributed-memory, Optically Connected Parallel Computer (OCPC).

Implementation 1 achieves linear speedup for large solution trees complying with a rather mild balance requirement (namely, $n/\max\{p^3, p(\log n \log p)^2\} = \Omega(h)$), which is likely to be met by most practical scenarios, where branch-and-bound is employed for the exploration of hard combinatorial optimization problems on coarse-grained parallel platforms featuring few processing elements. Implementation 1 is extremely simple and relies on the efficient management of a Parallel Priority Queue providing p-wise insertions and min-extractions. Moreover, for trees exhibiting the above-mentioned balance requirement, its optimal running time asymptotically outperforms the one of the randomized algorithm of [8] by a $\Theta(\log n)$ factor on the OCPC.

In contrast, Implementation 2 aims at better handling the more challenging case of trees characterized by small n/h ratios. To this end, we must resort to rather involved shared memory simulation techniques, needed to enable an efficient use of pointers in a distributed environment. In addition, such techniques rely on nonuniform memory maps, based on highly expanding graphs, which are hard to construct [4]. Finding simpler yet efficient strategies to deal with unbalanced trees on the OCPC and, in general, on distributed-memory machines, remains an open problem.

Another interesting direction for future work is to test our theoretically efficient branch-and-bound strategies (possibly simplified) for the solution of optimization problems arising in practical applications on real parallel platforms, and to compare their performance against that of known strategies, such as those proposed in [10, 1].

Acknowledgments

The authors wish to thank the anonymous referees of IRREGULAR'99 for their valuable feedback on the conference version of the paper. This research was supported, in part, by the CNR of Italy under Grant CNR97.03207.CT07 *Load Balancing and Exhaustive Search Techniques for Parallel Architectures*, and by MURST of Italy in the framework of the Project *Methodologies and Tools of High Performance Systems for Multimedia Applications*.

References

1. R. Diekmann, R. Lüling, and A. Reinefeld, "Distributed combinatorial optimization," *Proc. 20th Seminar on Current Trends in Theory and Practice of Informatics - SOFSEM'93*, Hrdoňov, CZ, Dec. 1993, pp. 33–60.
2. G. Frederickson, "The information theory bound is tight for selection in a heap," *Proc. 22nd ACM Symp. on Theory of Computing*, Baltimore, MD, May 1990, pp.

26–33.

3. L.A. Goldberg, M. Jerrum, F.T. Leighton and S. Rao, "A doubly logarithmic communication algorithm for the completely connected optical communication parallel computer," *Proc. 5th ACM Symp. on Parallel Algorithms and Architectures*, Velen, D, Jun./Jul. 1993, pp. 300–309.
4. K. Herley, "Representing shared data on distributed-memory parallel computers," *Math. Syst. Theory* **29** (1996) 111–156.
5. K. Herley, A. Pietracaprina and G. Pucci, "Implementing shared memory on multidimensional meshes and on the fat-tree," in *Algorithms – ESA'95, Proc. 3rd Annual European Symp. on Algorithms*, ed. P. Spirakis (Springer, Berlin, 1995) pp. 60–74.
6. K. Herley, A. Pietracaprina and G. Pucci, "Fast deterministic parallel branch-and-bound," *Parallel Proc. Lett.* (1999). To appear.
7. C. Kaklamanis and G. Persiano, "Branch-and-bound and backtrack search on mesh-connected arrays of processors," *Math. Syst. Theory* **27** (1995) 471–489.
8. R.M. Karp and Y. Zhang, "Randomized parallel algorithms for backtrack search and branch and bound computation," *J. of the ACM* **40** (1993) 765–789.
9. F.T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays • Trees • Hypercubes* (Morgan Kaufmann, San Mateo, CA, 1992).
10. R. Lüling and B. Monien, "Load balancing for distributed branch & bound algorithms," *Proc. 6th Int. Parallel Processing Symp.*, Beverly Hills, CA, Mar. 1992, pp. 543–549.
11. M.C. Pinotti and G. Pucci, "Parallel priority queues," *Inf. Proc. Lett.* **40** (1991) 33–40.
12. F.P. Preparata, "Holographic dispersal and recovery of information," *IEEE Trans. on Inf. Theory* **IT-35** (1989) 1123–1124.
13. M.O. Rabin, "Efficient dispersal of information for security, load balancing, and fault tolerance" *J. of the ACM* **36** (1989) 335–348.
14. E. Upfal and A. Wigderson, "How to share memory in a distributed system," *J. of the ACM* **34** (1987) 116–127.