

Deterministic Branch-and-Bound on Distributed Memory Machines^{*}

(Extended Abstract)

Kieran T. Herley¹, Andrea Pietracaprina², and Geppino Pucci²

¹ Department of Computer Science, University College Cork, Cork, Ireland.
k.herley@cs.ucc.ie

² Dipartimento di Elettronica e Informatica, Università di Padova, Padova, Italy.
{andrea,geppo}@artemide.dei.unipd.it

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 each 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 Machine. 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 $O(n/p + h(\max\{p, \log n \log p\})^2)$ for general values of n , and can be made to run in time $O((n/p + h \log^4 p) \log \log p)$ for n polynomial in p . For large ranges of the relevant parameters, our algorithm is provably optimal or outperforms the well-known randomized strategy by Karp and Zhang.

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 [KZ93], 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 labelled with distinct integer-valued *costs*, the cost of each node being strictly less than

^{*} 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*.

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 *Optical Communication Parallel Computer* (OCPC) in the literature [GJRL93], 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.

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 [Fre90], 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 [KZ93], 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 a DMM would require $O(\log(n/p) + \log p / \log \log p)$ time per step, with high probability, if both priority queue and contention costs are fully accounted for. The resulting algorithm is nonoptimal for all values of the parameters n , h and p .

Kaklamanis and Persiano [KP95] present a deterministic branch-and-bound algorithm that runs in $O(\sqrt{nh} \log n)$ time on an n -node mesh. The mesh-specific techniques exploited by the algorithm, coupled with their assumption that the mesh size and

problem size are comparable, limits 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 [Fre90] appears in [HPP99]. The main result of this paper extends this approach to the distributed-memory DMM with the performance stated in the following theorem.

Theorem 1. *There is a deterministic algorithm running on a p -processor DMM 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/p 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 $h = O(n/(p \log^4 p))$, which is a weak balance requirement on the solution tree.

The rest of this abstract is organized as follows. Section 2 presents our general branch-and-bound strategy, while Section 3 sketches its DMM implementation.

2 A machine independent algorithm

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 constant degree. We begin by summarizing some terminology introduced in [Fre90]. For a set of tree nodes N , let $Best(N)$ denote the set containing the (at most) s internal or leaf 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 $N - Best(N)$. Clan C has two 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 $cost(C) = \infty$ (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 $Off(C) \cup PR(C)$, both $cost(C')$ and $cost(C'')$ are strictly greater than $cost(C)$.

It can be shown 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 such a property, Frederickson [Fre90] 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^* .

Our branch-and-bound algorithm for the DMM can be seen as an implementation of a general strategy based on a parallel exploration of \mathcal{TC} , which was also at the base of the PRAM algorithm presented in [HPP99]. Such a strategy is realized by the generic algorithm BB reported below, which applies to any p -processor machine. In the next section, we will show how to implement algorithm BB efficiently on the DMM.

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 labelled with an integer-valued key [PP91]. 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, \dots, 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.

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

A complete proof of Lemma 1 can be found in [HPP99]. Here, we briefly sketch the main ingredients of the proof. Correctness follows from the observation that at any time during the algorithm, all nodes in T with cost less than or equal to the current value of q

have already been visited and made members of some clan. Therefore, when ℓ becomes smaller than q , the algorithm will have visited at least one leaf (the one with cost ℓ) and all nodes (and, in particular all leaves) with cost less than or equal to $\ell < q$. The upper bound on the number of iterations crucially relies on the fact that clans containing exactly s nodes in T^* (called *good clans*) belong to the first $O(hs)$ levels of the tree of clans \mathcal{TC} , and that at each iteration of Step 2 but the last, at least one good clan is extracted from the PPQ. We call an iteration of Step 2 *full*, if exactly p good clans are extracted from the PPQ, and *partial* otherwise. Clearly, there can be no more than $n/(ps)$ full iterations. On the other hand, each partial iteration must complete the visit of all good clans at some level of \mathcal{TC} , hence there cannot be more than $O(hs)$ partial iterations.

3 DMM Implementation

In this section, we show how algorithm BB can be efficiently implemented on the DMM. 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 DMM so that both Insert and Deletemin operations take $O(\log(|Q|/p) \log p)$ time.*

Proof (Sketch). Use the p -Bandwidth Heap of [PP91] and store each of the p items held by a heap node in a distinct memory module. Then, the PRAM algorithms for Insert and Deletemin can be run directly by replacing the PRAM merging algorithm with a simple adaptation of bitonic merging for the DMM, 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 $\Theta(s)$, 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 be concentrated in a small number of memory modules, which might 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 running times depend on the relative values of n/p and h .

In what follows, we assume that the values of n and h be known in advance to the DMM processors. In the full paper, we will show how to relax such an assumption by means of guessing techniques similar to those employed in [HPP99] for the PRAM implementation.

Implementation 1 Let $s = \max\{p, \log n \log p\}$, 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 time $O(\log n \log p)$ 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 Fredrickson'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 .

Implementation 2 Note that the above implementation achieves optimal $\Theta(n/p)$ running time when n/p is much larger than h , however it becomes progressively less profitable for unbalanced trees. In the latter scenario, it is convenient to choose a smaller value for s which, however, requires a more sophisticated mechanism to avoid memory contention. In particular, when $s < p$ it becomes necessary to introduce some redundancy in the data representation 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 in the rest of the paper.

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* modelled by means of a bipartite graph $G = (U, V, E)$ with $|U|$ inputs, corresponding to the virtual cells, and $|V| = p$ outputs, corresponding to the DMM 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, \dots, C_p to be stored in the DMM 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, by using an information dispersal algorithm [Pre89,Rab89], so that any $k/3$ components suffice to recreate the original contents of u_i . The following lemma, proved in Subsection 3.1, 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 $k/3$ components 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 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 DMM 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.

Lemma 4. *There exists a suitable memory map G such that any set S of p virtual cells can be stored/retrieved in the DMM modules in $O(s + \log^2 p \log \log p)$ steps.*

Using the above techniques, 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 $O(\log(|Q|/p) \log p) = O(\log^2 p)$ and distributed one per processor. Then, by Lemma 4, $k/3$ components for each cell are retrieved in $O(s + \log^2 p \log \log p)$ time. Finally, the contents of each clan C are reconstructed in $O(|C| \log k) = O(s \log \log p)$ time, by Lemma 3. By combining all of the above contributions we obtain a running time of $O(s \log \log p + \log^2 p \log \log p)$. 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 ($O(s \log \log p + \log^2 p \log \log p)$ 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. Therefore, each iteration can be completed in time $O(s \log \log p + \log^2 p \log \log p)$. Since, by Lemma 1, there are $O(n/(ps) + hs)$ iterations, the second part of Theorem 1 follows by choosing $s = \log^2 p$.

3.1 Proof of Lemma 3

We will only describe the encoding procedure, since the reconstruction procedure embodies similar 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 an information dispersal algorithm [Pre89,Rab89], each row can be independently recoded into a string of k bits, so that any $k/3$ 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 columns is represented as a sequence of $\Theta(s/k)$ words. Each of these $\Theta(s/k)$ -word sequences constitutes a distinct component.

Rather than actually running the information dispersal algorithm, we can use a pre-computed 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.

The repackaging mentioned above can be implemented efficiently by transposing each $k \times k$ block of A'_u (k consecutive rows) in time $O(k \log k)$ through an oblivious sequence of bit manipulations which involve the k words that make up the block. (More details will be provided in the full version of this paper.)

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 is performed locally at each processor and requires no external communication.

3.2 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 of congestion h for S is a subset $B \subseteq E(S)$ where each $u \in S$ has degree c and any $v \in V$ has degree at most h 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 $d = \Theta(\log p)$ sufficiently large, 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 $\Theta(d)$. Moreover, one such bundle can be constructed in $O(d \log p) = O(\log^2 p)$ time on the DMM.*

Proof (Sketch). The existence of the bundle is a consequence of the results of [Her96], while its construction can be achieved through the techniques introduced in [UW87].

Let us consider the problem of writing a set S of p cells, u_1, u_2, \dots, u_p (the problem of reading p cells is similar). Suppose that each processor P_i has encoded the cell u_i it wishes to write into d component copies using the techniques described before, and that the processors have constructed a $(2d/3)$ -bundle B of congestion $\Theta(d)$ for S . 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 protocol implied by Lemma 1 merely indicates to each processor the locations to which the $2d/3$ component copies of its clan should be written: the actual physical movement of the component copies must be implemented as a separate step. This is a nontrivial task, since each processor must transmit/receive one component copy per edge in B , that is, it must dispatch $2d/3$ component copies and receive $O(d)$ of them. The transmission of copies must be coordinated to avoid ‘‘collisions’’ at receiving processors. Straightforward techniques based on sorting prove too slow for our purpose, so we employ an approach based on the idea that a Δ -edge colouring of B allows one word to be transmitted along each edge $(u, v) \in B$ in $O(\Delta)$ time. Let $\Delta = \Theta(d)$ denote the maximum degree of a node in $U \cup V$ with respect to the edges in B . We have:

Claim 2. *An $O(\Delta)$ -edge colouring for B may be computed in $O(\Delta^2 \log \Delta)$ time on the DMM.*

Proof (Sketch). It is easy to modify the construction of Claim 1 so that, together with B , it produces an $O(\Delta^2)$ -colouring for B within the same time bound. (We will use the term ‘‘colouring’’ to refer to edge colouring.) We transform this $O(\Delta^2)$ -colouring into a (2Δ) -colouring in two stages: first we produce an intermediate $O(\Delta \log \Delta)$ -colouring and then refine the intermediate colouring to produce the desired (2Δ) -colouring.

For the first stage we associate a set $\text{Colours}(x)$ of $2r - 1 = \Theta(\log \Delta)$ colours from $[1..O(\Delta \log \Delta)]$ with each $x \in [1..\Delta^2]$. This mapping has the property that for

any subset D of size Δ chosen from $[1.. \Delta^2]$, we may select for each $x \in D$ a *palette* of r colors from $\text{Colours}(x)$ such that no two elements of D have the same colour in their palettes. The choice of the mapping is based on the existence of suitable bipartite maps akin to those underlying Claim 1. (Details will be provided in the full version of this paper.) As a consequence, for each edge $(u, v) \in B$ of color $x \in [1.. \Delta^2]$ we can select two palettes $\text{Left}_{u,v}, \text{Right}_{u,v} \subset \text{Colours}(x)$ of r colours such that $\text{Left}_{u,v} \cap \text{Left}_{u,v'} = \emptyset$ for every $(u, v') \in B$ with $v' \neq v$, and $\text{Right}_{u,v} \cap \text{Right}_{u',v} = \emptyset$ for every $(u', v) \in B$ with $u' \neq u$. However, there must be at least one colour in $\text{Left}_{u,v} \cap \text{Right}_{u,v}$. This latter colour from $[1.. O(\Delta \log \Delta)]$ is adopted as the intermediate colour for (u, v) .

Processor P_i computes sequentially all sets $\text{Left}_{u_i,v}$ for each $(u_i, v) \in B$ and all sets Right_{u,v_i} for each $(u, v_i) \in B$. By means of standard techniques [UW87], this task can be accomplished in $O(\Delta \log \Delta)$ local computation time. Then, each set Right_{u_i,v_j} is transmitted from P_j to P_i . Based on the initial $O(\Delta^2)$ -colouring of B , this communication is completed in $O(\Delta^2 \log \Delta)$ steps. Overall, the first stage takes $O(\Delta^2 \log \Delta)$ time.

The second stage involves refining the intermediate $O(\Delta \log \Delta)$ -colouring to produce a (2Δ) -colouring, using the following standard technique. For each intermediate colour in $[1.. O(\Delta \log \Delta)]$ in turn, examine the edges with that colour and recolor each such edge (u, v) with the smallest colour in $[1.. 2\Delta]$ that does not clash with any new colour of other edges incident on u or v already assigned. It is easy to see that the second stage can be completed in $O(\Delta^2 \log \Delta)$ time. This is also the running time for the entire colouring procedure.

Given a $O(\Delta)$ -colouring for B , the component copies corresponding to the bundle can be accessed in $2\Delta = \Theta(\log p)$ phases: during the i -th phase, the component copies corresponding to edges bearing color i are transferred in $O(s/k) = \Theta(s/\log p)$ time. Therefore, the access is completed in overall $O(s)$ time.

Lemma 4 follows by adding up the times needed to determine the bundle B , to compute the $O(\Delta)$ -colouring and to access the component copies.

References

- [Fre90] G. Frederickson. The Information Theory Bound is Tight for Selection in a Heap. In *Proc. of the 22nd ACM Symp. on Theory of Computing*, pages 26–33, May 1990.
- [GJRL93] L.A. Goldberg, M. Jerrum, F.T. Leighton and S. Rao. A Doubly Logarithmic Communication Algorithm for the Completely Connected Optical Communication Parallel Computer. In *Proc. of the 5th ACM Symp. on Parallel Algorithms and Architectures*, pages 300-309, June/July 1993.
- [Her96] K. Herley. Representing Shared Data on Distributed-Memory Parallel Computers. *Mathematical Systems Theory*, 29:111-156, 1996.
- [HPP99] K. Herley, A. Pietracaprina and G. Pucci. Fast Deterministic Parallel Branch-and-Bound. *Parallel Processing Letters*, to appear.
- [KP95] C. Kaklamanis and G. Persiano. Branch-and-Bound and Backtrack Search on Mesh-Connected Arrays of Processors. *Mathematical Systems Theory*, 27:471–489, 1995.
- [KZ93] R.M. Karp and Y. Zhang. Randomized Parallel Algorithms for Backtrack Search and Branch and Bound Computation. *Journal of the ACM*, 40:765–789, 1993.

- [PP91] M.C. Pinotti and G. Pucci. Parallel Priority Queues. *Information Processing Letters*, 40:33–40, 1991.
- [Pre89] F.P. Preparata. Holographic Dispersal and Recovery of Information. *IEEE Trans. on Inform. Theory*, IT-35(5):1123–1124, May 1989.
- [Rab89] M.O. Rabin. Efficient Dispersal of Information for Security, Load Balancing, and Fault Tolerance. *Journal of the ACM*, 36(2):335–348, April 1989.
- [UW87] E. Upfal and A. Wigderson. How to Share Memory in a Distributed System. *Journal of the ACM*, 34(1):116–127, Jan 1987.