Towards a Taxonomy of Parallel Branch and Bound Algorithms

Harry W.J.M. Trienekens
(harryt@cs.few.eur.nl)

Arie de Bruin
(arie@cs.few.eur.nl)

Erasmus University Rotterdam

ABSTRACT

In this paper we present a classification of parallel branch and bound algorithms, and elaborate on the consequences of particular parameter settings. The taxonomy is based upon how the algorithms handle the knowledge about the problem instance to be solved, generated during execution. The starting point of the taxonomy is the generally accepted description of the sequential branch and bound algorithm, as presented in, for example, [Mitten 1970] and [Ibaraki 1976a, 1976b, 1977a, 1977b].

1980 Mathematical Subject Classification: 90C27, 68Q10, 68R05.

Key Words & Phrases: branch and bound, taxonomy, parallelism, nondeterminism, asynchronicity.

1. THE BRANCH AND BOUND ALGORITHM

Branch and bound algorithms solve optimization problems by partitioning the solution space. Throughout this paper, we will assume that all optimization problems are posed as minimization problems, and that solving a problem is tantamount to finding a feasible solution with minimal value. If there are many such solutions, it does not matter which one is found.

A branch and bound algorithm can be characterized by four rules [Mitten 1970]: a *branching rule* defining how to split a problem into subproblems, a *bounding rule* defining how to compute a bound on the optimal solution of a subproblem, a *selection rule* defining which subproblem to branch from next, and an *elimination rule* stating how to recognize and eliminate subproblems that cannot yield an optimal solution to the original problem. In the remainder of this paper, we will call these four rules the *basic rules* of the branch and bound algorithm.

The concept of *heuristic search* provides a framework to compare all kinds of selection rules, for example, depth first, breadth first, or best bound [Ibaraki 1976b]. In a heuristic search, a *heuristic function* is defined on the set of subproblems. This function governs the order in which the subproblems are branched from. The algorithm always branches from the subproblem with the smallest heuristic value.

As regards the elimination rule, three types of tests for eliminating subproblems are known. Firstly, the *feasibility test*: a subproblem can be eliminated if it can be proven not to have a feasible solution. Secondly, the *bound test*: a subproblem can be eliminated if its bound is better than or equal to the value of a known feasible solution. Finally, the *dominance test*: a subproblem that is dominated by another subproblem can be eliminated. A subproblem dominates another subproblem if it can be proven that the optimal solution to the former subproblem is better than or equal to the optimal solution to the latter

Report EUR-CS-92-01
Erasmus University Rotterdam, Department of Computer Science
P.O. Box 1738, 3000 DR Rotterdam, The Netherlands

subproblem. Note that the bound test can be viewed upon as a special case of the dominance test.

An *active subproblem* is a subproblem that has been generated and hitherto neither completely branched from nor eliminated. The active subproblems can be divided into two categories: those that are currently being branched from, and those that are currently not being branched from. In each stage of the computation, there exists an *active set*, i.e., the set containing all active subproblems that are not being branched from at that moment.

A subproblem is said to be a *currently dominating subproblem* if it has been generated and has not been dominated so far.

A possible sequential implementation of a branch and bound algorithm can be described as follows. There is a main loop in which the following steps are repeatedly executed. Using the selection rule, one of the subproblems in the active set is chosen to branch from. This subproblem is extracted from the active set, and decomposed into smaller subproblems using the branching rule. For each of the subproblems thus generated, the bounding rule is used to calculate a bound. If during the computation of this bound the subproblem is solved, i.e., the optimal solution to the subproblem is found, the value of the best known solution is updated, i.e., if the former value is better than the latter, the value of the latter is set to the value of the new solution. The value of the best known solution is an upper bound on the value of the optimal solution to the original problem. If a subproblem is not solved during computation of the bound, the subproblem is added to the active set. Finally, the elimination rule is used to prune the active set. The computations continue until there are no more subproblems in the active set. If this is the case, the best solution obtained so far is an optimal solution to the original problem.

The four basic rules constitute only a partial characterization of a branch and bound algorithm: the rules state the outcome of certain actions to be taken by the algorithm, but they do not state anything about how and when the algorithm will take these actions. As a consequence, each set of basic rules actually applies to a class of branch and bound algorithms. Algorithms in the same class use the same rules, but in a different way. Hence, to arrive at a complete characterization of a branch and bound algorithm, the way in which the algorithm uses the basic rules has to be specified.

An example of a branch and bound algorithm that belongs to the same class as the algorithm presented above is an algorithm that always works upon the current best subproblem: each time the algorithm completes the generation of a child that has a better heuristic value than its parent, the algorithm suspends the branching from the parent in order to start the branching from this child; the branching from the parent continues as soon as the branching from the child is completed.

1.1. Parallel Branch and Bound Algorithms

Branch and bound algorithms can be parallelized at a *low* or at a *high* level.

In case of *low level parallelization*, only part of the sequential branch and bound algorithm is parallelized in such a way that the interactions between the parallelized part and the other parts of the algorithm do not change. For example, the computation of the bound, the selection of the subproblem to branch from next, or the application of the elimination rule could be performed by several processes in parallel. Because the interactions between the various parts of the algorithm are not changed, low level parallelism does not have consequences for the branch and bound algorithm as a whole. The overall behavior of the thus created parallel branch and bound algorithm resembles the behavior of the original sequential branch and bound algorithm, i.e., the parallel algorithm will branch from the same subproblems in the same order.

In case of *high level parallelization*, the effects and consequences of the parallelism introduced are not restricted to a particular part of the branch and bound algorithm, but influence the algorithm as a whole. The thus created parallel algorithm is essentially different. The work performed by the parallel algorithm need not be equal to the work performed by the sequential algorithm. The order in which the work is performed can differ, and it is even possible that some parts of the work performed by the parallel algorithm are not performed by the sequential algorithm, or vice versa. For example, several iterations of the main loop can be performed in parallel (e.g., several processes executing the algorithm branch in parallel from their own subproblem).

2. USING THE KNOWLEDGE

In this section we investigate the mechanics of the (sequential) branch and bound algorithm. We do this by investigating how exactly the branch and bound algorithm solves a problem instance.

In section 2.1 we investigate what exactly knowledge is.

In section 2.2 we take a close look at how the branch and bound algorithm takes decisions. We divide the information used by the various rules and tests into categories depending on the origin of the information, and show that the category of information has consequences for the consistency of the decisions.

In section 2.3 we make a little excursion into the realm of program correctness to develop tools to state and analyze properties of algorithms. We use these tools to show the correctness of the branch and bound algorithm.

In section 2.4 we take a look at how the branch and bound algorithm handles the knowledge it generates during execution, and introduce the notion of exhaustive knowledge handling.

In section 2.5 we generalize the tools mentioned above for use with parallel algorithms.

2.1. Knowledge

During execution, a branch and bound algorithm generates *knowledge* about the problem instance to be solved. This knowledge consists amongst other things of all subproblems generated, branched from, and eliminated, bounds on the value of the optimal solution, and the feasible solutions, and upper bounds found. All decisions taken by the algorithm are based upon this knowledge, possibly augmented with additional information supplied by the environment in which the algorithm is executed. Examples of the latter type of information are the current amount of free memory in the computer system executing the algorithm, and ‘hints’ from the user.

Redundant knowledge is knowledge that is generated somewhere in the past by the algorithm, that from now on will never be used by the algorithm. An example of redundant knowledge is an old feasible solution, whose value has been improved upon by the value of a new feasible solution. Because redundant knowledge will never be used by the algorithm (again), there is no need to preserve it any longer.

Depending upon the exact characteristics of a branch and bound algorithm, part of the knowledge generated will be provably redundant. We will use the term *valid knowledge* to denote knowledge of which the algorithm cannot (yet) decide that it is redundant. For example, if a branch and bound algorithm uses an elimination rule that does not involve dominance tests, knowledge about subproblems branched from or eliminated can be proven to be redundant (because once a subproblem is branched from or eliminated, there is no way in which the algorithm can use this subproblem again). Note that additional knowledge can show knowledge that is currently valid to be redundant.

2.2. Categories of Knowledge

A branch and bound algorithm repeatedly applies rules and tests to sets of objects. For example, the bounding rule is used to compute a bound on the optimal value of a given subproblem, the branching rule is used to split a given subproblem into smaller subproblems, the selection rule is used to select a subproblem from a given set of subproblems, and the feasibility test is used to determine whether or not it can be proven that a given subproblem does not have a feasible solution.

The precise set of objects a rule or test is applied to, depends upon this particular rule or test. The bounding and branching rules, and the feasibility test, are all applied to singleton sets, i.e., sets consisting of a single subproblem. The set of objects the selection rule is applied to, consists of the subproblems in the active set. The set of objects the bound and dominance tests are applied to, consists of all active subproblems and the current upper bound, or all currently dominating subproblems respectively.

The outcome of the application of a given rule or test to a given set of objects depends upon the contents of the set, but it might also be dependent upon additional information. For example, the computation of a bound may depend upon the current value of the upper bound (e.g., [Jonker & Volgenant 1982]), and the outcome of the branching rule can depend upon a branching scheme that was found to be successful in other parts of the search tree. Note that in these examples the upper bound, or the successful branching scheme respectively, is not an element of the set of objects the rule or test is applied to.

The information that can be used by a rule or a test when it is applied to a set of objects can be divided into two major categories: controllable information, and non-controllable information. *Controllable information* consists of the knowledge generated by the algorithm itself while solving the problem, for example, all feasible solutions found. *Non-controllable information* consists of information not generated by the algorithm itself, but by the environment in which the algorithm is executed. For example, while solving integer programming problems, the user could be asked to enter the next cutting plane to be used [Grötschel 1980], or some test used by the algorithm could base its outcome on the amount of free memory [Ibaraki & Katoh 1991].

Controllable information can be subdivided into local information and global information. *Local information* consists of all information derivable from the set of objects that the rule or test is applied to. For example, the information used by a selection rule that selects the subproblem with the smallest heuristic value, can be determined completely from the set of subproblems this rule is applied to. *Global information* consists of all information derivable from all existing objects. For example, global information is used when a bounding rule improves upon its initial bound by applying a Lagrangean technique that uses the current value of the upper bound [Shapiro 1979]. Or, a feasibility test might be able to prove that a given subproblem does not have a feasible solution because somewhere in the past a similar subproblem has been proven by complete decomposition to have no feasible solution. Note that local information is a subset of global information.

The description of the branch and bound algorithm as presented in the first section does not state anything about the specific information used by the various rules and tests. Hence this description is incomplete: it does not state the knowledge to be used. If the algorithm uses global or non-controllable information, it enhances the understanding of the algorithm if this information is added as an additional input parameter to the specific rule or test.

In theory, global information is under control of the algorithm because the algorithm decides whether and when to create an object. However, as soon as the order in which the various objects are generated changes, rules and tests based upon global information can yield different outcomes when applied to the same set of objects. Hence, if the order in which the objects are generated differs among consecutive executions of a branch and bound algorithm that uses rules and tests based upon global information, it is very hard to compare these executions. This is even more so for rules and tests based upon non-controllable information.

Note that for a careful comparison and analysis of consecutive executions of the same branch and bound algorithm solving the same problem instance, it is desired that the decisions taken are consistent for the various runs, i.e., independent of the order of application of the various rules and tests. The application of the same rule or test to the same set of objects should always yield the same outcome. This can only be guaranteed if all rules and tests use only local information.

2.3. Invariants

In this section we make a little excursion into the realm of program correctness to introduce the notions of assertion and invariant. These notions constitute a mechanism for stating basic properties of algorithms in a concise and precise way, thus providing for a good starting point for deriving other properties. The notions also allow for the splitting of long and tedious proofs into small, and well-organized parts by providing a mechanism for describing the exact points the various parts have to be joined in order to form the complete proof.

At the end of this section, we use these notions to show the correctness of the branch and bound algorithm. In the remainder of this paper, we will use these notions to describe properties of branch and bound algorithms.

For a more complete description (and the exact formal definitions) of program correctness, we refer to [Reynolds 1981].

An *algorithm* can be conceptualized as a prescription defining a series of actions to be performed upon a set of *variables*, i.e., memory locations which can accommodate a value. The *state* of a variable is defined as the value stored in it. The series of actions to be performed is called the *program*. The order in which the actions are actually performed during execution need not equal the order in which the actions are specified in the program. Some actions can be performed more than once, whereas other actions might never be performed.

An algorithm is executed by a *process*. A process consists of the program to be executed, a set of variables to be operated upon, and a *program counter*. This counter denotes the action in the program the process is currently performing. The state of a process is defined as the states of its variables. To prevent ambiguity about the state of a variable when an action is being applied to this variable, the state of such a variable is defined as the state at the time the action started.

Let *pred* denote a predicate on the state of a process, let *progc* denote the program counter of this process, and let *pc* denote a value the program counter can take, i.e., a specific point in the program the process is executing. Let a *program counter set PCS* be a set of values the program counter can take during execution, i.e.,

(2.1) $$PCS = \{pc_1, pc_2, \ldots, pc_{n_{PCS}}\},$$

with n_{PCS} the number of program counter values in this set. Finally, let *PCS** denote the set containing all possible values the program counter can take during execution.

An *assertion* is a statement that a predicate will be true whenever a given point in the program is reached. The statement $assert(pred, pc)$ is true for a given algorithm if and only if each time the program counter of a process executing the algorithm equals *pc*, the predicate *pred* on the state of the process is true, i.e.,

(2.2) $$assert(pred, pc) = \textbf{true} \quad <==> \quad ((progc = pc) ==> (pred = \textbf{true})).$$

An *invariant* is a statement that an assertion is true for a series of points in the program. The statement $invar(pred, PCS)$ is true for a given algorithm if and only if the predicate *pred* is true for all program counter values in *PCS*, i.e.,

(2.3) $$invar(pred, PCS) = \textbf{true} \quad <==> \quad ((pc \in PCS) ==> (assert(pred, pc) = \textbf{true})).$$

The set *PCS* is called the *invariance set* of the invariant, and the points in this set are called *invariance points*.

Having defined the notions of assertion, and invariant, we now will use these notions to show the correctness of the branch and bound algorithm.

The argument is easy using the following invariant: for each possible value of the program counter, the value of the optimal solution to the original problem is either the value of the best solution found hitherto, or the value of the best solution to the active subproblems, i.e., the best solution to be found while solving the active subproblems. Initially, there is no solution found hitherto, and the root, i.e., the original problem, is the only active subproblem. Therefore, the predicate in the invariant is initially true. Each action of the algorithm preserves the truth of the invariant predicate. Therefore, at the end of the execution of the algorithm this predicate still holds, which implies that the value of the optimal solution to the original problem is the value of the best solution found during the branch and bound process (because there are no more active subproblems).

2.4. Knowledge Handling

In this section we take a look at how a branch and bound algorithm handles the knowledge it generates during execution. We define three specific properties of ways the algorithm can handle the knowledge generated. In the remainder of this paper, these properties will be used to classify parallel branch and bound algorithms.

A branch and bound algorithm decides which knowledge to generate, store, and use during execution. The four basic rules, and the order in which they are applied, determine the knowledge to be generated. The algorithm decides which knowledge will be stored, and which knowledge will be discarded. The knowledge stored in turn determines the outcomes of the various rules and tests to be applied, and hence the knowledge to be generated in the future.

The decisions of which knowledge to generate, store, and use influence the work to be done. Each successive step to be performed during the execution of a branch and bound algorithm depends upon the knowledge obtained thus far. Therefore, changing the basic rules, or the way these rules are used, has consequences for the execution of the algorithm. For example, the use of another selection or bounding rule can generate different knowledge, and thus lead to a different order in which the subproblems are branched from, whereas the use of another branching rule can result in a complete different search tree. Without making additional assumptions, nothing can be said about how these changes will influence the actual execution and the work involved in it. Changing a rule or test can lead to all kinds of anomalies. The total amount of work done can increase or decrease by an arbitrary factor [Ibaraki 1976a, 1977a, 1977b].

New knowledge can be used by the branch and bound algorithm as soon as it is generated. However, the algorithm need not do so. The algorithm can intentionally introduce delays between the generation of new knowledge and the use of this knowledge. For example, the algorithm can neglect the additional knowledge generated by the creation of the children until the branching from the parent has completed and the algorithm has to apply the selection rule to determine the next subproblem to branch from.

Branch and bound algorithms can be classified according to the way the process executing them handles the knowledge generated. In this regard, we define the following three predicates:

$PRED_1$: The process is aware of all valid knowledge generated thus far with respect to the elimination rules.

$PRED_2$: The process is aware of all the work still to be completed generated thus far to solve the problem instance on hand.

$PRED_3$: The work the process is executing is consistent with the knowledge the process is aware of, i.e., the work the process is performing is the most suitable work the process is aware of, and this work cannot be eliminated using the knowledge the process is aware of.

The above formulation of the predicates is not free from some ambiguity. This ambiguity is due to the fact that the precise formulation of the predicates depends heavily upon the precise characteristics of the particular branch and bound algorithm under consideration. However, given the precise description of the algorithm, the predicates can be formulated in a straightforward way.

As regards the sequential branch and bound algorithm, $PRED_1$ and $PRED_2$ are trivially true as long as the algorithm does not discard valid knowledge. The ideas behind these predicates will surface once we have advanced to parallel branch and bound algorithms (cf. section 3.3).

If $PRED_1$ and $PRED_3$ are not both true, it is possible that the process does not eliminate a given subproblem even though enough knowledge has been generated for such an elimination. Hence it is possible that the process branches from subproblems that it could have eliminated. If $PRED_2$ and $PRED_3$ are not both true, it is possible that the work the process is performing is not the work with the highest priority.

Intuitively one expects that if all three predicates always hold, the algorithm cannot improve its performance. However, this intuition turns out to be false. We refer to [Trienekens 1990] for some examples of misled intuition, in which it is shown that it can be worthwhile to branch from a subproblem that can be eliminated, as well as that it can be worthwhile to perform work with lower priority.

A branch and bound algorithm has *exhaustive knowledge handling* for an invariance set *PCS* if and only if for the process executing the algorithm the following statement is true:

(2.4) $invar(PRED_1 \textbf{ and } PRED_2 \textbf{ and } PRED_3, PCS)$.

In the remainder of this paper we will repeatedly use the notion of exhaustive knowledge handling. We therefore introduce the mnemonic *EXH* to denote the predicate that states such knowledge handling, i.e.,

(2.5) $EXH = (PRED_1 \textbf{ and } PRED_2 \textbf{ and } PRED_3)$.

Let for a given branch and bound algorithm and a given invariance set *PCS* the statement $invar(EXH, PCS)$ be true. The invariant states that whenever its program counter *progc* equals *pc* ($pc \in PCS$), the process executing the algorithm has exhaustive knowledge handling. However, nothing is stated about the knowledge handling when the value of the program counter does not belong to *PCS*. The relation between the invariance set *PCS* and the set of all possible program counter values *PCS** says something about when the process (and hence the algorithm) manages to have exhaustive knowledge handling. Branch and bound algorithms can be classified in terms of this relationship.

Examples of exhaustive knowledge handling can be demonstrated from the two branch and bound algorithms presented in section 1. The first algorithm is guaranteed to have exhaustive knowledge handling at the start of each iteration of the main loop. At all other places, it might have exhaustive knowledge handling; however, this cannot be guaranteed. The second algorithm has exhaustive knowledge handling all the time.

2.5. Invariants for Parallel Computing

The notions of assertion and invariant as presented in section 2.3 apply only to a single process. In this section, we will generalize these notions such that they apply to several processes concurrently.

Let Q denote the set of processes executing the parallel algorithm, and let m denote the cardinality of this set. We will use m-tuples for representing the various processes. By convention, a barred variable will denote an m-tuple; the p-th element of this tuple is a variable which corresponds to the p-th process. A barred constant denotes an m-tuple whose elements are all equal to this constant.

The meaning of the various variables to be used in this section is identical to the meaning presented in section 2.3. For example, the m-tuple $\overline{progc} = <progc_1, progc_2, \ldots, progc_m>$ denotes the program counters of the various processes, whereas the m-tuple $\overline{pc} = <pc_1, pc_2, \ldots, pc_m>$ denotes a particular point in the parallel program. Note that this particular point need not be reached during execution.

A predicate $\overline{pred}$ is true if and only if all the constituting predicates in its tuple are true, i.e.,

$$(2.6) \qquad \overline{pred} = \textbf{true} \quad <==> \quad (pred_p = \textbf{true},\ p \in Q).$$

The notions of assertion and invariant as presented in section 2.3 can be generalized in a straightforward manner to be defined upon m-tuples.

From now on, we will call the original notions *local* notions, whereas we will call the new notions *global* notions. The naming convention is derived from the fact that the local notions apply only to a specific process, independent of all other processes, whereas global notions apply to all processes concurrently.

The local notions are special cases of the global notions, as can easily be seen from the following argument. Let q denote the process the local notion applies to ($q \in Q$).

A local predicate $pred$ on process q is equivalent with the following global predicate:

$$(2.7) \qquad \overline{pred}^{\,q} = <pred_1^q, \ldots, pred_m^q>,\ \ (\,(pred_q^q = pred,\ pred_p^q \equiv \textbf{true}),\ (p \in Q,\ p \neq q)\,).$$

A specific point pc in the program of process q corresponds to the following global program counter set:

$$(2.8) \qquad \overline{PC}^{\,q} = <PC_1^q, \ldots, PC_m^q>,\ \ (\,(PC_q^q = \{pc\},\ PC_p^q = PCS^*),\ (p \in Q,\ p \neq q)\,).$$

A local program counter set PCS for process q is equivalent with the following global program counter set:

$$(2.9) \qquad \overline{PCS}^{\,q} = <PCS_1^q, \ldots, PCS_m^q>,\ \ (\,(PCS_q^q = PCS,\ PCS_p^q = PCS^*),\ (p \in Q,\ p \neq q)\,).$$

Using formulae 2.7 - 2.9, the local notions of assertion and invariant as defined in section 2.3 can be straightforwardly formulated as global notions.

3. A TAXONOMY OF PARALLEL BRANCH AND BOUND ALGORITHMS

In this section we present a taxonomy of high level parallel branch and bound algorithms.

In section 3.1 we take a look at the major difference between parallel branch and bound algorithms and sequential branch and bound algorithms: now, knowledge is generated and used concurrently by several processes, instead of by a single process only.

In section 3.2 we present our taxonomy of high level parallel branch and bound algorithms.

In section 3.3 we take a look at how the processes executing a parallel branch and bound algorithm handle the knowledge generated during execution. We define the nine classes of complete knowledge sharing/use, and generalize the notion of exhaustive knowledge handling, introduced in section 2.4.

In section 3.4 we demonstrate the versatility of our taxonomy by classifying and comparing some of the parallel branch and bound algorithms described in the literature.

3.1. Concurrent Knowledge Handling

The main difference between parallel branch and bound algorithms lies in the way the algorithms handle the knowledge generated, i.e., in the way the knowledge generated is shared among the various processes, and in the way the processes detect that new knowledge has been generated and subsequently use this knowledge. Just as in the sequential case, the processes executing a parallel branch and bound algorithm generate knowledge about the problem instance to be solved, and base their decisions of what to do upon this knowledge, possibly augmented with additional information supplied by the environment the algorithm is executing in. However, the knowledge is generated and used by several processes concurrently, instead of by a single process.

The concurrent knowledge handling has two major consequences for the construction of a parallel branch and bound algorithm.

Firstly, because knowledge generated by a particular process can be of use to other processes, there is a need for sharing the knowledge generated among the various processes. For example, a feasible solution found by a particular process can be used by another process to eliminate by a bound test the subproblem it just generated. Therefore this solution ought to be made available to other processes.

Secondly, because the processes can be executing independently of each other, there is a need for a process to become aware of the fact that some other process has generated new knowledge. As long as the branch and bound algorithm is executed by a single process, this process automatically knows when new knowledge is generated. However, as soon as the algorithm is executed by several independent processes, a process need not know anymore when a fellow process generates new knowledge. Hence, the possibility exists that a process cannot use new knowledge, simply because the process does not know of its existence.

The sharing of knowledge among the processes, and the subsequent use of the knowledge by the processes, can be conceptualized using the notion of knowledge bases. A *knowledge base* is an entity which contains knowledge. The knowledge generated by the various processes is transferred to the knowledge bases, and a process accesses the knowledge bases to obtain the knowledge it needs. Transferring the knowledge generated to the knowledge bases is called *sharing the knowledge*, whereas accessing the knowledge bases and subsequently using the knowledge is called *using the knowledge*. The term *knowledge handling* comprises the complete process of sharing and using the knowledge generated.

3.2. Parameters of Parallel Branch and Bound Algorithms

Our taxonomy generates a four dimensional space containing high level parallel branch and bound algorithms. Each dimension corresponds to a parameter of such algorithms. The four parameters are the way in which the knowledge generated is shared among the processes, the way in which the knowledge is actually used, the way in which the work is divided among the various processes, and the synchronicity of the processes.

The knowledge generated during execution is stored in knowledge bases. The number and the types of the knowledge bases to be used, as well as a strategy for transferring the knowledge generated to the various knowledge bases, have to be specified.

The knowledge stored in the knowledge bases can be used on two different occasions. Firstly, when a process has to take a decision, it can access the knowledge bases to obtain the knowledge to base the decision upon. Secondly, when a knowledge base is updated, i.e., when new knowledge generated by some process has been transferred to a knowledge base, a process can react to this update. Hence, strategies for accessing the knowledge bases, and for reacting to updates of the knowledge bases, have to be specified.

The work involved in the execution of a parallel branch and bound algorithm is performed by the various processes executing the algorithm. Without carrying out the actual computations, it is impossible to get an accurate estimate of this work. Therefore, the only way to achieve an equitable division of the work among the various processes is to divide the work as it is generated, i.e., dynamically during execution. To be able to divide the work, basic *units of work* have to be defined. An example of a unit of work is the branching from a single subproblem, including the computation of bounds to the subproblems thus created. The units of work still to be completed constitute knowledge about the problem instance to be solved, and are stored in the knowledge bases. Each time a process becomes idle, it accesses the knowledge bases to get new work, i.e., a unit of work is extracted from a knowledge base and given to the process to execute.

Finally, it has to be specified what happens when a process completes the unit of work it is currently working upon. There are two extremes: before starting with its next unit of work the process can wait for all other processes to complete their unit of work, or the process can start working upon a new unit of work immediately without waiting for fellow processes to complete their unit.

In the next sections, we will elaborate on these parameters. What constitutes good choices of the various parameter settings depends upon the characteristics of the problem to be solved, upon the four basic rules of the branch and bound algorithm, as well as upon the characteristics of the parallel machine to be used for executing the algorithm.

Some examples of particular parameter settings are presented in section 3.4.

3.2.1. Knowledge sharing

Knowledge bases can be classified according to the processes they are accessed by, and according to the knowledge stored in them.

A knowledge base can be accessed by all processes, or by a subset of the processes only. A *global knowledge base* is a knowledge base accessible by all processes. A *local knowledge base* can be accessed by a subset of the processes only. A *dedicated, local knowledge base* is a knowledge base that is accessed by a single process only, but updated by several processes.

All knowledge generated by the various processes, or only part of it, can be transferred to a given knowledge base. Note that part of the knowledge can mean all knowledge generated by part of the processes, as well as part of the knowledge generated by all the processes. A *complete knowledge base* is a knowledge base to which all knowledge generated by all processes is transferred. A *partial knowledge base* is a knowledge base to which only part of the knowledge generated is transferred. For example, a parallel branch and bound algorithm can use a partial knowledge base that contains only part of the active subproblems. Note that the completeness of a knowledge base as defined above applies to all knowledge generated. It is also possible to define completeness at a lower level, i.e., applied only to specific types of knowledge. For example, a knowledge base can contain all feasible solutions found.

If there are multiple knowledge bases, it is possible that specific knowledge is replicated in several knowledge bases, i.e., that the same ‘piece’ of knowledge is replicated, and subsequently transferred to several knowledge bases. The replication of specific types of knowledge can introduce difficulties while accessing this knowledge. Some types of knowledge, for example, the units of work still to be completed, ought to be guaranteed a mutually exclusive access. If knowledge is replicated, the only way to guarantee a mutually exclusive access to this particular knowledge is through communication.

The advantage of a global, complete knowledge base is that it contains a good approximation of the knowledge generated hitherto. This makes it easy to provide each process with an attractive unit of work (e.g., an attractive subproblem to branch from), and to prune the units of work still to be completed. However, the disadvantage is that operating upon the knowledge base tends to be a bottleneck because the knowledge base can only be operated upon by a single process at a time.

The advantage and disadvantage of local, partial knowledge bases are just the opposite. The bottleneck of all processes operating upon the same knowledge base is avoided, but some of the processes might be working on bad units of work (e.g., branching from subproblems with a high heuristic value), simply because there happened to be no attractive units stored in the knowledge base they accessed, or because the knowledge accumulated in this knowledge base did not show these units to be redundant with respect to the solution of the problem instance on hand.

The advantage of several local, complete knowledge bases is that each knowledge base contains a good approximation of the knowledge generated hitherto, and that the bottleneck of too many processes accessing a given knowledge base within a short period of time is reduced. However, keeping the contents of the local knowledge bases consistent, and securing mutually exclusive access to specific types of knowledge is difficult.

The advantage of a global, partial knowledge base is that it reduces the number of operations by the various processes (simply because there is less knowledge used by the algorithm). The disadvantage however is that valid knowledge about the problem instance to be solved is intentionally discarded by the algorithm.

All kinds of strategies for sharing the knowledge generated, i.e., for transferring the knowledge generated by the various processes to the various knowledge bases, are possible. We call these strategies *update strategies*. For example, the processes could transfer the knowledge immediately once generated (e.g., [Trienekens 1989]), or the processes could transfer new knowledge once every *n* time steps or units of work (e.g., [Clausen & Träff 1989]). Note that if the algorithm employs local knowledge bases, the update strategy also specifies which knowledge to transfer to which local knowledge base. Note next that knowledge that constitutes non-valid knowledge for all processes accessing a given knowledge base, as well as knowledge that can be proven to be redundant with respect to the knowledge already stored in the knowledge base, need not be transferred to a knowledge base.

The best possible knowledge sharing is obtained if each process transfers all knowledge it generates to all the relevant knowledge bases immediately once this knowledge has been generated. Such a sharing, however, increases the communication complexity of the parallel algorithm. It is clear, that there exists a tradeoff between the (possible) number of units of work eliminated and the amount of communication carried out.

Delaying the transfer of new knowledge to a knowledge base can have the following positive consequences. Firstly, the delay can result in the generation of additional knowledge in the mean time, which proves the former knowledge to be redundant. Secondly, it enables the concatenation of knowledge, such that a knowledge base need only be updated once instead of several times.

Knowledge bases can be implemented either as *active* or as *passive* entities. If a knowledge base is a passive entity, it just provides storage for knowledge. Manipulating this knowledge (e.g., updating and accessing the knowledge base and pruning the units of work stored in a knowledge base) is done by the processes themselves. If a knowledge base is an active entity, it is able not only to provide storage, but also to provide active services. For example, the knowledge base can return the best unit of work stored in it, or can prune the units of work stored in it. In essence, an active knowledge base is a process of a parallel branch and bound algorithm that is specialized in taking decisions based upon the knowledge stored in it.

3.2.2. Knowledge use

A process can exploit knowledge on two different occasions. Firstly, when the process has to take a decision, and secondly, when new knowledge generated by some other process becomes available.

When a process has to take a decision, it accesses the knowledge bases to obtain the knowledge to base this decision upon. The *access strategy* specifies which knowledge base(s) to access and when. Note that a process need not always access the same knowledge base(s).

To be able to use new knowledge generated by other processes, a process first has to detect the corresponding update of the knowledge base(s). The process of becoming aware of an update of a knowledge base, taking the new knowledge into account, and deciding upon how to continue, is called *reacting to new knowledge*. A reaction can have the effect that the unit of work currently being worked upon is preempted, or even eliminated. Hence, an early reaction tends to reduce the total amount of work involved in the execution of the parallel branch and bound algorithm. It is clear that there exists a tradeoff between the (possible) reduction in the work to be performed and the work involved in the reaction.

The *reaction strategy* specifies how a process becomes aware of an update of a knowledge base, and how the process reacts to such an update. There are two basic ways in which a process can become aware of an update: the process can *poll* the knowledge base, i.e., check on a regular basis whether the knowledge base has been updated, or the process can be *interrupted* whenever the knowledge base is updated, i.e., the process is notified of the update. A process that is interrupted immediately stops its current action, and starts handling the interrupt. Once the interrupt has been handled, the action is resumed (if the interrupt did not reset the program counter of the process). For more information on interrupts, we refer to [Tanenbaum

1984]. The two extremal reactions to an update of a knowledge base are an instantaneous reaction and a neglect of the update until the next decision has to be taken.

Applying dominance tests can be very hard if the units of work still to be completed are stored without replication in distinct partial knowledge bases. Because a subproblem can be dominated by an arbitrary other subproblem, the process pruning the units of work stored in a given knowledge base has to have access to all currently dominating subproblems, even those stored in knowledge bases not accessed by this process, and those already branched from or eliminated by bound tests. The only way to accomplish this is by replicating and transferring all subproblems generated to all the appropriate knowledge bases (i.e., all knowledge bases containing knowledge about the subproblems generated). This however severely increases the communication complexity of the algorithm.

Instead of applying dominance tests at a global level by checking all pairs of subproblems generated, these tests can also be applied at a local level by checking only pairs of subproblems stored in the same knowledge base. This way some of the advantages of dominance tests can be preserved without an increase in communication complexity.

3.2.3. Dividing the work

In principle, the *units of work* can be chosen arbitrarily. In real life however, the units of work tend to interact with the communication complexity of the resulting parallel branch and bound algorithm. If the units are too small, the communication network can become saturated. Examples of units of work are branching from a single subproblem (e.g., [Trienekens 1990]), solving a subproblem (e.g., [Jansen & Sijstermans 1989]), and computing a bound to a subproblem generated by decomposition (e.g., [Pekny & Miller 1989]).

Note that there is no need for a parallel branch and bound algorithm to use only a single unit of work. It is possible to use several diverse units of work concurrently. A process executing such an algorithm is either *specialized*, i.e., can operate only upon units of work of a given type, or *generalized*, i.e., can operate on units of several types. An example of a parallel branch and bound algorithm that uses two different units of work can be found in [Pekny & Miller 1989]. The two units are the computation of bounds, and the generation of new subproblems by branching (without computing bounds to the new subproblems). As long as there are subproblems to which a bound still has to be computed, the processes work upon these subproblems. As soon as there are no more such subproblems, an idle process branches from a subproblem (to which a bound has been computed), and generates new subproblems to which the bounds have to be computed.

A *single subproblem branch and bound algorithm* is a high level parallel branch and bound algorithm, whose unit of work is the branching from a single subproblem, including the computation of bounds to the subproblems thus created.

A *single subtree branch and bound algorithm* is a high level parallel branch and bound algorithm, whose unit of work is to determine the optimal solution of a single subproblem.

In essence, the units of work still to be completed constitute knowledge about the problem instance to be solved. Hence, dividing the work among the processes can be carried out via the knowledge bases. New work created by the processes is transferred to the knowledge bases, and each time a process becomes idle, it obtains new work out of a knowledge base. Preempted work, i.e., work that was being executed when a process reacted to an update of a knowledge base and subsequently decided to start working upon other work, is also transferred to the knowledge bases. Note that in order to prevent work being done twice, a mutually exclusive access to the units of work must be guaranteed.

An *active unit of work* is a unit of work that has been generated by some process, and whose execution is as yet not completed. Just as with active subproblems, the active units of work can be divided into two categories: those that are being worked upon at the moment, and those that are not being worked upon.

Just as a branch and bound algorithm has a selection rule to determine the next subproblem to branch from, there must be a selection rule to determine the next unit of work to start working upon. Again the framework of heuristic search can be used by defining a heuristic function upon the units of work: the unit of work chosen, is the unit of work with minimal heuristic value (cf. section 1).

A knowledge base that does not contain active units of work (anymore), is said to have *dried up*. If knowledge about the active units of work is stored without replication in distinct partial knowledge bases, all kinds of strategies can be applied to prevent these knowledge bases from drying up, or from containing no attractive units of work. We call these strategies *load balancing strategies*. The extremal strategies are to refill knowledge bases from other knowledge bases only when they have actually dried up (e.g., [Kindervater 1989]), or to continuously reshuffle units of work between the various knowledge bases (e.g., [Vornberger 1987]). Note that the load balancing strategy has consequences for the communication complexity of the resulting parallel branch and bound algorithm.

3.2.4. Synchronicity

The last parameter of high level parallel branch and bound algorithms to be specified is what happens when a process completes the unit of work it is working upon. Again there are two extremes: before accessing knowledge bases and starting with its next unit of work, the process can wait for all other processes to complete their unit of work, or the process can access knowledge bases and start working upon a new unit of work immediately without waiting for fellow processes to complete their unit.

Note that the fact that all processes are fully synchronized does not imply that at each point in time the processes are executing the same instruction.

Whether or not the processes perform their work in a synchronized way has consequences for the utilization of the various processes, as well as for the communication complexity of the parallel algorithm.

For synchronized processes, it holds that if the tasks to be performed are not of equal size, or if the processes are not of equal power, computing power will be lost while waiting for other processes to complete their task. In addition, synchronization involves communication because the processes must find out whether the other processes have yet completed their tasks or not. Therefore, synchronization tends to increase the communication complexity of the algorithm. An advantage of synchronization is that dividing the work among the various processes, and transferring the knowledge generated to the various knowledge bases, tend to be easy since these actions can be performed at the synchronization points when the knowledge is stable, i.e., during this work no new knowledge will be generated.

The characteristics of asynchronous processes are exactly the opposite.

Note that the synchronicity of the processes has consequences for the *determinism* of the resulting parallel branch and bound algorithm. If not all processes are fully synchronized, the transfer of the knowledge generated to the knowledge bases, and the becoming aware of and the subsequent use of new knowledge, can introduce a *nondeterministic behavior* in the resulting algorithm. This is due to the fact that during execution variations can occur with respect to the exact order in which the transfers, accesses, and prunings in the knowledge bases are carried out by the various processes, or with respect to the point in time a process will use, for example, a better feasible solution found. These variations are caused by environmental factors not under control of the algorithm, for example, other processes in the system, collisions on the network connecting the processing elements, or diverging clocks of the processing elements. Interchanging an update of a knowledge base by a process with an access of the same knowledge base by another process might cause the algorithm to follow a different path, resulting in a different solution. Even if two consecutive executions of the algorithm yield the same solution, the work carried out during these executions might be completely different. In [Trienekens 1990], we called this particular behavior a fluctuation anomaly, and described it in detail.

The nondeterministic behavior does not have consequences for the correctness of the parallel branch and bound algorithm. The correctness of an asynchronous parallel branch and bound algorithm can be proven using the same invariant as in the case of the sequential branch and bound algorithm: the value of the optimal solution to the original problem is either the value of the best solution found hitherto, or the value of the best solution to the remaining active subproblems. Therefore the solution yielded by an asynchronous parallel algorithm is always a correct one.

3.2.5. Summary

Figure 3.1 presents a summary of the parameters of parallel branch and bound algorithms.

Parameters of parallel branch and bound algorithms	
Knowledge sharing (among knowledge bases)	Number + type of knowledge bases
	Update strategy
Knowledge use (by processes)	Access strategy
	Reaction strategy
Work division	Processes
	Units of work
	Load balancing strategy
Synchronicity	Synchronicity of each process
The four basic rules	Branching rule
	Bounding rule
	Selection rule
	Elimination rule

Figure 3.1. Summary of parameters.

3.3. Concurrent Exhaustive Knowledge Handling

In this section we investigate how parallel branch and bound algorithms handle the knowledge generated during execution. We divide the parallel branch and bound algorithms into nine classes according to the way the processes executing the algorithm handle the knowledge generated, i.e., how they share and use the knowledge generated. This classification is not an additional taxonomy for parallel branch and bound algorithms, but describes properties that accompany specific settings of the parallel parameters described in the previous sections.

We generalize the notion of exhaustive knowledge handling defined earlier for sequential branch and bound algorithms (cf. section 2.4), and introduce the notion of mutual exhaustive knowledge handling.

As regards the way a particular process executing a parallel branch and bound algorithm handles the knowledge generated by itself, as well as by the other processes, we define the following two predicates:

$PRED_s$: All knowledge generated thus far by the process has been transferred to the knowledge bases.

$PRED_u$: The unit of work the process is executing is consistent with the knowledge the process has access to, i.e., the unit of work the process is performing is the most suitable unit of work the process is aware of, and this unit of work cannot be eliminated using the knowledge the process has access to.

Predicate $PRED_s$ states that the process has shared all the knowledge it has generated. As a consequence, all other processes can access (via the appropriate knowledge bases) all knowledge generated by this process. Predicate $PRED_u$ states that the process uses all knowledge stored in the knowledge bases it accesses.

A process has *complete knowledge sharing* at a certain point during execution if and only if $PRED_s$ is true, whereas a process has *complete knowledge use* at a certain point during execution if and only if predicate $PRED_u$ is true.

Let Q denote the set of processes used for executing a parallel branch and bound algorithm. Let for each process p ($p \in Q$) the following two statements be true for given invariance sets PCS_s^p and PCS_u^p:

(3.10) $invar(PRED_s,\ PCS_s^p)$,

(3.11) $\quad invar(PRED_u,\ PCS_u^p)$.

Note that the above two statements do not state anything about whether the process has complete knowledge sharing, or complete knowledge use, when the program counter does not belong to PCS_s^p, or PCS_u^p respectively. It is quite possible that the process has complete knowledge sharing, or complete knowledge use, when the program counter does not belong to this set; however, this particular kind of knowledge sharing, or use, cannot be guaranteed to occur.

We distinguish for either of the statements 3.10 and 3.11 three possible forms of the invariance set PCS_u^p, or PCS_s^p respectively: PCS^*, PCS ($PCS^* \neq PCS \neq \emptyset$), or $\emptyset$.

If the invariance set equals PCS^*, the process has *permanent complete knowledge sharing*, or *permanent complete knowledge use*. Note that permanent complete knowledge sharing is only possible if the process transfers new knowledge to the knowledge bases instantaneously once generated. Note next that if the processes are executing independently of each other, permanent complete knowledge use is only possible if a process is notified of every update of the knowledge bases it accesses.

If the invariance set equals PCS ($PCS^* \neq PCS \neq \emptyset$), the process has *delayed complete knowledge sharing*, or *delayed complete knowledge use*. The process intentionally delays the sharing, or using, of new knowledge, i.e., it takes a while before the process transfers new knowledge to the various knowledge bases, or before new knowledge stored in the knowledge bases is actually used by the process.

If the invariance set equals $\emptyset$, the process has *no complete knowledge sharing*, or *no complete knowledge use*. It can never be guaranteed that all knowledge generated by the process is transferred to the knowledge bases, or that the process uses all knowledge stored in the knowledge bases it accesses respectively. Note that occasionally it can happen that the process has complete knowledge sharing, or complete knowledge use; however, such knowledge sharing, or use, cannot be guaranteed to occur.

Parallel branch and bound algorithms can be classified according to the way the processes executing the algorithm share and use the knowledge they generate. In this classification it is assumed that all processes behave similarly, i.e., they all share, and use, knowledge in the same way.

Using the above described characteristics, nine distinct classes of parallel branch and bound algorithms can be distinguished. The nine classes are shown in figure 3.2. The name of each class consists of two characters. The first character denotes the kind of knowledge sharing exerted by the processes executing the algorithm, whereas the second character denotes the kind of knowledge use.

complete knowledge sharing	complete knowledge use		
	permanent	delayed	no
permanent	*PP*	*PD*	*PN*
delayed	*DP*	*DD*	*DN*
no	*NP*	*ND*	*NN*

Figure 3.2. Classes of parallel branch and bound algorithms.

Now for the generalization of the notion of exhaustive knowledge handling. As defined in section 2.4, the process executing a sequential branch and bound algorithm has exhaustive knowledge handling for a given invariance set *PCS* if and only if the following statement is true:

(3.12) $\quad invar(EXH,\ PCS)$.

Exhaustive knowledge handling for a process executing a parallel branch and bound algorithm is defined analogously.

Firstly, the predicates $PRED_1$, $PRED_2$, and $PRED_3$ must be appropriately redefined to take care of multiple processes generating knowledge, and of multiple knowledge bases.

$PRED_1$: The knowledge base(s) the process accesses contain(s) all valid knowledge generated thus far with respect to the elimination rules.

$PRED_2$: All active units of work that are currently not being worked upon are stored in the knowledge bases.

$PRED_3$: $PRED_u$

Just as the formulation of their sequential counterparts, the formulation of the concurrent predicates is not free from some ambiguity. Again, given the precise description of the algorithm, the concurrent predicates can be formulated in a straightforward way.

Note the relation between the predicates $PRED_1$ and $PRED_2$ on the one hand, and the predicate $PRED_s$ on the other hand. ($PRED_1$ **and** $PRED_2$) can only be true if $PRED_s$ is true for all processes executing the algorithm. Next to that, the validity of ($PRED_1$ **and** $PRED_2$) implies a special way of complete knowledge sharing by the processes: a process has access to all the nonredundant knowledge generated as regards elimination rules, but does not need to have access to all the units of work still to be completed to solve the problem instance on hand.

The definition of the predicate *EXH*, which states exhaustive knowledge handling for a given process, remains as it was, i.e.,

(3.13) $EXH = (PRED_1 \textbf{ and } PRED_2 \textbf{ and } PRED_3)$.

Note that this predicate does not state that the process is executing the most suitable active unit of work, but the most suitable active unit of work the process is aware of.

Finally, let *m* denote the cardinality of the set *Q* (the set of processes executing the parallel algorithm), let *q* denote a particular process the parallel algorithm is executed by ($q \in Q$), let $\overline{EXH}^q$ denote the local predicate that states that process *q* has exhaustive knowledge handling, i.e.,

(3.14) $\overline{EXH}^q = \langle pred_1^q, pred_2^q, \ldots, pred_m^q \rangle$,

$pred_q^q = EXH$,

$pred_p^q = \textbf{true} \quad (p \in Q, p \neq q)$,

and let $\overline{PCS}^q$ be a global program counter set, i.e.,

(3.15) $\overline{PCS}^q = \langle PCS_1^q, PCS_2^q, \ldots, PCS_m^q \rangle$.

Process *q* has exhaustive knowledge handling for a given invariance set $\overline{PCS}^q$ if and only if the following global statement is true:

(3.16) $invar(\overline{EXH}^q, \overline{PCS}^q)$.

A parallel branch and bound algorithm has *exhaustive knowledge handling* if and only if each process executing the algorithm has exhaustive knowledge handling for a given, process dependent, program counter set $\overline{PCS}^p$, i.e., if the following *m* statements are true:

(3.17) $invar(\overline{EXH}^p, \overline{PCS}^p) \quad (p \in Q)$.

Note that because $\overline{PCS}^p$ need not be equal to $\overline{PCS}^q$ ($p \neq q$, $p \in Q$, $q \in Q$), it does not follow from the above statements that whenever it can be guaranteed that process *p* has exhaustive knowledge handling, process *q* has exhaustive knowledge handling as well.

A parallel branch and bound algorithm has *mutual exhaustive knowledge handling* for a given program counter set $\overline{PCS}$ if and only if all processes executing the algorithm have exhaustive knowledge handling for the same program counter set $\overline{PCS}$, i.e., if the following statement is true:

(3.18) $invar(\overline{EXH}, \overline{PCS})$.

Note that given the invariance sets $\overline{PCS}^q$ ($p \in Q$) mentioned in formula 3.17, the invariance sets $\overline{PCS}$ mentioned in formula 3.18 can be constructed as follows:

(3.19) $\overline{PCS} = \bigcap_{p \in Q} \overline{PCS}^p$.

Remember that if the invariance set $\overline{PCS}$ in formula 3.18 is not equal to $\overline{PCS*}$, and if the algorithm is completely asynchronous, it cannot be guaranteed that invariance points are actually reached during execution.

If care is exercised about the knowledge base(s) to which the knowledge generated is transferred, about how a process reacts to an update of a knowledge base, and about which knowledge base(s) a process accesses, the knowledge handling classification as presented in figure 3.2 says something about exhaustive knowledge handling. In this case, the update, reaction, and access strategies must be such that the validity of the predicates $PRED_1$, $PRED_2$, and $PRED_3$ can be guaranteed for specific invariance sets.

For *PP* algorithms, it holds that the processes executing these algorithms have permanent complete knowledge sharing, as well as permanent complete knowledge use. As a consequence, each process has permanent exhaustive knowledge handling, and hence, the algorithm has permanent mutual exhaustive knowledge handling.

For *DD* algorithms, it holds that the processes executing these algorithms have delayed complete knowledge sharing, as well as delayed complete knowledge use. A process can only have exhaustive knowledge handling if the process itself has complete knowledge use, whereas at the same time all processes are guaranteed to have complete knowledge sharing. Note that such knowledge handling can only occur at discrete points in the parallel program. Mutual exhaustive knowledge handling can be guaranteed if the intersection of the various invariance sets of complete knowledge sharing and the various invariance sets of complete knowledge use, is not empty (cf. formula 3.19).

For *PD* and *DP* algorithms, it holds that the processes executing these algorithms have permanent complete knowledge sharing, or permanent complete knowledge use respectively. As a consequence, a process executing a *PD* algorithm has exhaustive knowledge handling whenever it has complete knowledge use. However, in order for a process executing a *DP* algorithm to have exhaustive knowledge handling, all other processes must have complete knowledge sharing at the same time. As regards mutual exhaustive knowledge handling, these algorithms are similar to *DD* algorithms.

For all other algorithms, i.e., algorithms in the classes *PN*, *NP*, *DN*, *ND*, or *NN*, it cannot be guaranteed that the processes executing these algorithms have complete knowledge sharing, or complete knowledge use respectively. Hence, exhaustive knowledge handling, and therefore mutual exhaustive knowledge handling, cannot be guaranteed to occur.

Note that *PP* algorithms exhibit the strongest form of exhaustive knowledge handling, whereas *DD* algorithms exhibit the weakest form. Remember that the degree of exhaustive knowledge handling does not state anything about the performance of the algorithm (cf. section 2.4).

3.4. Examples of Parallel Branch and Bound Algorithms

In this section we will demonstrate the versatility of our taxonomy by classifying some of the parallel branch and bound algorithms described in the literature. For a complete description of the algorithms we refer to the original papers. Note that the examples are concerned mainly with the four parameters related with the parallelism; they hardly touch upon the four basic rules presented in section 1.

At the end of this section, we will use our taxonomy to analyze the similarities and differences between these algorithms.

The majority of the algorithms described uses a fixed number of processes, which we will denote by *m*. In principle, *m* is an arbitrary number; in real life however, *m* tends to be equal to the number of processing elements available.

3.4.1. Li & Wah [1984]

Li and Wah describe a synchronous, single subproblem, *DD* branch and bound algorithm that uses a single global, complete knowledge base. The particular parameter settings are presented in figure 3.3.

3.4.2. Clausen & Träff [1989]

Clausen and Träff developed an asynchronous, single subproblem, *PD* branch and bound algorithm for execution on a 32 node hypercube. The particular parameter settings are presented in figure 3.4.

Knowledge sharing	●Global, complete knowledge base.
	●Delay the transfer of knowledge until the synchronization. The order in which the processes transfer knowledge to the knowledge base is fixed.
Knowledge use	●Access global knowledge base after synchronization. The order in which the processes access the knowledge base is fixed.
	●Do not react to update of knowledge base.
Work division	●*m* processes.
	●Branching from single subproblem.
	●No load balancing (needed).
Synchronicity	●Synchronize after each branching.

Figure 3.3. Li & Wah [1984].

Knowledge sharing	●Each process has its own dedicated, local, partial knowledge base, that contains active subproblems and the current best solution.
	●Transfer knowledge about better solutions found immediately to all knowledge bases. ●Transfer knowledge about subproblems generated immediately to dedicated knowledge base.
Knowledge use	●Access dedicated local knowledge base.
	●Do not react to update of knowledge base.
Work division	●*m* processes.
	●Branching from single subproblem.
	●Send on a regular basis subproblems to neighboring knowledge bases if dedicated knowledge base contains more than a given minimum number of subproblems.
Synchronicity	●Completely asynchronous.

Figure 3.4. Clausen & Träff [1989].

3.4.3. Vornberger [1987]

Vornberger developed an asynchronous, single subproblem, *PD* branch and bound algorithm for execution on a parallel machine consisting of 32 transputers [INMOS 1986]. The particular parameter settings are presented in figure 3.5.

Because communication between transputers is very cheap (i.e., does not involve much effort or time), the processes executing the algorithm communicate extensively. The algorithm emphasizes the fact that all knowledge bases should contain attractive subproblems to branch from and employs a special load balancing strategy to encourage this.

3.4.4. Lavallée & Roucairol [1985]

Lavallée and Roucairol developed an asynchronous, single subproblem, *PD* branch and bound algorithm, which they called ‘parallélisation verticale’. The particular parameter settings are presented in figure 3.6.

Lavallée and Roucairol claim that the unit of work used by their algorithm is the branching from a complete subtree, and hence, that their algorithm is a single subtree algorithm. Making this claim, they neglect the consequences of their load balancing strategy: their unit of work turns out to be not atomic, and can be split into smaller units, i.e., the branching from single subproblems. The smaller units can be

Knowledge sharing	•Each process has its own dedicated, local, partial knowledge base, that contains active subproblems and the current best solution.
	•Transfer knowledge about better solutions found immediately to all neighboring knowledge bases. •Transfer knowledge about subproblems generated immediately to dedicated knowledge base.
Knowledge use	•Access dedicated knowledge base.
	•Do not react to update of knowledge base.
Work division	•*m* processes.
	•Branching from single subproblem.
	•Send on a regular basis subproblems to neighboring processes. The frequency with which to send to a given neighbor depends upon the quality of the subproblems recently received from this neighbor.
Synchronicity	•Completely asynchronous.

Figure 3.5. Vornberger [1987].

transferred to other knowledge bases. If this happens, it is very likely that a process will never know the optimal solution to its subtree. Because the load balancing strategy is an essential part of the algorithm, the algorithm must be classified as a single subproblem algorithm.

Knowledge sharing	•Each process has its own dedicated, local, partial knowledge base, that contains active subproblems and the current best solution.
	•Transfer knowledge about better solutions found immediately to all knowledge bases. •Transfer knowledge about subproblems generated immediately to dedicated knowledge base.
Knowledge use	•Access dedicated local knowledge base.
	•Do not react to update of knowledge base.
Work division	•*m* processes.
	•Branching from single subproblem.
	•Request new subproblems from the other knowledge bases whenever dedicated knowledge base has dried up.
Synchronicity	•Completely asynchronous.

Figure 3.6. Lavallée & Roucairol [1985].

3.4.5. Kindervater [1989]

Kindervater developed an asynchronous, single subproblem, *PN* branch and bound algorithm. The algorithm is similar to the ‘parallélisation verticale’ algorithm of Lavallée and Roucairol. It mainly differs from this algorithm in the way the work is redivided among the various local knowledge bases. The particular parameter settings are presented in figure 3.7.

The design of Kindervater’s algorithm was heavily influenced by the rigid communication primitives of the IBM-LCAP, the parallel machine used for executing the algorithm. A description of this machine can be found in [Di Chio & Zecca 1985]. On this machine, a sending process synchronizes with the receiving process, i.e., a sending process cannot continue until the receiving process has received the message. Fortunately, a process can detect whether or not another process wants to communicate.

Kindervater implemented the global knowledge base as an active knowledge base. Because the sending process synchronizes with the receiving process, the active, global, knowledge base cannot send a message to a local knowledge base without synchronizing with this local knowledge base, and hence becoming inaccessible for all others. Therefore the processes have to poll the global knowledge base.

Knowledge sharing	●Global, partial, knowledge base, containing active subproblems and the current best solution. ●Each process has its own dedicated, local, partial knowledge base, that contains active subproblems and the current best solution.
	●Transfer knowledge about better solutions found immediately to global knowledge base. ●Transfer knowledge about subproblems generated immediately to dedicated knowledge base.
Knowledge use	●Access dedicated local knowledge base.
	●Poll global knowledge base on regular basis. Update dedicated knowledge base if global knowledge base has been updated.
Work division	●*m* processes.
	●Branching from single subproblem.
	●Get new subproblems from global knowledge base when dedicated knowledge base has dried up. If global knowledge base has dried up, try to refill this base from the other local knowledge bases.
Synchronicity	●Completely asynchronous.

Figure 3.7. Kindervater [1989].

3.4.6. Trienekens [1990]

The first author experimented with an asynchronous, single subproblem, *PD* branch and bound algorithm that uses a single global, complete knowledge base. The particular parameter settings are presented in figure 3.8.

Knowledge sharing	●Global, complete knowledge base.
	●Transfer knowledge immediately to global knowledge base.
Knowledge use	●Access global knowledge base.
	●Do not react to update of knowledge base.
Work division	●*m* processes.
	●Branching from single subproblem.
	●No load balancing (needed).
Synchronicity	●Completely asynchronous.

Figure 3.8. Trienekens [1989].

3.4.7. Jansen & Sijstermans [1989]

Jansen and Sijstermans developed an asynchronous, single subtree, *PD* branch and bound algorithm that uses a global, complete knowledge base. The algorithm employs a variable number of identical processes which each examine a subtree of the problem tree. The particular parameter settings are presented in figure 3.9.

While examining a given subtree, a process can decide to create an additional process to perform part of this examination. The newly created process performs its work independently of the creating process. A process quits once it has completed its examination and has returned the results to its creator. A parent process cannot complete its examination unless its children have completed their examinations and returned the results. The number of processes that can coexist, and hence the decision whether or not to create a new process, depends upon the characteristics of the parallel machine used, the characteristics of the algorithm, as well as upon the particular problem instance to be solved.

Knowledge sharing	•Global, complete knowledge base containing the current best solution and the current number of processes.
	•Transfer knowledge immediately to global knowledge base.
Knowledge use	•Access global knowledge base.
	•Do not react to update of knowledge base.
Work division	•Variable number of processes. A process can create another process to perform part of its work.
	•Solution of single subproblem.
	•No load balancing (needed).
Synchronicity	•Completely asynchronous.

Figure 3.9. Jansen & Sijstermans [1989].

3.4.8. Pekny & Miller [1989]

Pekny and Miller developed an asynchronous parallel, *PD* branch and bound algorithm, which they called a ‘processor shop model’. The algorithm uses two distinct units of work, as well as three distinct global knowledge bases. The particular parameter settings are presented in figure 3.10.

For each subproblem extracted from the bound knowledge base, a bound is computed. Then the subproblem is added to the branching knowledge base, unless the subproblem is solved to optimality during the bound computations, in which case it is checked whether the upper bound must be updated. A subproblem extracted from the branching knowledge base is decomposed into smaller subproblems. The subproblems thus created are added to the bound knowledge base.

An idle process tries to extract a subproblem from the bound knowledge base. If this knowledge base has dried up, the process tries to extract a subproblem from the branching knowledge base. If both knowledge bases have dried up, the process waits until work becomes available in either one of the knowledge bases.

The basic idea behind the algorithm is a low level parallelization of the branching from a single subproblem. If there are more processes than the number of processes required for this low level parallelization, additional subproblems can be branched from in parallel.

3.4.9. A comparison of branch and bound algorithms.

In this section we use our taxonomy for comparing the various parallel branch and bound algorithms presented in the previous sections with each other.

All the algorithms presented, except the algorithm described by Li & Wah, are asynchronous parallel branch and bound algorithms. For these algorithms, it holds that the interactions between the various processes are not completely defined. Hence, all algorithms, except the one by Li & Wah, can show a nondeterministic behavior during execution.

All the algorithms presented, except the algorithm by Kindervater, are either of the *PD* type or of the *DD* type. The processes executing these algorithms completely neglect updates of knowledge bases, i.e., new knowledge, until they have to take their next decision. The algorithm by Kindervater checks on a regular basis, but with a frequency which is smaller than the frequency with which decisions are taken,

Knowledge sharing	●Global, complete knowledge base containing subproblems to decompose. ●Global, complete knowledge base containing subproblems to compute a bound to. ●Global, complete knowledge base containing the current best solution.
	●Transfer knowledge immediately to appropriate global knowledge base.
Knowledge use	●Access appropriate global knowledge base.
	●Do not react to update of knowledge bases.
Work division	●*m* processes.
	●Generation of the children of a subproblem. ●Computation of bound to a subproblem.
	●No load balancing (needed).
Synchronicity	●Completely asynchronous.

Figure 3.10. Pekny & Miller [1989].

whether the global knowledge base has been updated.

A mechanism that lets the processes react to an update of the knowledge base(s) can be added in a straightforward way to most algorithms. However, implementing the mechanism on a real parallel machine can be quite difficult.

For all algorithms presented, except the algorithm by Li & Wah, it cannot be guaranteed that the invariance points of mutual exhaustive knowledge handling are actually reached during execution. However, for all algorithms presented, except the algorithm by Kindervater, it can be guaranteed that the invariance points of exhaustive knowledge handling for the various processes are reached during execution.

All algorithms presented, except the algorithm by Pekny & Miller, use a single unit of work. All units of work, except the unit used by Jansen & Sijstermans, are atomic, i.e., cannot be split into smaller units.

All algorithms presented, except the algorithm by Jansen & Sijstermans, use a fixed number of processes.

The algorithms by Li & Wah, Trienekens, and Pekny & Miller are the only ones that use a global, complete knowledge base containing all subproblems generated. Hence, these algorithms can handle elimination rules involving dominance tests on a global level, i.e., between all currently dominating subproblems. All other algorithms employ partial knowledge bases as regards the subproblems generated, and hence no process has a total view of all the subproblems generated. Therefore these algorithms can only handle dominance tests on a local level.

The algorithms by Clausen & Träff, Vornberger, Lavallée & Roucairol, and Kindervater are very much alike. Their main difference lies in the way they divide the work among the various processes.

Clausen & Träff, and Vornberger try to prevent knowledge bases from drying up by periodically exchanging active subproblems. But, once a knowledge base dries up, this knowledge base has to wait until some other process has the courtesy to send some active subproblems. The difference between these two algorithm lies in the fact that Clausen & Träff exchange subproblems independently of the subproblems stored in the receiver's knowledge base, whereas Vornberger's exchange is governed by the quality of the subproblems currently stored in the receiver's knowledge base.

Lavallée & Roucairol, and Kindervater let a process whose dedicated knowledge base has dried up request new subproblems. The difference between these two algorithms lies in the fact that Lavallée & Roucairol just broadcast a request to all other processes, whereas Kindervater accesses the global knowledge base. Therefore a process executing the former algorithm will receive multiple reactions to its request, whereas a process of the latter algorithm will receive a single reaction.

The fact that the execution of the algorithm is completed, i.e., that the problem instance on hand has been solved, can be straightforwardly detected for all algorithms, except the algorithms by Clausen & Träff, Vornberger, and Lavallée & Roucairol. For the last three algorithms it holds that, due to the load balancing strategy, active subproblems can be floating between the various knowledge bases, without the knowledge bases being aware of them. In order to conclude that the execution has completed, it has to be checked that there are no floating subproblems.

4. CONCLUSIONS AND ONGOING RESEARCH

The taxonomy presented clearly shows how the essential parts of the parallel branch and bound algorithm cooperate in solving the problem instance at hand, and hence, allows for a better understanding of the working of parallel branch and bound algorithms.

However, we believe that the taxonomy presented is governed too much by the rigid frame imposed upon it by the generally accepted description of the sequential branch and bound algorithm, as presented in, for example, [Mitten 1970] and [Ibaraki 1976a, 1976b, 1977a, 1977b]. This description specifies an order in which the various rules and tests must be applied, and hence, contains an implicit way in which the knowledge generated, is to be used. A branch and bound algorithm should be free to determine itself when and in which order to use the various rules and test. Currently we are working upon a description of the branch and bound algorithm, in which these ideas are attained.

Acknowledgments

This paper is based upon Chapter 4 of the first author's PhD dissertation. The authors would like to thank Gerard Kindervater, Alexander Rinnooy Kan, and Bobby Schnabel for their help during preparation of this dissertation.

References

J. Clausen, J.L. Träff (1989). *Parallel Graph Partitioning using Branch-and-Bound with Dynamic Distribution of Subproblems*. Dept. of Computer Science, University of Copenhagen.

P. Di Chio, V. Zecca (1985). *IBM ECSEC Facilities: User's Guide*, Report G513-4080, IBM European Center for Scientific and Engineering Computing, Rome.

M. Grötschel (1980). On the symmetric travelling salesman problem: Solution of a 120-city problem. *Mathematical Programming Study, No. 12,* 61-77.

T. Ibaraki (1976a). Computational efficiency of approximate branch-and-bound algorithms. *Mathematics of Operations Research, Vol. 1, No. 3*.

T. Ibaraki (1976b). Theoretical comparisons of search strategies in branch-and-bound algorithms. *Int'l Jr. of Comp. and Info. Sci., Vol. 5, No. 4*.

T. Ibaraki (1977a). The power of dominance relations in branch-and-bound algorithms. *Journal of the ACM, Vol. 24, No. 2*.

T. Ibaraki (1977b). On the computational efficiency of branch-and-bound algorithms. *Journal of the Operations Research Society of Japan, Vol. 20, No. 1*.

T. Ibaraki, Y. Katoh (1991). Searching minimax game trees under memory space constraint. *Annals of Mathematics and Artificial Intelligence, Vol. 1*.

INMOS LTD (1986). *The Transputer Family - Product Information*.

J.M. Jansen, F.W. Sijstermans (1989). Parallel branch-and-bound algorithms. *Future Generation Computer Systems, Vol. 4,* 271-279.

R. Jonker, T. Volgenant (1982). A branch and bound algorithm for the symmetric travelling salesman problem based on the 1-tree relaxation. *European Journal of Operational Research, Vol. 12*, 394-403

G.A.P. Kindervater (1989). *Exercises in Parallel Combinatorial Computing.* PhD thesis, Centre for Mathematics and Computer Science, Amsterdam.

I. Lavallée, C. Roucairol (1985). *Parallel Branch and Bound Algorithms.* Technical report MASI nr. 112, Université Paris VI, Paris.

G. Li, B.W. Wah (1984). *Computational Efficiency of Parallel Approximate Branch-and-Bound Algorithms*, Report TR_EE 84-6, School of Electrical Engineering, Purdue University.

L.G. Mitten (1970). Branch-and-bound Methods: General formulation and properties. *Operations Research 18*, 24-34.

J.F. Pekny, D.L. Miller (1989). Results from a parallel branch and bound algorithm for the asymmetric traveling salesman problem. *OR Letters 8*, 129-135.

J.C. Reynolds (1981). *The Craft of Programming*. Prentice Hall Int.

J.F. Shapiro (1979). A survey of Lagrangean techniques for discrete optimization. *Annals of Discrete Mathematics 5*, 113-138.

A.S. Tanenbaum (1984). *Structured Computer Organization.* Prentice-Hall, Inc., Englewood Cliffs, NJ.

H.W.J.M. Trienekens (1989). *Computational Experiments with an Asynchronous Parallel Branch and Bound Algorithm.* Report EUR-CS-89-02, Dept. of Computer Science, Erasmus University, Rotterdam.

H.W.J.M. Trienekens (1990). *Parallel Branch and Bound Algorithms.* PhD thesis, Dept. of Computer Science, Erasmus University, Rotterdam.

O. Vornberger (1987). Load balancing in a network of transputers. *Proc. of the 2nd International Workshop on Distributed Algorithms,* Lecture Notes in Computer Science Vol. 312, Springer Verlag.