Parallelization of Branch-and-Bound Algorithms in a Functional Programming Environment

J.C. Glas

R.F.H. Hofman W.G. Vree

Department of Computer Systems, University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

Abstract

The referential transparancy of functional languages imposes restraints on the communication between parallel tasks. This seriously hampers the implementation of parallel branch-and-bound algorithms, because the parallel tasks cannot update a shared bound asynchronously. Maintaining local bounds only causes enormous performance loss, because parallel branch-and-bound algorithms exploit speculative parallelism. Therefore, the parallel tasks have to exchange their bounds synchronously, but this lowers the average processor utilization. Three parallel functional implementations of a generally applicable branch-and-bound algorithm have been developed to investigate this trade-off. Experiments have been performed with the cargo loading problem. Two functional equivalents of imperative parallel branch-and-bound approaches are outperformed by a dedicated parallel functional branch-and-bound implementation. A new speculative parallel annotation is proposed to tackle the trade-off between speculative transformation loss and average processor utilization on system level.

1 Introduction

Functional programming languages are based on the computational model of the lambda calculus. The most important property of functional languages is referential transparancy. It guarantees that the same function applied to the same arguments returns the same result in any context. Side effects like the assignment of global variables in imperative languages do not exist in functional languages. The simple semantics of functional languages facilitate proving the correctness of sequential programs, after which strictness analysis suffices to generate a correct parallel equivalent. Another advantage of functional languages is their support for modular program design by means of higher-order functions and lazy evaluation [Hug89].

Functional languages are extended with parallel annotations, but referential transparancy imposes restraints on the underlying parallel model. The applied fork-join model consists of a hierarchy of parallelizable jobs, which only permits direct communication along child-parent relations. Interaction between arbitrary child jobs involves a high degree of synchronization, because the indirect communication has to be performed by a common ancestor job, which requires termination and regeneration of all its descendant jobs. Parallel divide-and-conquer applications only commit child-parent communication, which makes them pre-eminently suitable for implementation in this parallel model.

Branch-and-bound algorithms are related to divide-and-conquer applications, because they repeatedly partition the search space of an optimization problem into smaller and smaller subspaces. However, they have to maintain a best solution found so far to exclude inferior subspaces from further partitioning. This requires interaction between the parallel jobs of a parallel branch-and-bound algorithm, because ignorance about the best solution found so far causes speculative transformation loss as a result of the partitioning of too many inferior subspaces. An imperative parallel branch-and-bound algorithm can either give parallel jobs asynchronous access to shared memory for reading and updating the best solution found so far or broadcast new best solutions to all of them. Those asynchronous communication methods contribute to the generally good parallel performance of those algorithms. Also divide-and-conquer skeletons can be extended with asynchronous communication methods to support parallel branch-and-bound algorithms [MBS87], but this violation of the referential transparancy encounters opposition in functional communities.

A functional parallel branch-and-bound algorithm cannot use asynchronous communication methods. It has to commit indirect communication between the parallelizable jobs, which causes the average processor utilization to decrease as a result of inherent task shortage and load imbalance. This paper studies the communicative restrictions of parallel functional programming environments with regard to the parallelization of branch-and-bound algorithms. An existing parallel annotation has been applied to develop a dedicated parallel functional branch-and-bound implementation, whose indirect communication pattern optimizes the trade-off between speculative transformation loss and average processor utilization. Furthermore, a new parallel annotation is proposed to tackle this trade-off on system level.

2 A parallel functional programming environment

The HyperM-project (Hybrid parallel experimental reduction Machine) is engaged with the development of a parallel functional programming environment from application layer to hardware layer. The environment provides a lazy functional programming language with divide-and-conquer skeleton [Har88], [Vre89]. Sequential functional programs are extended with annotations to generate parallel equivalents. The implementation of the functional language is realized with graph reduction [Pey87]. Referential transparancy imposes restraints on the underlying parallel model, which is formed by a hierarchy of graph reducing processes and can be denoted as fork-join model. The evaluation of the main expression of a parallel functional program is started by a main job. Every parent job in the hierarchy forks into a number of parallelizable child jobs by applying a parallel annotation. Direct communication can only be committed along child-parent relations, which makes the parallel functional programming environment pre-eminently suitable for divide-and-conquer applications.

The hierarchy of graph reducing processes can be mapped on any MIMD-architecture (Multiple Instruction – Multiple Data). The system layer has to apply a run-time scheduling algorithm to distribute the generated jobs over the available processors. The parallel functional programs can be executed by a sequential graph reducer, which produces a job graph of all generated jobs. The job graph is used to analyse the performance of a parallel functional program on an arbitrary parallel architecture with uniform processors.

2.1 The "sandwich"-annotation

The "sandwich"-function is a parallel annotation, which is executed by a parent job to generate a number of child jobs. The ethymology refers to the change in reduction strategy by inserting a layer of parallel eager evaluation (call by value) between two layers of sequential lazy evaluation (call by need). A formal definition of the "sandwich"-function is given by the following term rewrite rule:

$$\begin{aligned} &\texttt{sandwich}\ \mathcal{F}\ x_1 \ldots x_m \\ &\quad = \mathcal{F}\ x_1 \ldots x_m \\ &\qquad \textbf{where} \\ &\qquad\quad x_i = \mathcal{G}_i\ y_{i,1} \ldots y_{i,n_i} \end{aligned}$$

The result of the "sandwich"-function is equal to the result of function $\mathcal{F}$ applied to its arguments x_1 to x_m, but the evaluation is performed with an alternative reduction strategy. The parent job is suspended and m child jobs are generated to evaluate the redexes x_1 to x_m in parallel. After all child jobs have terminated, the awakened parent job applies function $\mathcal{F}$ to join the returned normal forms. The grain size of the redexes x_1 to x_m has to be coarse enough to justify parallel evaluation and function $\mathcal{F}$ has to be strict in all its arguments. Graph reduction allows the redexes x_1 to x_m to share inner redexes. The "sandwich"-function avoids multiplication of work by having the parent job evaluate every function $\mathcal{G}_i$ and all arguments $y_{i,1}$ to y_{i,n_i}, before generating the child jobs x_1 to x_m. Every function $\mathcal{G}_i$ has to be strict in all its arguments.

2.2 Analysis of parallel performance

The performance of a parallel implementation on a parallel architecture with n processors can be expressed by the speedup $S(n)$, This speedup is calculated with regard to the execution of the sequential implementation on one processor of the same type. The job graph of the parallel implementation enables the derivation of simple speedup bounds for any parallel system with a work-conserving scheduling algorithm [EZL89]. The influence of system and communication overhead on the speedup is considered negligible for simplicity. The basis for the speedup calculations is formed by the average parallelism A in the job graph, which is calculated as follows:

$$A = \frac{R_{total}}{R_{max}}$$

R_{total} is the total number of executed reduction steps in the job graph and R_{max} is the number of reduction steps executed along the longest path from the left to the right of the job graph. The fraction of inherently sequential work in the job graph is denoted by the fraction f, which is calculated as follows:

$$f = \frac{R_{main}}{R_{total}}$$

R_{main} is the number of reduction steps executed by the inherently sequential main job.

The efficiency $E(n)$ denotes the average utilization degree of a processor. Temporary idleness of processors is caused by load imbalance or inherent task shortage in specific job graph sections. The lower bound $E_l(n)$, the upper bound $E_u(n)$ and the estimated efficiency $E_e(n)$ are calculated as follows:

$$\begin{aligned} E_l(n) &= \frac{A}{A+n-1} \\ E_u(n) &= \min\left(\frac{A}{n}, \frac{1}{1+f\cdot(n-1)}\right) \\ E_e(n) &= \frac{2\cdot E_l(n)\cdot E_u(n)}{E_l(n)+E_u(n)} \end{aligned}$$

If the number of runnable jobs in the job graph never exceeds the number of available processors, then the efficiency $E(n)$ is equal to its upper bound $E_u(n)$. The relative error of the estimated efficiency $E_e(n)$ is guaranteed to stay below one third for any work-conserving scheduling algorithm. The lower bound $S_l(n)$, the upper bound $S_u(n)$ and the estimated speedup $S_e(n)$ arise from the following equation:

$$S_\xi(n) = TL \cdot E_\xi(n) \cdot n$$

The fraction TL denotes the transformation loss, expressing the possible increase of the total number of executed reduction steps, which results from transforming the sequential implementation into its parallel equivalent. The transformation loss fraction TL is calculated as follows:

$$TL = \frac{R_{seq}}{R_{total}} = CTL \cdot STL$$

R_{seq} is the number of reduction steps executed by the sequential implementation. The transformation loss is factorized into a conservative and a speculative component. Conservative transformation loss results from extra program code required to generate enough coarse-granular parallel jobs. If parallel programs do not contain enough conservative parallelism, speculative parallel jobs can be generated to put idle processors to work. Speculative transformation loss results from the wasted reduction steps of unnecessarily executed speculative parallel jobs. The profit of speculative parallelism and the methods to calculate speculative transformation loss are highly dependent on the application. The speculative transformation loss fraction STL of a parallel branch-and-bound implementation can be calculated as follows:

$$STL = \frac{P_{seq}}{P_{par}}$$

P_{seq} and P_{par} are the number of subspace partitions executed by respectively the sequential and parallel branch-and-bound implementations. The conservative transformation loss fraction CTL is derived by calculating TL/STL.

3 Branch-and-bound algorithms

Many optimization problems belong to the class of NP-complete problems [GaJ79]. Examples of NP-complete optimization problems are integer programming, graph colouring, graph partitioning, optimal scheduling, the traveling salesman problem and the knapsack problem. The computation time to solve NP-complete problems increases exponentially with the problem size for any known algorithm. Implicit enumeration of all feasible solutions seems to be the only way to solve an NP-complete optimization problem. The class of branch-and-bound algorithms contains the fastest and most commonly used algorithms to solve NP-complete optimization problems with the help of computers [LaW66], [Mit70].

A branch-and-bound algorithm is a structured search for the best member in the space of all feasible solutions of an optimization problem. This search can be represented by a search tree, whose root node represents the initial space of all feasible solutions. The branching part of the algorithm consists of partitioning the initial space into smaller and smaller subspaces, which are represented implicitly rather than by explicit enumeration of all members. The search tree represents each partitioned search space by an internal node, whose child nodes correspond with the generated subspaces. If the partitioning of a subspace results in the generation of singletons, the discovered solutions have to be compared with the best solution found so far to determine the currently best solution. The singletons are represented by leaf nodes in the search tree.

The bounding part of the algorithm consists of calculating an upper bound on the quality of the solutions in each generated composite subspace. If this upper bound is not better than the quality of the best solution found so far, the corresponding subspace can be excluded from further partitioning, because it will certainly not provide the discovery of a new best solution. These inferior composite subspaces are also represented by leaf nodes in the search tree and their exclusion from further partitioning is called pruning in that context. If all composite subspaces have been pruned or partitioned into singletons, the algorithm terminates and the best solution found so far is returned as optimal solution.

3.1 Generally applicable branch-and-bound algorithms

Generally applicable branch-and-bound algorithms are specified in terms of a few modifiable functions, behind which the problem-specific information of the optimization problem has been hidden [KNK88]. This generalizing approach has two important advantages. Firstly, a new optimization problem can profit from all existing general branch-and-bound algorithms after defining its problem-specific functions once and for all. Secondly, a new general branch-and-bound algorithm can be used immediately by every optimization problem for which the problem-specific functions have been defined once. This is a very practical property in view of the development of parallel branch-and-bound algorithms.

Functional programming languages are pre-eminently suitable for the implementation of general branch-and-bound algorithms, because the algorithms can be defined as higher-order functions with the problem-specific functions as arguments [Hug89]. The main function of the general branch-and-bound algorithm has been defined as follows:

$$\texttt{optimize}\ \langle \textit{prob\_spec\_func\_list} \rangle\ \langle \textit{search\_space} \rangle\ \langle \textit{worst\_case\_solution} \rangle \rightarrow \langle \textit{best\_solution} \rangle$$

This function returns $\langle \textit{best\_solution} \rangle$ as result of the search for the best member in $\langle \textit{search\_space} \rangle$, which is an implicit representation of the initial space of all feasible solutions of the problem instance. The problem-specific functions of the optimization problem are supplied by the $\langle \textit{prob\_spec\_func\_list} \rangle$. The $\langle \textit{worst\_case\_solution} \rangle$ serves as best solution found so far, until a better solution has been discovered. The worst case solution is usually a dummy solution of infinitely bad quality.

3.2 The problem-specific functions

The specification of the general branch-and-bound algorithm requires the definition of only three problem-specific functions. All three functions are applied to a single argument, which is an implicit representation of a subspace of the initial space of all feasible solutions of the problem instance. The three problem-specific functions are:

$$\texttt{singleton}\ \langle \textit{search\_space} \rangle \rightarrow \langle \textit{boolean\_value} \rangle$$

This function checks whether $\langle$*search_space*$\rangle$ consists of exactly one solution and returns $\langle$*boolean_value*$\rangle$.

`divide` $\langle$*search_space*$\rangle$ $\rightarrow$ $\langle$*list_of_subspaces*$\rangle$

This function partitions a necessarily composite $\langle$*search_space*$\rangle$ into a $\langle$*list_of_subspaces*$\rangle$, which contains two or more disjoint elements.

`upper_bound` $\langle$*search_space*$\rangle$ $\rightarrow$ $\langle$*numeric_value*$\rangle$

This function returns $\langle$*numeric_value*$\rangle$ as upper bound on the quality of the solutions in $\langle$*search_space*$\rangle$. This quality is measured by a numerical property of the optimization problem. Minimization problems have their numerical property inversely proportional to the quality of the solutions, but this can be rectified by systematic negation of the numerical property. The algorithm to calculate the upper bound of $\langle$*search_space*$\rangle$ has to be of lower complexity than any algorithm to determine the exact value. The upper bound of $\langle$*search_space*$\rangle$ may never be smaller than the upper bound returned for any of its subspaces. If $\langle$*search_space*$\rangle$ is a singleton, the exact quality of its solution has to be returned.

Most general branch-and-bound algorithms will apply the "upper_bound"-function more than once to the same search space. To avoid repeated calculation, the upper bounds can be pre-calculated by the "divide"-function and stored in the implicit representation of the generated subspaces.

3.3 Search strategies

The search strategy of a branch-and-bound algorithm determines after each step of the partitioning process, which unpruned composite subspace of the initial space of all feasible solutions has to be partitioned next. Three sequential search strategies are distinguished:

- The **depth-first** strategy always partitions the last generated unpruned composite subspace next. The partitioning order of simultaneously generated subspaces is undefined. The search tree will be generated in a depth-first order. Implementation requires storing the composite subspaces in a LIFO-queue (Last–In First–Out). The memory consumption increases linearly with the problem size.

- The **depth-first best-first** strategy is almost identical to the depth-first strategy, except that simultaneously generated subspaces are partitioned into descending order of upper bound. Implementation requires sorting of simultaneously generated subspaces before adding them to the LIFO-queue.

- The **best-first** strategy partitions the composite subspaces into descending order of upper bound. The search tree will be generated in a best-first order. Implementation requires storing the composite subspaces in an ordered list, sorted into descending order of upper bound. The memory consumption increases exponentially with the problem size. The best-first branch-and-bound algorithm is better known as the A*-algorithm in artificial intelligence [Nil80].

The search strategy to be preferred depends on the finiteness of the optimization problem and the tightness of its upper bounds. Tightness can be defined as a high probability to find the best solution in the subspace with the highest upper bound. Tight optimization problems can be solved best with the depth-first best-first strategy. At the cost of a little sorting overhead, the partitioning of non-critical subspaces is restricted by early discovery of (near)-optimal solutions. A subspace is called critical and unprunable under every search strategy, if its upper bound exceeds the quality of the final optimal solution. The depth-first strategy can be used for solving non-tight optimization problems with a finite number of feasible solutions.

To avoid endless subspace partitioning without discovering any solution, non-tight optimization problems with infinitely many feasible solutions have to be solved with the best-first strategy. The partitioning of non-critical subspaces is minimized at the cost of exponential memory consumption and a lot of sorting overhead. The IDA*-algorithm (Iterative Deepening A*) repeatedly performs a bounded depth-first search on the initial space of all feasible solutions [Kor85]. The search depth is increased after every iteration, until singletons are discovered. The IDA*-algorithm exhibits linear memory consumption to solve non-tight infinite optimization problems at the cost of repartitioning the subspaces from previous iterations in every next iteration. The parallelization of the IDA*-algorithm in a functional programming environment is extensively described in [Gla92].

4 The cargo loading problem

The cargo loading problem is a sugared version of the knapsack problem [MaT79]. A ship with tonnage T has to be loaded with a subset of the indivisible elements from a depot. Each element in the depot has its own weight w and price p. The total weight of all elements exceeds the tonnage of the ship. The total price of the cargo has to be maximized, while the total weight of the cargo may not exceed the tonnage of the ship. Tonnage, weights and prices are positive integers. Every subspace of the initial space of all feasible solutions of a problem instance is represented by the following tuple:

$(\langle cargo\_list\rangle, \langle trash\_list\rangle, \langle quay\_queue\rangle, \langle depot\_list\rangle, UB)$

This subspace tuple includes all solutions with the elements in the $\langle$*cargo_list*$\rangle$ already loaded, the elements in the $\langle$*trash_list*$\rangle$ discarded and the remaining elements not yet considered. The $\langle$*cargo_list*$\rangle$, the $\langle$*trash_list*$\rangle$ and the $\langle$*quay_queue*$\rangle$ are empty initially, while the $\langle$*depot_list*$\rangle$ contains all elements from the depot, sorted into descending order of p/w-ratio. The upper bound UB is stored in the subspace tuple to avoid repeated calculation (see 3.2). Tighter upper bounds can be calculated by previewing more than just the first element in the $\langle$*depot_list*$\rangle$. Efficiency underlies the transfer of these elements to a FIFO-queue (First–In First–Out) called $\langle$*quay_queue*$\rangle$, from which they can either be loaded or discarded. The preview to calculate the upper

rule	(	*cargo,*	*trash,*	*quay,*	*depot,*	*UB*	)	*comments*
–	(	[ ],	[ ],	[ ],	[8/3, 9/4, 2/1, 6/5],	–	)	initial search space
3	(	[ ],	[ ],	[8/3],	[9/4, 2/1, 6/5],	–	)	
5	(	[ ],	[ ],	[8/3],	[9/4, 2/1, 6/5],	12	)	$UB = 8 + (9 \cdot (5 - 3))$ div 4
A	(	[8/3],	[ ],	[ ],	[9/4, 2/1, 6/5],	–	)	first subspace
4	(	[8/3],	[9/4],	[ ],	[2/1, 6/5],	–	)	
3	(	[8/3],	[9/4],	[2/1],	[6/5],	–	)	
4	(	[8/3],	[9/4, 6/5],	[2/1],	[ ],	–	)	
1	(	[8/3, 2/1],	[9/4, 6/5],	[ ],	[ ],	10	)	$UB = 8 + 2$
B	(	[ ],	[8/3],	[ ],	[9/4, 2/1, 6/5],	–	)	second subspace
3	(	[ ],	[8/3],	[9/4],	[2/1, 6/5],	–	)	
3	(	[ ],	[8/3],	[9/4, 2/1],	[6/5],	–	)	
2	(	[ ],	[8/3, 6/5],	[9/4, 2/1],	[ ],	–	)	
1	(	[9/4, 2/1],	[8/3, 6/5],	[ ],	[ ],	11	)	$UB = 9 + 2$

$$T = 5\ ; \quad depot\ \langle p/w \rangle = [8/3,\ 9/4,\ 2/1,\ 6/5]$$

Table 1: Tuple transitions of example cargo loading problem.

bound UB requires each subspace tuple to traverse the following scheme of conditional transition rules:

1. **if** the $\langle depot\_list \rangle$ is empty **then**
   empty the $\langle quay\_queue \rangle$ into the $\langle cargo\_list \rangle$;
   $UB = P_{cargo}$; **exit**
2. **if** $W_{cargo} + W_{quay} = T$ **then**
   empty the $\langle depot\_list \rangle$ into the $\langle trash\_list \rangle$;
   goto 1.
3. **if** $W_{cargo} + W_{quay} + w_1 \leq T$ **then**
   transfer first element p_1/w_1 from the
   $\langle depot\_list \rangle$ to the $\langle quay\_queue \rangle$; **goto** 1.
4. **if** $W_{cargo} + w_1 > T$ **then**
   transfer first element p_1/w_1 from the
   $\langle depot\_list \rangle$ to the $\langle trash\_list \rangle$; **goto** 1.
5. $UB = P_{cargo} + P_{quay} + (p_1 \cdot (T - W_{cargo} - W_{quay}))$ div w_1

P_{cargo} and W_{cargo} are the total price and weight of all elements in the $\langle cargo\_list \rangle$, P_{quay} and W_{quay} are the total price and weight of all elements in the $\langle quay\_queue \rangle$ and p_1 and w_1 are the price and weight of the first element in the $\langle depot\_list \rangle$.

The three problem-specific functions of the general branch-and-bound algorithm are defined as follows for the cargo loading problem:

- The function `singleton` $\langle search\_space \rangle$ returns **true**, if the $\langle depot\_list \rangle$ in tuple $\langle search\_space \rangle$ is empty and **false** otherwise.
- The function `divide` $\langle search\_space \rangle$ returns a list with two disjoint subspaces of tuple $\langle search\_space \rangle$. Starting with tuple $\langle search\_space \rangle$, the first subspace tuple is generated by transferring the first element of the $\langle quay\_queue \rangle$ to the $\langle cargo\_list \rangle$ (rule A). The second subspace tuple is generated by transferring the first element of the $\langle quay\_queue \rangle$ to the $\langle trash\_list \rangle$ instead (rule B). Next the transition scheme has to be traversed for both generated subspace tuples to calculate the value of their upper bounds (rule 1–5).
- The function `upper_bound` $\langle search\_space \rangle$ simply returns the value of the upper bound UB in tuple $\langle search\_space \rangle$.

As an illustration, the initial space of all feasible solutions of a small problem instance has been partitioned into two subspaces and their upper bounds have been calculated. The executed transition rules are shown in Table (1).

4.1 Sequential experiments

Three fixed test-sets of cargo loading problems have been generated to analyse the performance of both the sequential and parallel general branch-and-bound algorithms. All test-sets contain a considerable amount of problem instances (100), because the computational behaviour of branch-and-bound algorithms is highly dependent on the input data. The first test-set contains problem instances with uncorrelated elements. Price p and weight w are both random integers from the interval [1,100]. The other test-sets contain problem instances with correlated elements. The elements of the second test-set have random integer weights w from the interval [1,100], from which the prices $p = w + c$ are derived. The elements of the third test-set have random integer prices p from the interval [1,100], from which the weights $w = p + c$ are derived. The constant c has been assigned the value 10 for both correlated test-sets. All three test-sets consist of depots with 20 elements. The tonnage of the ship is half of the average total weight of all elements in the depot, being 600 for the third test-set and 500 for the other test-sets. The characteristics of the test-sets are similar to those used by [MaT79], [JAM88].

Table (2) shows the experimental results of three cargo loading test-sets solved by the sequential general branch-and-bound algorithm. The sorting overhead of the best-first strategy causes the execution of many extra reduction steps without a significant decrease of subspace partitions. This shows the tightness of the upper bounds calculated for the cargo loading problem. The small performance difference between the depth-first and depth-first best-first strategy shows that the

test-set	*strategy*	*nr_red*	*nr_part*	*inh_seq*	*avg_par*
p/w	depth-first	21,200	16.98	12.79	1.33
	d-1st b-1st	21,947	17.00	12.79	1.33
	best-first	21,420	16.48	?	?
$(w+10)/w$	depth-first	172,596	143.56	17.98	7.98
	d-1st b-1st	176,695	143.56	17.98	7.98
	best-first	225,775	137.03	?	?
$p/(p+10)$	depth-first	134,348	89.25	9.80	9.11
	d-1st b-1st	136,779	89.01	9.80	9.08
	best-first	234,226	89.43	?	?

nr_red = *Average number of reduction steps per problem instance.*
nr_part = *Average number of subspace partitions per problem instance.*
inh_seq = *Average number of inherently sequential subspace partitions per problem instance.*
avg_par = *Average number of useful subspace partitions, that can be executed in parallel (= nr_part / inh_seq).*

Table 2: Sequential results of the cargo loading test-sets.

"divide"-function of the cargo loading problem returns its subspaces almost always in descending order of upper bound. This causes the depth-first strategy to act like the depth-first best-first strategy. The speedup of the parallel branch-and-bound implementations will be calculated with regard to the sequential results of the depth-first strategy.

4.2 Potential parallelism

Table (2) also gives an approximation of the amount of potential parallelism in the three cargo loading test-sets. The partitioning of one search space into subspaces with pre-calculated upper bounds is the finest-granular parallelizable task in a parallel general branch-and-bound implementation. Subdivision of the "divide"-function to create parallelizable tasks of even finer grain size would conflict with the generalizing branch-and-bound approach. The depth of the generated search tree equals the number of inherently sequential subspace partitions of a problem instance. The ratio of the total number of subspace partitions to the number of inherently sequential subspace partitions is the average number of useful subspace partitions that can be executed in parallel. This measure provides an upper bound on the attainable speedup for any parallel architecture.

The test-set with uncorrelated elements turns out to contain almost no potential parallelism. It produces chain-like search trees and is unsuitable for analysing parallel branch-and-bound implementations. The moderate amount of potential parallelism in the test-sets with correlated elements is large enough for analysing parallel branch-and-bound implementations on a limited number of processors. The amount of potential parallelism in a test-set can be increased by putting more elements in the depots to enlarge the problem size or by decreasing the pruning efficiency. The latter option can be realized by applying a simpler but less tight upper bound calculation, arranging a test-set with difficult instead of random problem instances or choosing the constant c lower than 10 to reduce the variation in the quality of the feasible solutions, which increases the average difficulty of the problem instances. However, care has to be taken that decreasing the pruning efficiency does not push the total number of executed subspace partitions towards an order of magnitude b^{inh_seq}, where the cargo loading problem has a branching factor $b = 2$. A low pruning efficiency turns a branch-and-bound algorithm into an ennobled divide-and-conquer algorithm, which would veil the limitations of parallel functional programming environments.

5 Parallel branch-and-bound implementations

Three parallel functional implementations of a generally applicable branch-and-bound algorithm have been developed. Two functional equivalents of imperative parallel branch-and-bound algorithms are compared with a dedicated parallel functional branch-and-bound implementation.

5.1 Imitation of imperative approaches

Imperative parallel branch-and-bound algorithms exist between two extreme approaches. The centralized approach gives each processor asynchronous access to shared memory or the local memory of a particular processor. Search spaces to partition are taken from centralized subspace storage and generated composite subspaces are reinserted. The centralized best solution found so far is updated with discovered superior solutions and inferior search spaces are pruned against it [WaM84], [Qui90]. The distributed approach consists of private subspace storage and a private best solution found so far in the local memory of every processor. Each local partitioning process can distribute generated composite subspaces to other processors by asynchronous communication and broadcasts new best solutions to all processors [EDH80], [JAM88], [TrO89], [Qui90].

Two straightforward parallel functional implementations have imitated the imperative approaches without using their asynchronous communication methods, because the fork-join model does not permit any direct communication between parallel jobs. Indirect communication has to be committed to limit the speculative transformation loss, which is caused by the partitioning of too many non-critical subspaces. The number of available processors has to be known in advance by means of the constant `NR_PROC`. Both implementations alternate parallel processing phases and explicit information exchange phases, until all composite subspaces have been pruned or partitioned into singletons. The former phase evaluates `NR_PROC` parallel jobs as leaf nodes of a binary hierarchy. The latter phase performs the indirect communication between the `NR_PROC` parallel jobs, joining the hierarchy into the sequential main job and regenerating it for the next parallel processing phase. Bounding the number of generated parallel jobs by the number of processors has to limit speculative speculative transformation loss. The implementations operate as follows:

- The **centralized implementation** performs a parallel depth-first or depth-first best-first strategy and minimizes speculative transformation loss. Each parallel job may partition one search space during a parallel processing phase. The information exchange phase optionally sorts all generated composite subspaces before adding them to a centralized LIFO-queue, compares discovered singletons with the centralized best solution found so far, prunes inferior subspaces and takes search spaces from the LIFO-queue for partitioning in the next parallel processing phase. Experiments with the centralized implementation resulted in a very disappointing speedup, because the processor utilization suffered a lot from inherent task shortage. The grain size of the mainly sequential information exchange phase is too big in comparison with the grain size of the jobs in the parallel processing phase.

- The **distributed implementation** allows each parallel job to execute the same number of subspace partitions in a parallel processing phase. The local partitioning process maintains a private LIFO-queue to perform a depth-first strategy, compares discovered singletons with its private best solution found so far and prunes inferior subspaces. The information exchange phase compares all private best solutions to determine the global best solution found so far, predetermines the number of subspaces that each processor may partition in the next parallel processing phase and takes the first search space from each non-empty private LIFO-queue to insert it best-first into another private LIFO-queue, which is determined by a tree-oriented static distribution function [Qui90]. Experiments with the distributed implementation resulted in a disappointing speedup, because the processor utilization suffered a lot from load imbalance. The pruning and non-adaptive distribution of composite subspaces can result in parallel jobs with empty private LIFO-queues without having partitioned the allowed number of subspaces.

Detailed information about the centralized and distributed implementations and their performance has been described in [Gla92].

5.2 Fully parallel partitioning

Fully parallel partitioning performs the repeated partitioning of one or several search spaces into smaller and smaller subspaces, until all have been pruned or identified as singleton. Every generated subspace is partitioned as a separate job. This divide-and-conquer strategy generates an abundance of parallelizable jobs, which enables the system scheduler to minimize load imbalance and maintain a high processor utilization. However, the generated composite subspaces can only be pruned against the best solution found before the start of the fully parallel partitioning. Because the fork-join model does not permit direct communication between child jobs, new best solutions discovered during fully parallel partitioning can not be accessed before this partitioning process has been finished. Therefore, uncontrolled use of fully parallel partitioning can severely decrease the pruning efficiency, which results in a huge speculative transformation loss. Simply applying fully parallel partitioning to the initial space of all feasible solutions is useless, because all composite subspaces have to be pruned against the worst case solution in that case.

Fully parallel partitioning makes demands on the scheduling algorithm used by the system layer. If the hierarchy of jobs is executed in a parallel breadth-first order, storing the generated jobs causes an exponential memory consumption. Storage problems can be avoided with a parallel depth-first scheduling strategy, which executes the last generated jobs first and exhibits a linear memory consumption. Implementation of this scheduling strategy requires one centralized or a number of distributed LIFO-queues.

5.3 The parallel "colonizer"-algorithm

The parallel "colonizer"-algorithm profits from a controlled use of fully parallel partitioning. The simple version consists of a sequential (SPINE_BRANCH) section followed by several fully parallel (FULL_BRANCH) sections. The SPINE_BRANCH section performs a sequential branch-and-bound search with a depth-first best-first strategy, until the first singleton is discovered. Figure (1a) shows the generated main path (the spine) of the search tree for an optimization problem with branching factor 2. Figure (1b) shows the application of the FULL_BRANCH sections during the node by node (the vertebrae) ascent of the spine. Every FULL_BRANCH section partitions the composite subspaces (the ribs) branching from one of the vertebrae fully in parallel. The resulting best solution found so far is passed along the spine for the benefit of the pruning process in the next FULL_BRANCH section. The vertebrae serve as implicit information exchange points. The ethymology of the parallel "colonizer"-algorithm refers to the colonial

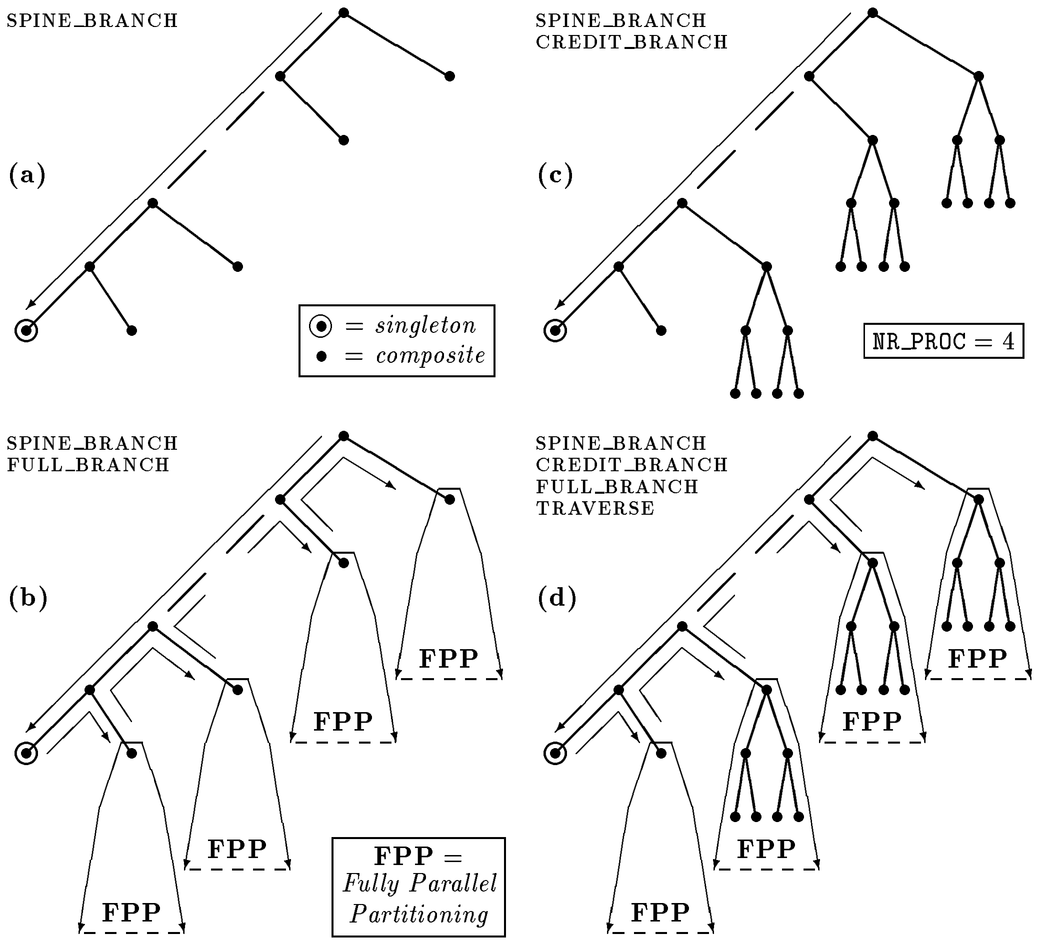
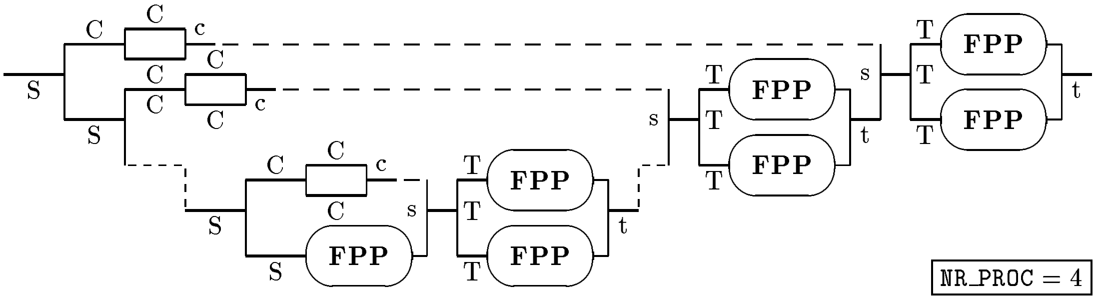


Figure 1: The parallel "colonizer"-algorithm

era, when the colonization of a tropical country started with paving a highway (the spine) through the jungle (search tree), which was ransacked on both sides (the ribs) during retreat to take the valuables (best solutions) to the home country.

The sequential SPINE_BRANCH section of the simple version of the parallel "colonizer"-algorithm badly influences the average processor utilization and the speedup. Inherent task shortage can be minimized by performing a pre-unfolding (CREDIT_BRANCH) section in parallel with the SPINE_BRANCH section at each vertebra of the spine. Every CREDIT_BRANCH section partitions the composite subspaces branching from one of the vertebrae in advance, following a parallel breadth-first strategy, which partitions every generated subspace in a separate job. However, the total number of subspace partitions executed in a CREDIT_BRANCH section is explicitly bounded by the assigned credit. When a CREDIT_BRANCH job generates child jobs for its composite subspaces, its remaining credit is divided as equally as possible. The credit assignment requires the number of available processors to be known in advance by means of the constant NR_PROC. The CREDIT_BRANCH sections are susceptible to speculative transformation loss, because all subspaces have to be pruned against the worst case solution. Therefore, the CREDIT_BRANCH sections should not generate more work than NR_PROC − 1 processors can execute during the execution of the SPINE_BRANCH section by the remaining processor. This requirement can be met by assigning each CREDIT_BRANCH section credit for NR_PROC − 1 subspace partitions. The grain sizes of all partitioning operations are assumed to be roughly equal.

Figure (1c) shows the search tree after the execution of the SPINE_BRANCH and CREDIT_BRANCH sections for NR_PROC = 4. Each CREDIT_BRANCH section returns the partially unfolded ribs at its vertebra as a subtree with composite subspaces as leaf nodes. Every FULL_BRANCH section partitions the non-prunable subspaces from one of the subtrees fully in parallel. It has to be preceded by a TRAVERSE section, which performs a fully parallel traversal towards the leaf nodes of this subtree. Separate jobs are generated for each leaf node, which reprune the composite subspaces against the best solution returned by the previous FULL_BRANCH section, before starting the next FULL_BRANCH section. Figure (1d) shows the search tree generated by the extended version of the parallel "colonizer"-algorithm for NR_PROC = 4. Figure (2) shows a schematic generalization of the job graphs produced by the parallel "colonizer"-algorithm for NR_PROC = 4. The job lengths do not scale with the number of executed reduction steps, although jobs partitioning a search space have been displayed twice as long. The nesting of the SPINE_BRANCH jobs depends on the problem instance. The FULL_BRANCH sections have been abstracted from the scheme for clarity.



S, s	=	*Fork job, join job of type* SPINE_BRANCH.
C, c	=	*Fork job, join job of type* CREDIT_BRANCH.
T, t	=	*Fork job, join job of type* TRAVERSE.
FPP	=	*Fully Parallel Partitioning by means of fork and join jobs of type* FULL_BRANCH.

Figure 2: Job-graph of the parallel "colonizer"-algorithm

5.4 Performance evaluation

Table (3) shows the results of the correlated cargo loading test-sets solved by the parallel "colonizer"-algorithm. Taking NR_PROC = 1 corresponds with the simple version (see 5.3). The extension with CREDIT_BRANCH and TRAVERSE sections has minimized its large inherently sequential fraction and increased its moderate average parallelism. The small speculative transformation loss has increased as well, but every critical subspace partitioned by the SPINE_BRANCH sections is pure gain. The difference between the speedup bounds of both correlated test-sets relates to their difference in amount of potential parallelism (see 4.2).

Figure (3) shows the speedup bounds of the parallel "colonizer"-algorithm in comparison with the inferior speedups $S^c(n)$ and $S^d(n)$ of the centralized and distributed parallel functional implementations (see 5.1). Simpler subspace storage and the absence of explicit information exchange phases has turned considerable conservative transformation loss into a minimal gain. Controlled use of fully parallel partitioning has hardly increased the speculative transformation loss. Because the absence of explicit information exchange phases decreases inherent task shortage and fully parallel partitioning decreases the load imbalance of the distributed implementation, the efficiency $E(n)$ should improve as well. The increased average parallelism and decreased inherently sequential fraction confirm this assumption. Because the estimated efficiency $E_e(n)$ of the parallel "colonizer"-algorithm is still slightly lower than the efficiency of the distributed implementation, the real speedup $S(n)$ is probably located above the estimated speedup $S_e(n)$.

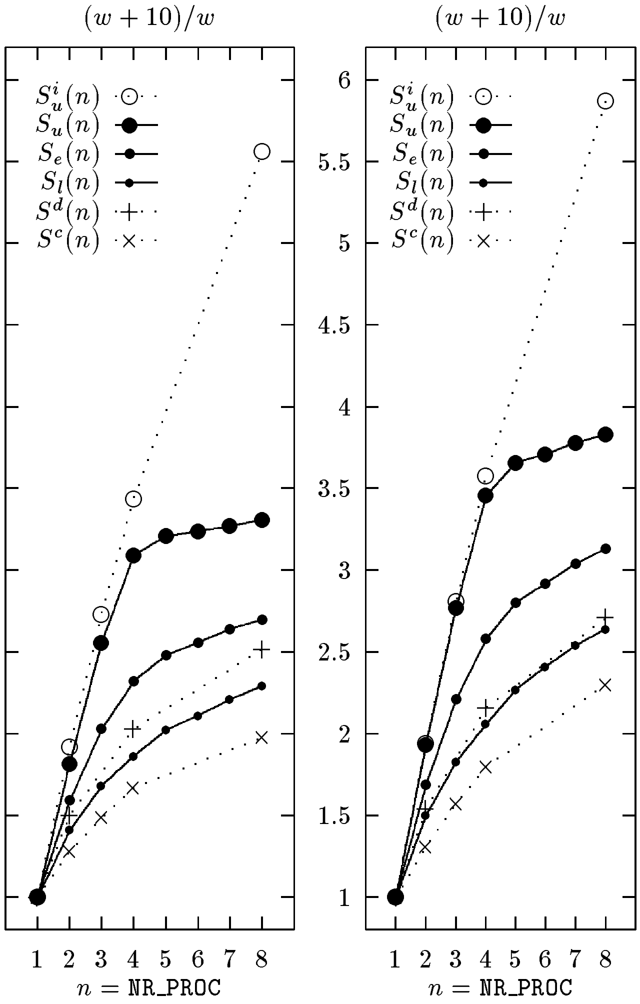


Figure 3: Speedup comparison.

Table (3) and Figure (3) also show the upper bound $S_u^i(n)$ on the speedup of imperative parallel branch-and-bound algorithms solving the cargo loading test-sets. Only the minimum speculative transformation loss and the inherent task shortage in the first and last part of the partitioning process have been taken into account. Assuming that the grain-sizes of all partitioning operations are roughly equal, a synchronous information exchange phase after every parallel partitioning operation does not provide more speculative transformation loss than asynchronous information exchange. Therfore, the upper bound $S_u^i(n)$ has been defined as the ratio of P_{seq} (see 2.2) to the average number of parallel processing phases executed by the centralized parallel functional implementation with depth-first best-first strategy (see 5.1).

The difference between the upper bound $S_u^i(n)$ on the imperative speedup and the upper bound $S_n(n)$ on the speedup of the parallel "colonizer"-algorithm stays rel-

test-set	NR_PROC	A	f	CTL	STL	$E_e(n)$	$S_e(n)$	$S_u(n)$	$S_u^i(n)$
$(w+10)/w$	1	2.90	0.19	1.00	0.96	–	–	–	–
	$n=2$	3.15	0.02	1.01	0.92	0.85	1.59	1.82	1.92
	$n=3$	3.46	0.02	1.00	0.88	0.77	2.03	2.56	2.73
	$n=4$	3.63	0.02	1.01	0.84	0.68	2.32	3.09	3.44
	$n=8$	4.57	0.02	1.02	0.71	0.47	2.70	3.31	5.56
$p/(p+10)$	1	2.92	0.18	1.00	0.98	–	–	–	–
	$n=2$	3.21	0.02	1.02	0.96	0.86	1.69	1.93	1.94
	$n=3$	3.55	0.02	1.01	0.94	0.77	2.21	2.77	2.81
	$n=4$	3.71	0.02	1.02	0.91	0.69	2.58	3.46	3.58
	$n=8$	4.61	0.02	1.03	0.81	0.47	3.13	3.83	5.87

Table 3: Experimental results of the parallel "colonizer"-algorithm.

atively small for up to four processors. The greater difference for a higher number of processors can only be reduced by increasing the average parallelism, which requires a smaller number of FULL_BRANCH sections. However, simultaneous fully parallel partitioning of the subtrees at more than one vertebra did more increase speculative transformation loss than it increased the average parallelism. A fair comparison of the parallel "colonizer"-algorithm with real imperative parallel branch-and-bound algorithms requires a full imperative implementation with the same optimization problems, the same upper bound calculations and test-sets with equal characteristics.

6 Conclusions

The implementation of parallel branch-and-bound algorithms in a functional programming environment requires making the trade-off between a lower average processor utilization as result of synchronous communication and the performance loss caused by exploitation of speculative parallelism. This trade-off has been optimized by the parallel "colonizer"-algorithm. This dedicated parallel functional branch-and-bound implementation shows a better performance than the centralized and distributed implementations, which imitate imperative parallel branch-and-bound approaches. The parallel "colonizer"-algorithm provides considerable speedup despite the communicative restrictions of the fork-join model. However, it cannot compete with imperative parallel branch-and-bound algorithms, if the number of available processors is close to the amount of potential parallelism in the problem instance.

6.1 Future work

Recently, the experience obtained with the implementation of parallel functional branch-and-bound algorithms has resulted in an idea for a new speculative parallel annotation. This annotation enables the system layer to exercise run-time control over speculative parallel jobs, by which the limitation of speculative transformation loss does not decrease the average processor utilization any longer. The new annotation is referential transparant, provides simpler parallel functional branch-and-bound implementations, improves the portability and probably approximates the performance of imperative parallel branch-and-bound algorithms. The implementation of this annotation has been planned, but performance results are not yet available. The idea has been worked out in the appendix.

Appendix

A.1 A speculative parallel annotation

The parallel branch-and-bound implementations have applied the "sandwich"-function to express their speculative parallelism, although this annotation has been designed for expressing conservative parallelism. The "sandwich"-annotation does not provide the possibility to discard a speculative job, when its evaluation has been made superfluous by the results of other jobs. The programmer has to exercise compile-time control over the generation of speculative jobs to limit speculative transformation loss, but this information exchange harms the average parallelism and with that the processor utilization. The introduction of a speculative parallel annotation is desirable, but referential transparancy imposes restraints on its semantics. Utilizing speculative parallelism without affecting referential transparancy can only be realized by using the conditional expressions of functional languages. While the "if–then–else"-operator evaluates the "if"-expression, both the "then"- and "else"-expression can be evaluated by speculative parallelizable jobs [Pey87]. After the "if"-expression has returned its boolean value, either the "then"- or the "else"-expression becomes vital, while the other one may be discarded.

Dedicated use of the speculative parallelism in the conditional expression results in a definition proposal for a new speculative counterpart of the conservative "sandwich"-function. It is presented in isolation, although mixed use with conservative parallel annotations has not been ruled out. The underlying parallel model is formed by a hierarchy of speculative graph reducing processes. Every speculative parent job in the hierarchy forks into a sequence of speculative parallelizable child jobs by applying a speculative parallel annotation. Non-vital jobs may be discarded together with all their descendant jobs. The vitality of a speculative job depends on a joint result of all jobs on the left side of its path to the root of the hierarchy. This joint result propagates sequentially through the hierarchy of jobs along the boundary between vital and non-vital jobs, while the bulk of the work is executed in advance by parallel evaluation of speculative jobs.

The evaluation of non-vital jobs has to be minimized by the system layer, which requires the introduction of priority scheduling. Every speculative job in the hierarchy is assigned a unique priority. If the vitality of a speculative job depends on the result returned by another job, the latter job has to be assigned a higher priority. Recursive application of this rule results in a priority assignment, in which the priority descends with the left-first preorder traversal of the job hierarchy. The priority scheduling algorithm could apply a pre-emptive scheduling strategy, which means that running jobs are suspended immediately, when runnable jobs of higher priority become available. However, the possibly huge intermediate reduction graphs can provide storage problems. Therefore, the run-to-completion strategy is proposed. It does not suspend running jobs, until they fork by applying a parallel annotation. If the grain sizes of the jobs in the hierarchy are roughly equal, the total number of reduction steps will not increase much in comparison with pre-emptive scheduling. The killing of non-vital running jobs can be executed immediately without problems.

A.2 The "casino"-annotation

The "casino"-function is a proposal for a new speculative parallel annotation, which is executed by a parent job to generate a sequence of speculative child jobs. The ethymology refers to the double sense of the word "casino". The casino as gambling house stresses the speculative aspect and casino as name for a kind of square bread stresses the relation with the "sandwich"-function. A formal definition of the "casino"-function is given by the following term rewrite rule:

$$
\begin{aligned}
&\texttt{casino}\ \ \mathcal{P}_0\ \mathcal{C}_1\,\mathcal{F}_1\ \ldots\ \mathcal{C}_m\,\mathcal{F}_m\ \ \delta_0 \\
&\quad =\ d_{m+1} \\
&\qquad \textbf{where} \\
&\qquad\quad \begin{array}{rcl}
d_1 & = & \mathcal{P}_0\,\delta_0 \\
d_{i+1} & = & \textbf{if}\ \mathcal{C}_i\,d_i\ \ \textbf{then}\ d_i\ \ \textbf{else}\ \mathcal{F}_i\,d_i \\
\mathcal{F}_i\,\delta_i & = & \mathcal{G}_i\ \ y_{i,1}\ \ldots\ y_{i,n_i}\ \ \delta_i
\end{array}
\end{aligned}
$$

If the parent job applies the "casino"-function, m speculative child jobs are generated to evaluate the redexes $\mathcal{F}_1\,\delta_1$ to $\mathcal{F}_m\,\delta_m$ in parallel. Apart from non-strictness and the extension with an extra last parameter, the definition of the child job redexes is equivalent to their definition in the "sandwich"-function.

The extra parameter δ_i denotes a delayed argument, on which the vitality of the speculative child job depends, but whose value is possibly unknown at the time of generation. The delayed argument does not prevent the (lazy) evaluation of the speculative child job, because the delayed argument cannot be used in an ordinary functional expression. The parent job is suspended after generating the speculative child jobs, if the value of its own delayed argument is still unknown at that time. When the delayed argument has been received from the grandparent job, the awakened parent job substitutes its value for the last parameter δ_0 of the applied "casino"-function, which is the only place where delayed arguments do reappear. Next the value of d_1 is derived by applying the preprocess function $\mathcal{P}_0$ to the value of δ_0.

If the value of d_i has become known, the parent job evaluates the condition $\mathcal{C}_i\ d_i$ to determine the vitality of the speculative child job $\mathcal{F}_i\ \delta_i$. If **true** is returned, then the system layer can kill child job $\mathcal{F}_i\ \delta_i$ and all its descendant jobs and the value of d_{i+1} is derived from the assignment $d_{i+1} = d_i$. If **false** is returned, then the evaluation of child job $\mathcal{F}_i\ \delta_i$ is vital. If no normal form has been returned already, the parent job is suspended after sending the value of d_i as substitute for parameter δ_i to child job $\mathcal{F}_i\ \delta_i$, which may need the delayed argument to determine the vitality of its descendant jobs. When the child job has returned its normal form, the parent job awakes for the assignment of d_{i+1}. This pattern is repeated until the value of d_{m+1} has become known and can be returned as result of the applied "casino"-function.

A.3 Application of the "casino"-annotation

The introduction of the "casino"-annotation enables the implementation of a simple parallel branch-and-bound algorithm, which performs fully parallel partitioning on the initial space of all feasible solutions (see 5.2), but uses the "casino"-function instead of the "sandwich"-function. Every speculative job receives the best solution found so far as delayed substitute for parameter δ_i. A single partitioning operation precedes the application of the "casino"-function. The preprocess function $\mathcal{P}_0$ determines the new best solution d_1 by comparing generated singletons with the delayed best solution δ_0. The m generated composite subspaces are sorted into descending order of upper bound and partitioned by the functions $\mathcal{F}_1$ to $\mathcal{F}_m$ respectively. The conditional function $\mathcal{C}_i$ determines the vitality of the speculative child job $\mathcal{F}_i\ \delta_i$ by pruning its composite subspace against delayed best solution d_i. The main job can use the preprocess function $\mathcal{P}_0$ to introduce the worst case solution as delayed best solution.

The parallel branch-and-bound implementation with "casino"-annotation hardly requires more program code than the sequential branch-and-bound implementation, while the parallel "colonizer"-algorithm needed four times as much code. The portability improves as well, because the number of available processors does not have to appear as the constant `NR_PROC` in the program code. The "casino"-function can minimize speculative transformation loss without harming average parallelism and average processor utilization, because it enables the system layer to exercise run-time control over the execution of speculative jobs. The speedup of the parallel branch-and-bound implementation with "casino"-annotation will definitely exceed the speedup of the parallel "colonizer"-algorithm and can probably approximate the speedup of an imperative branch-and-bound algorithm with parallel depth-first strategy. However, the neglected influence of system and communication overhead on the speedup will be larger for a speculative parallel functional programming environment, because priority scheduling is required. The "casino"-annotation can also benefit the parallel functional implementation of other search applications, including the IDA*-algorithm [Gla92], AND/OR tree search and game tree search algorithms [KNK88].

References

[EDH80] O.I. El-Dessouki, W.H. Huen, *Distributed Enumeration on Between Computers*, IEEE Transactions on Computers, Vol. 29, No. 9, pp. 818–825, September 1980.

[EZL89] D.L. Eager, J. Zahorjan, E.D. Lazowska, *Speedup Versus Efficiency in Parallel Systems*, IEEE Transactions on Computers, Vol. 38, No. 3, pp. 408–423, March 1989.

[GaJ79] M.R. Garey, D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman & Co. , 1979.

[Gla92] J.C. Glas, *The Parallelization of Branch-and-Bound Algorithms in a Functional Programming Language*, M.Sc. Thesis, Department of Computer Systems, University of Amsterdam, April 1992.

[Har88] P.H. Hartel, *Performance Analysis of Storage Management in Combinator Graph Reduction*, Ph.D. Thesis, Department of Computer Systems, University of Amsterdam, 1988.

[Hug89] J. Hughes, *Why Functional Programming Matters*, The Computer Journal, Vol. 32, No. 2, pp. 98–107, February 1989.

[JAM88] V.K. Janakiram, D.P. Agrawal, R. Mehrotra, *A Randomized Parallel Branch-and-Bound Algorithm*, Proceedings of the 1988 International Conference on Parallel Processing, Vol. III, *Algorithms and Applications*, ed. D.H. Bailey, pp. 69–75, 1988.

[KNK88] V. Kumar, D.S. Nau, L.N. Kanal, *A General Branch-and-Bound Formulation for AND/OR Graph and Game Tree Search*, in *Search in Artificial Intelligence*, ed. L.N. Kanal, V. Kumar, pp. 91–130, Springer-Verlag, 1988.

[Kor85] R.E. Korf, *Depth-First Iterative Deepening: An Optimal Admissible Tree Search*, Artificial Intelligence, Vol. 27, No. 1, pp. 97–109, September 1985.

[LaW66] E.L. Lawler, D.E. Wood, *Branch-and-Bound Methods: A Survey*, Operations Research, Vol. 14, No. 4, pp. 699–719, July 1966.

[MaT79] S. Martello, P. Toth, *The 0–1 Knapsack Problem*, in *Combinatorial Optimization*, ed. N. Christofides, A. Mingozzi, P. Toth, C. Sandi, pp. 237–279, Wiley Interscience, 1979.

[MBS87] D.L. McBurney, M.R. Sleep, *Transputer Based Experiments with the ZAPP Architecture*, Parle Conference on Parallel Architectures and Languages, Lecture Notes in Computer Science 259, ed. J.W. de Bakker, A.J. Nijman, P.C. Treleaven, pp. 242–259, Eindhoven, The Netherlands, June 1987.

[Mit70] L.G. Mitten, *Branch and Bound Methods: General Formulations and Properties*, Operations Research, Vol. 18, No. 1, pp. 23–34, January 1970. Errata in Operations Research, Vol. 19, No. 2, pp. 550, March 1971.

[Nil80] N.J. Nilsson, *Principles of Artificial Intelligence*, Tioga Publishing Company, 1980, Springer-Verlag, 1982.

[Pey87] S.L. Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice-Hall, 1987.

[Qui90] M.J. Quinn, *Analysis and Implementation of Branch-and-Bound Algorithms on a Hypercube Multicomputer*, IEEE Transactions on Computers, Vol. 39, No. 3, pp. 384–387, March 1990.

[TrO89] J.M. Troya, M. Ortega, *A Study of Parallel Branch-and-Bound Algorithms with Best-First Search*, Parallel Computing, Vol. 11, No. 1, pp. 121–126, July 1989.

[Vre89] W.G. Vree, *Design Considerations for a Parallel Reduction Machine*, Ph.D. Thesis, Department of Computer Systems, University of Amsterdam, 1989.

[WaM84] B.H. Wah, Y.W.E. Ma, *MANIP – A Multicomputer Architecture for Solving Combinatorial Extremum-Search Problems*', IEEE Transactions on Computers, Vol. 33, No. 5, pp. 377–390, May 1984.