Parallel Asynchronous Tabu Search for Multicommodity Location-Allocation with Balancing Requirements

Teodor Gabriel Crainic
Centre de recherche sur les transports
Université de Montréal
and
Département des sciences administratives
Université du Québec à Montréal

Michel Toulouse
Centre de recherche sur les transports
Université de Montréal
and
Département de génie électrique
École Polytechnique

Michel Gendreau
Centre de recherche sur les transports
and
Département d'informatique et de recherche opérationnelle
Université de Montréal

November 1994

Abstract

We study and compare asynchronous parallelization strategies for tabu search, and evaluate the impact on performance and solution quality of some important algorithmic design parameters: number of processors, handling of exchanged information, etc. Parallelization approaches are implemented and compared by using a tabu search algorithm for multicommodity location-allocation problems with balancing requirements.

Key words: Tabu search methods, Parallel algorithms, Asynchronous strategies, Multicommodity location-allocation with balancing requirements

Résumé

Nous étudions et comparons plusieurs méthodes parallèles asynchrones de recherche avec tabous. Nous identifions les approches de parallélisation les plus prometteuses et nous évaluons l'impact sur le comportement des méthodes et sur la qualité des solutions de plusieurs aspects importants du design algorithmique: nombre de processeurs, traitement de l'information échangée, etc. Les approches de parallélisation sont examinées, implantées et comparées, à partir d'un algorithme de recherche avec tabous pour le problème de localisation-allocation multiproduits avec demandes d'équilibrage.

Mots clés: Méthodes de recherche avec tabous, stratégies de parallélisation asynchrones, localisation-allocation multiproduits avec d'équilibrage

1 Introduction

Tabu search [9, 10, 11, 12] is often described as a *higher level* heuristic for solving optimization problems, designed to guide other heuristics, or parts thereof, to avoid the trap of local optimality. Thus, tabu search is an adaptive search technique that aims to intelligently explore the solution space in search of good, hopefully optimal, solutions. Broadly speaking, two mechanisms are used to direct the search trajectory. The first is intended to avoid cycling through the use of *tabu lists* that keep track of recently examined solutions. The second mechanism makes use of one or several *memories* to direct the search either into a thorough exploration of a promising neighbourhood, or towards previously unexplored regions of the solution space.

Parallel computer architectures offer the possibility to design procedures that explore more efficiently the solution space. Generally, this additional efficiency may be achieved by accelerating some particularly tedious computational phases of the algorithm, by redesigning parts of the algorithm to take advantage of the parallel architecture, or by performing several searches simultaneously. One usually refers (e.g., [18] in the context of branch-and bound algorithms) to the first two approaches as *low* and *high level parallelization*, respectively, because a low level parallel implementation of an algorithm does not change the interactions between its various parts; hence, it is not intrinsically different from its sequential version, only faster. In this sense, the third option is similar to the first, and also qualifies as low level parallelism.

Crainic, Toulouse and Gendreau [6, 7] show, however, that this distinction becomes significantly more blurred in the context of tabu search. Indeed, the interactions among several tabu threads often yield a search pattern different from the one obtained by the simple sequential execution of the same tabu searches. Furthermore, these same interactions may change the relative order of execution of the various parts of a tabu search algorithm, or alter the contents of its tabu lists and memories. Hence, high level parallelism may be achieved by a careful implementation of parallel search threads, especially when asynchronous parallelization strategies are used.

Crainic, Toulouse and Gendreau [7] review the literature relative to parallel tabu search algorithms, and observe that few parallelization paradigms have yet been called for in the reported experiments. Indeed, synchronization seems to be the adopted norm, parallel computation being mostly used to evaluate moves or to accelerate restarting strategies [13, 2, 16, 17, 15]. The authors then proceed to propose a taxonomy based, on the one hand, on how the search space is partitioned and, on the other hand, on the control and communication strategies used in the design of the parallel tabu search procedures. Finally, the authors present results that indicate that several parallelization strategies, identified by the taxonomy but not yet found

in the literature, may be advantageously used to develop efficient parallel tabu search procedures.

From a software point of view, synchronization refers to a class of strategies (barriers, semaphores, monitors, guards, ... – some of these mechanisms may also be implemented at the hardware level) that delay, prevent or authorize limited access to code sections in the program. For example, the use of barriers means that all or a subset of the processes involved in parallel computations should reach a certain state (number of iterations, CPU time interval, etc.) before the next parallel computation step may start. Most synchronous parallel tabu search procedures proposed in the literature ([6, 13, 16, 1, 2], etc.) make use of such barrier synchronization mechanisms.

The introduction of synchronization points into parallel iterative search procedures is often motivated by a desire to enforce the sequential semantic of the algorithm during parallel execution, that is, to ensure that parallel computations display a deterministic behavior and a search trajectory similar to that of a sequential method. Yet, in most cases, this is achieved at a heavy price in algorithmic efficiency, since a significant number of processes are often idle waiting for other processes to complete their activities. This price is even higher for heterogeneous parallel systems, such as networks of workstations, where all processes wait for the one on the machine with the slowest operation cycle or the heaviest computation load. Consequently, to improve the algorithmic performance, various levels of asynchronism are introduced into the parallel procedure.

Our goal is to further explore these issues and, based on the ideas exposed in [7], to perform a more comprehensive analysis of asynchronous parallelization paradigms for tabu search.

The study encompasses the performance evaluation and comparison of several asynchronous parallel implementations of a tabu search procedure developed to solve multicommodity location-allocation problems with balancing requirements. Hence, we first briefly review the formulation of this problem and the sequential tabu search procedure originally proposed to solve it [5]. The description of the parallel procedures comes next, followed by the presentation of the experimental results. We conclude by identifying interesting parallelization strategies and promising research directions.

2 Model and Sequential Tabu Search Procedure

The multicommodity location-allocation problem with balancing requirements typically arises in the context of the medium term management of a fleet of heterogeneous vehicles (e.g., containers), where vehicle depots have to be selected, the assignment of customers to depots has to be established for each type of container, and the interdepot container traffic has to be planned to account for differences in supplies and demands in various zones of the geographical territory served by the company. One aims to minimize the total system cost: the cost to select and operate the depots, plus the transportation costs between customers and depots, plus the costs of the inter-depot vehicle movements required to balance supply and demand. The model is formulated as follows [3]:

$$\text{Minimize } Z = \sum_{j \in D} f_j y_j + \sum_{r \in P} \left\{ \sum_{i \in C} \sum_{j \in D} (c_{ijr} x_{ijr} + c_{jir} x_{jir}) + \sum_{j \in D} \sum_{k \in D} s_{jkr} w_{jkr} \right\}$$

subject to

$$\sum_{j \in D} x_{ijr} = O_{ir} \quad \forall i \in C, r \in P$$

$$\sum_{j \in D} x_{jir} = D_{ir} \quad \forall i \in C, r \in P$$

$$x_{ijr} \leq O_{ir} y_j \quad \forall i \in C, j \in D, r \in P$$

$$x_{jir} \leq D_{ir} y_j \quad \forall i \in C, j \in D, r \in P$$

$$\sum_{i \in C} x_{ijr} + \sum_{k \in D} w_{kjr} - \sum_{i \in C} x_{jir} - \sum_{k \in D} w_{jkr} = 0 \quad \forall j \in D, r \in P$$

$$x_{ijr}, x_{jir}, w_{jkr} \geq 0 \quad \forall i \in C, j \in D, k \in D, r \in P$$

$$y_j \in \{0, 1\} \quad \forall j \in D$$

where C, D and P represent the sets of customers, candidate depots and products (vehicles), respectively, and

$y = (y_j)$: Opening decision variables for depots;
$y_j = 1$ if depot j is open and 0 otherwise, $j \in D$;

x_{ijr}, x_{jir} : Flows of product $r \in P$ between customer $i \in C$ and depot $j \in D$;

w_{jkr}, w_{kjr} : Flows of product $r \in P$ between depots $j \in d$ and $k \in D$;

f_j : "Fixed" cost for depot $j \in D$;

c_{ijr}, c_{jir} : Unit transportation cost for product $r \in P$ between customer $i \in C$ and depot $j \in D$;

s_{jkr}, s_{kjr} : Unit transportation cost for product $r \in P$ between depots $j \in D$ and $k \in D$;

O_{ir} : Supply at customer $i \in C$ of product $r \in P$;

D_{ir} : Demand at customer $i \in C$ of product $r \in P$.

The formulation displays a rather interesting network structure. In particular, for fixed y variables, it becomes an uncapacitated multicommodity minimum cost network flow problem (UMCNF), a well known model for which efficient solution methods exist. We use this property to define the tabu search procedure, which is fully described and analyzed in Crainic et al. [5]. In the following we only summarize its main characteristics, illustrated in Figure 1.

The search space is defined with respect to the y depot decision variables that specify the *depot configuration*. For each configuration $\bar{y}$, the optimal values of the continuous variables $x^{\star}(\bar{y})$ and $w^{\star}(\bar{y})$, as well as $Z^{\star}(\bar{y})$ the corresponding value of the objective function, may be computed by solving an UMCNF. Through this space, the search strategy aims at (i) determining the good number of open depots, and (ii) finding the best configuration for a given number of open depots. This is achieved by combining a local search with intensification and diversification phases.

For local search, the neighbourhood of a given solution $\bar{y}$ is made up of all solutions that may be reached by one of the following moves:

- Add: Open a currently closed depot;
- Drop: Close a currently open depot;
- Swap: Open a depot, while closing another.

However, such a neighbourhood is usually too large. Hence, sampling is used to build a candidate list. Furthermore, the evaluation of all moves by solving an UMCNF would be too time consuming. Surrogate functions based on estimates of differences in objective function values are therefore used in most instances; the real value is however computed once a move is selected and implemented.

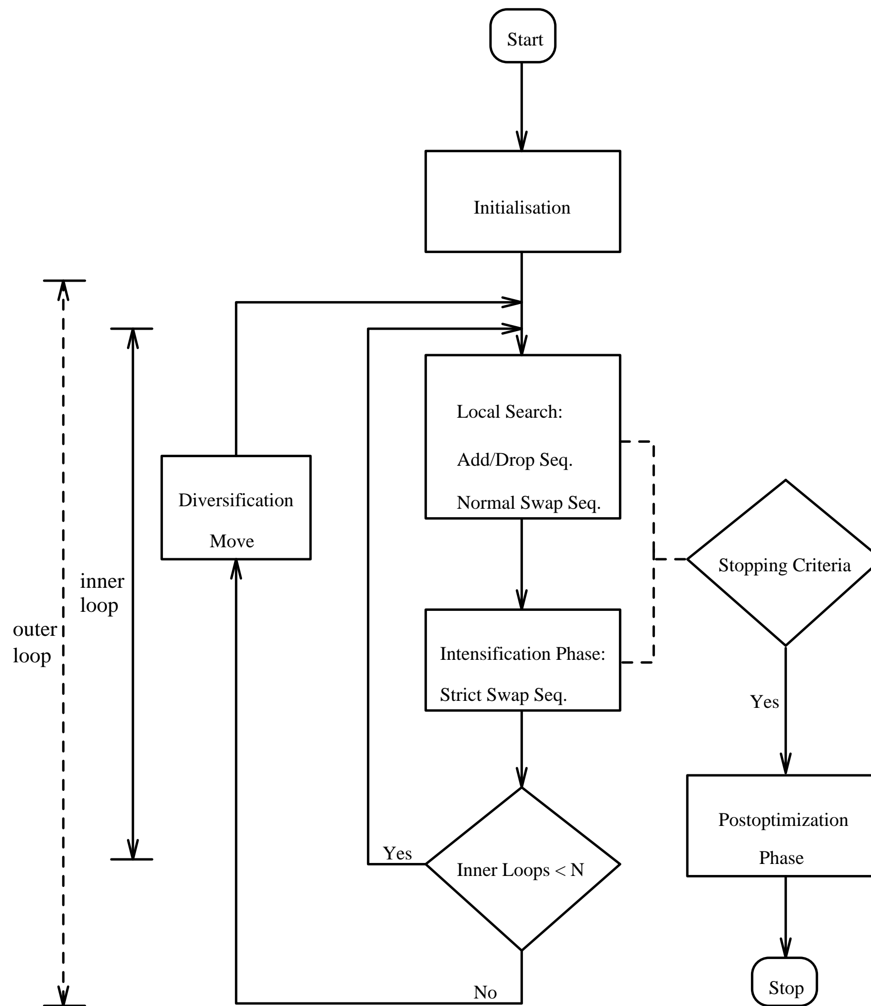


Figure 1: Sequential Tabu Search

Add/drop moves are used to change the cardinality of the examined depot configurations, while *swaps* allow to search for good configurations for a given cardinality. Swap sequences are further divided into *normal* and *strict* swap sequences, respectively. While performing the former, the best candidate move (evaluated by using the surrogate functions) is always implemented, regardless of its real impact on the economic function; it can therefore be non-improving. In strict swap sequences, however, the real cost of the selected move is evaluated, and the move is implemented only if it improves on the current solution.

Local search consists of an add/drop sequence followed by a normal swap sequence that is initiated from the best solution found by performing the add/drops. This combined add/drop-normal swap sequence yields a best local solution, which is either feasible or infeasible. If it is feasible, search intensification is immediately performed, by executing a strict swap sequence starting from this best local solution. If, on the other hand, the best local solution is infeasible, another add/drop and normal swap sequence is performed, starting from the last solution found by the previous normal swap sequence. This process is repeated until a feasible local solution is encountered.

Add/drop and swap sequences use different short-term memory tabu lists. For add and drop moves, lists record the last depots added or dropped from the solution, and the reverse moves are forbidden. The swap tabu list records the most recently performed swaps as pairs of depots, and the reversal or repetition of the moves is forbidden. Note that long term (diversification) tabu lists further affect the status of candidates while performing local search.

A sequence of local search and intensification phases is called an inner loop. After executing N inner loops, the search procedure is re-directed to previously unexplored regions of the search space by performing a diversification step.

Diversification is performed starting from the best global solution found up to this point in the search, and is based on a long-term memory that records the level of "activity" of each depot: the number of times its status has been modified (changed from open to closed or vice-versa). Based upon the values stored in this memory, the η depots with the lowest activity counts are selected and complemented. Considering the fact that values in the long-term memory tend to evolve rather slowly, another memory has been provided to record the last set of depots selected for diversification. This list is used both to exclude depots from being considered in the next diversification phases, and to prevent too quick a reversal of the diversification moves during the following local search steps.

N inner loops followed by a diversification phase make up an outer loop. The overall search procedure starts from an initial solution and performs a sequence of

outer loops until some termination criterion is met. In the current implementation, this termination criterion is the total number of iterations since the beginning of the search.

A postoptimization phase, which aims at ensuring that no better solution exists close to the best solution identified so far, is invoked once the prespecified number of regular iterations has been performed. This phase consists in a comprehensive neighbourhood exploration search that considers all possible simple (add, drop) moves. Surrogates are used to rank moves, while exact evaluations determine the (first) improving move to be implemented. This procedure continues for as long as strictly better solutions are found.

Several parameters influence the efficiency of the search: the lengths of the tabu lists and memories, the lengths of the add/drop and swap sequences, the selection probabilities of the add, drop and swap moves, how these probabilities vary during the search, the initial solution that is chosen, etc. Crainic et al. [5] study these issues and show, in particular, that several combinations of parameters may be efficiently used for different types of problem characteristics.

3 Asynchronous Tabu Search Procedures

We have designed several asynchronous parallel tabu search variants of the sequential tabu search procedure presented in the previous section, principally characterized by the amount of global information available, and the number of different starting points and search strategies used.

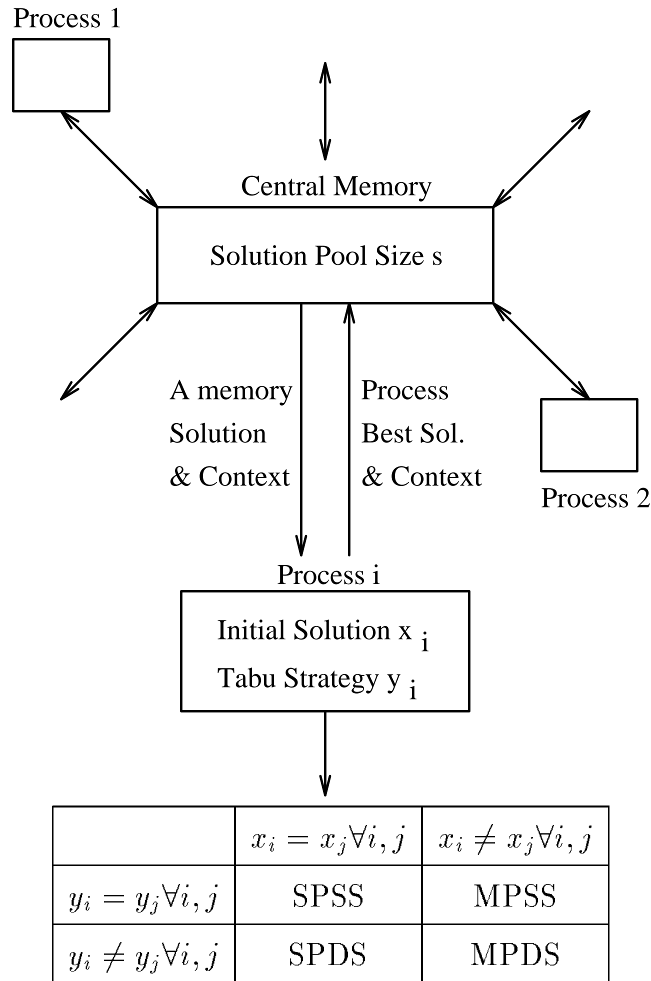
Let the *context* of a solution be a vector of cardinality n that contains the values of the n decision variables of the problem. There could be several contexts for a given objective function value but, given a context, there is only one possible objective function value. The asynchronous parallel framework that we analyze belongs to the p-control collegial class of the taxonomy proposed by Crainic, Toulouse and Gendreau [7], and is illustrated in Figure 2: here, p independent tabu search threads (processes) explore the problem domain, each informing the others when it improves its best global solution. The asynchronous type of parallelization implies that communications are initiated by an individual process according to its own internal logic and timing, and do not depend upon any global condition that would affect all threads. Therefore, in theory, at any moment during the search, the information that is available and used by each thread is different from the one used by any other thread. This is in sharp contrast to synchronous parallel approaches.

The procedures make use of a *central memory* through which pass all communications, and that captures the global knowledge acquired during the search. Note that this is an implementation device which helps to keep in check communication and accounting efforts, as compared, for example, with a strategy where each process broadcasts its solution to all other processes, which, in turn, have to accept, compare, update, and store the information. It also enforces the asynchronous paradigm, since it lifts the need for an acceptance decision by each process at broadcast time: each thread decides to access the central memory information based exclusively on its own internal schedule and history.

The *basic* mechanism proceeds as follows:

- The central memory keeps and updates the best global solution found so far. Together with its associated context, we call it the *central memory best solution.*
- Each process sends its solution and context to the central memory each time it improves its best global solution.
- If the best global solution of a process is worst than the central memory best solution, the process recuperates the central memory best solution.

- After a certain number of iterations without improving its global solution, a process requests the best central memory solution.



	$x_i = x_j \forall i, j$	$x_i \neq x_j \forall i, j$
$y_i = y_j \forall i, j$	SPSS	MPSS
$y_i \neq y_j \forall i, j$	SPDS	MPDS

Figure 2: Asynchronous Parallel Framework

Note that a process always resumes its computations from the same state (short term tabu lists and memories, best local search solution, current solution context, etc.) it was in just before communicating with the central memory. However, before initiating a diversification phase, the process compares and, eventually, replaces its best global solution and context by the central memory best solution. Then, either the search resumes from the central memory best solution (this amounts to an externally imposed diversification), or a normal diversification step is performed.

We have also implemented a modified version of the basic mechanism that increases the amount of information stored into the central memory, and introduces a certain degree of randomness into the precise information which is returned from the central memory to each individual process. The two main differences between this *solution pool* strategy and the basic one are:

- The central memory keeps and updates a list of the best s solutions, and associated contexts, ever found by the individual processes; the central memory best solution obviously belongs to this list.
- When a process requests the central memory solution, it does not automatically get the best one; it rather receives a solution randomly chosen from the s solutions in the list.

Note that in the solution pool framework, a process may receive several times the same solution. Hence, when it receives a central memory solution (i.e., one that is better than its own best global one), the process accepts it only if it is different from the improving central memory solutions it received previously. Note also that no infeasible solutions are exchanged among processes.

For the two strategies, we developed implementations that differ in their use of different or similar initial solutions, and of different or similar search strategies for the individual tabu threads. We did not implement the Single (Initial) Point Single Strategy (*SPSS*) [7] since it does not make much sense in the present context: it reduces to p repetitions of the same search, the only variations being introduced by the differences in speed of the various processors. We have developed, however, implementations for the Single Point Different Strategies (*SPDS*), Multiple Points Single Strategy (*MPSS*), and Multiple Points Different Strategies (*MPDS*) approaches.

Various elements may define a tabu search strategy: the parameter settings, the type of tabu lists and memories and their updating mechanisms, the existence and design of search intensification and diversification phases, etc. In order to facilitate comparisons between the sequential procedure and the parallel versions, we have decided to use for all parallel implementations the same basic search mechanism defined for the sequential version, and only modify the values of a number of key parameters. For the MPSS approach, the best parameter settings for the sequential tabu search are used. When different search strategies are implemented, in SPDS and MPDS approaches, we vary the lengths of the short and medium term tabu lists, the number of consecutive add/drop iterations without improvement, and the number of depots temporarily fixed by a diversification move. An initial solution is obtained by arbitrarily closing a number of depots; different initial solutions may thus

be obtained by varying this number. The calibration results for the sequential tabu search procedure [5] are used to define the required initial points and tabu search variants.

For comparison purposes, we also consider three methods that use different strategies to synchronize p search threads, each implementing the best sequential tabu search settings, that start from various initial points. These procedures are representative of their respective parallelization strategies, and have been identified in another study [6] as ensuring best performances on the class of models considered in this paper.

The first approach, that we identify as p-RS MPSS [7], and which is often referred to in the literature, consists simply in running p separate search threads, without any communication among processes during the search. The best solutions of the individual threads are then collected and compared to identify the best global solution.

The second synchronous approach, called p-KS MPSS, implements a communication phase amongst the p processes at regular intervals. During this synchronization phase, each process communicates its list of best solutions. The lists are then merged and sorted, and the p best solutions (and contexts) are returned to the individual search threads. For efficiency reasons, a central memory mechanism is used for the merging, sorting and solution distribution operations.

Finally, the third strategy, identified as 1-KS SPSS, consists in a master-slave approach. Here, a master process executes a sequential tabu search by using the other processors to perform computing intensive tasks, and to explore more thoroughly the neighbourhood. There is no communication among slave processes; subordinate processes exchange information only with the master process, which initiates communications to distribute work and gather results. The master divides the evaluation of the N elements of the candidate list of its local search phase among the slave processes. Each slave then evaluates N/p moves (no sampling) by using a surrogate function, according to the tabu lists handed out by the master process, and then performs a few local search iterations. The master then selects the sequence of moves that has resulted in the best improvement, and implements it by appropriately updating the various tabu lists and memories.

For all parallel implementations, the postoptimization phase is performed by each individual process, starting from its own best solution.

4 Experimental Results

Although the parallelization approaches described in the previous section can be implemented on different computing platforms, they are particularly suited to coarse grain, message passing parallel architectures. Hence, all tests have been conducted on a heterogeneous network of SUNSparc workstations. Communications are handled by our own set of procedures, written in C, that use the TLI/UDP protocol, modified to ensure that all packets reach their destination. The tabu search is programmed in FORTRAN77, while the minimum cost network flow subproblems are solved by using the RNET code [14].

The experiments aim to explore the design alternatives implied by the information management structures described in the previous section, and to examine the impact of some important design parameters, number of processors, pool size, etc., on the behaviour of the algorithms. These results are also used to address the issues of what parallelization approach appears to perform the best, and of how asynchronous parallelization strategies fare when compared to more classical synchronous ones.

Note that, our objective is to compare parallelization strategies, not to fine-tune a given procedure on a given set of problems. Hence, in order to facilitate the comparisons, we did not calibrate each individual procedure for best performance over the problem set. Instead, we use the best parameter settings observed for the sequential tabu search [5] for all the experiments reported in the remainder of this section. Furthermore, all procedures are stopped on a number of iterations. Hence, the"wall clock" time of the parallel procedures is similar to the total time of the sequential version (in order not to overload the paper, these figures are not indicated here). Note, however, that the total computing effort of the parallel procedures is larger than that of the sequential version by a factor related to p, the number of processors used. Hence, it is interesting to verify if the increase in computing power is rewarded by an improvement in the behaviour of the algorithm and the quality of the final solution.

Sixteen test problems are used: twelve (P1 to P12) randomly generated, and four (P13 to P16) based on an actual application [4]. Problems P7 to P12 are similar to problems P1 to P6 except that fixed costs have been multiplied by 10. Similarly, problems P14 to P16 have been obtained by multiplying the fixed costs in problem P13 by 10, 100 and 1000, respectively. The problem dimensions are indicated in Table 1.

Tables 2, 3, and 4 summarize the results obtained by the three search differentiation strategies, SPDS, MPSS, and MPDS, by using each one of the two knowl-

Problem	1/7	2/8	3/9	4/10	5/11	6/12	13-16
Depots	44	44	43	44	44	44	130
Customers	219	219	220	219	219	220	289
Products	1	2	1	2	1	2	12
Customer Links	5260	10520	5282	10516	5262	10588	45936
Location Links	1892	3784	1892	3784	1892	3612	10680

Table 1: Problem Dimensions

edge exchange mechanisms: the basic central memory approach, and the solution pool strategy. Procedures are stopped after 300 iterations (plus the postoptimization phase). Results are reported for $p = 4$, 8 and 16 processors, and are also graphically illustrated in Figures 3, 4 and 5. The tables display the following information:

- The *Gap*, in percentage, between the best solution determined by each procedure and the optimal solution. Optimal solutions have been computed by a branch-and-bound algorithm [8].

- When the best solution is found during the postoptimization phase, we indicate, in between parentheses, the corresponding gap of the solution from which the postoptimization phase has been initiated.

- The iteration (*Iter*) at which the best solution was found. When the best solution is found during the postoptimization phase, we indicate the iteration corresponding to the solution from which this phase has been initiated.

- The gap information (*Seq*) corresponding to the solution determined by using the sequential procedure.

- The size s of the pool which equals the number p of processors used.

Note that the results of the sequential procedure used in this paper are generally better that those reported in [5]. This improvement results from an observation made during the development of the parallel implementations: it appears that, due to the limited number of iterations that we use, it is more efficient not to allow the search to explore unfeasible solutions. Note also that the sequential procedure has been calibrated by using eight out of the twelve problems P1 to P12.

Prob	p = 4		p = 8		p = 16		Seq
	Gap	Iter	Gap	Iter	Gap	Iter	
	Basic Central Memory Mechanism						
P1	0	18	0	47	0	165	0
P2	0.54	286	0.14 (0.62)	290	0.20 (0.37)	299	0.42 (1.48)
P3	0.01	113	0.01	45	0	8	0.01 (0.8)
P4	0 (0.13)	290	0 (0.06)	299	0 (0.01)	290	0.9 (1.1)
P5	0.33	233	0.64	285	0	93	0.77 (1.45)
P6	0.01 (1.01)	297	0	278	0.01	241	0.42 (3.67)
P7	0	247	0	182	0	31	0.07
P8	0.85	137	0	185	0	228	2.15
P9	0	243	0	101	0	151	0
P10	0.3	242	0.17	15	0	158	1.49
P11	0	175	0	127	0	110	1.52
P12	0.11	13	0.11	127	0	10	0.11
P13	0.002 (0.12)	297	0.002 (0.04)	296	0 (0.03)	217	0.002 (0.07)
P14	0.02 (0.21)	266	0.04 (2.04)	223	0.02 (0.2)	172	0.02 (0.59)
P15	0.59 (0.71)	290	0.73 (1.06)	295	0.26 (0.6)	270	1.23 (1.54)
P16	0.23	226	0.89	241	0.14	250	0.89 (9.21)
	Pool Strategy $s = p$						
P1	0	109	0	125	0	18	0
P2	0.43 (0.67)	219	0.14 (0.95)	290	0.16 (0.29)	145	0.42 (1.48)
P3	0	301	0	198	0	270	0.01 (0.8)
P4	0.29	247	0.11 (0.75)	287	0.14 (0.79)	232	0.9 (1.1)
P5	1.0 (1.39)	141	0	238	0 (1.73)	287	0.77 (1.45)
P6	0.39 (1.16)	142	0 (0.63)	158	0 (0.91)	204	0.42 (3.67)
P7	0	190	0	242	0	94	0.07
P8	0	188	0	80	0	213	2.15
P9	0	194	0	253	0	130	0
P10	0.37	298	0.38	228	0.14 (3.08)	226	1.49
P11	0	268	0	135	0	233	1.52
P12	0.11	205	0.11	17	0	242	0.11
P13	0.002 (0.07)	261	0.002 (0.09)	298	0.002 (0.12)	75	0.002 (0.07)
P14	0.06 (0.76)	298	0.06 (0.18)	252	0 (1.12)	225	0.02 (0.59)
P15	0.19 (0.52)	240	0.73 (0.75)	287	1.22 (5.78)	293	1.23 (1.54)
P16	0.23	267	0.11	285	0.89 (9.1)	297	0.89 (9.21)

Table 2: p-C SPDS Results

The first general conclusion which emerges from this experiment is that all asynchronous parallel versions perform generally better, and most of them significantly so, than the sequential algorithm. Compare, for example, the 0.63% average gap of the sequential procedure, to the 0.17%, 0.12%, and 0.03% average gaps for the SPDS approach (without pool), with respectively 4, 8, and 16 processors. Furthermore, asynchronous parallel tabu search procedures help reach high quality solutions. Indeed, even with basic parameter settings, and even without performing the postoptimization phase, either the optimal solution is found or a very low gap exists between the optimum and the proposed solution. Moreover, once notices the robustness of the asynchronous parallel approach: all procedures generally find better solutions than the sequential version, and, usually, in a smaller number of iterations. Hence, for a given allocation of ("wall clock") time, parallelization allows one to reach better solutions.

Before proceeding further to analyze the results and compare the various strategies, a few remarks are in order relative to their particular implementations. The first concerns the definition of the Single (Initial) Point Different Strategies approach. It is easy to enforce a SPDS rule in a synchronous framework: at each synchronization step, all processes receive the same solution. When asynchronous implementations are contemplated, however, it is not readily apparent that a strict SPDS paradigm may be easily applied, especially when the solution pool strategy is used. In fact, since one does not know when processes communicate, one cannot ensure that they are all working from the same solution. To partially overcome this problem, we modified the method in such a way that when an (central memory) improving solution is accepted by an individual process, it replaces the best local solution right away. We thus ensure some sort of "local" SPDS framework for a subset of processes that happen to communicate at about the same time (we use this approach for the experiments reported in this paper), at the cost of possibly disrupting the normal search pattern.

A stricter implementation of the SPDS strategy could be achieved if, for example, once an improving solution is obtained by an individual process, this same solution is kept unchanged and is used for the next $p-1$ exchanges. This approach presents some obvious implementation difficulties: for example, one has to be careful in case the same process communicates several times consecutively with the central memory, otherwise not all processes will get the same (re)starting point. More important, however, such approaches contradict the objective of the solution pool strategy which aims to diversify the search by sending good but different solutions to the various processes. Furthermore, if there may be very good reasons to start all processes from the same initial solution (e.g., problems P13 to P16 where the only solution that appears to function well starts the search with all depots open), to constrain the asynchronous search threads into accepting the same solution by using one of these

Prob	p = 4		p = 8		p = 16		Seq
	Gap	Iter	Gap	Iter	Gap	Iter	
	Basic Central Memory Mechanism						
P1	0	20	0	20	0	19	0
P2	0.21 (0.23)	297	0.39	298	0.14 (0.55)	20	0.42 (1.48)
P3	0 (0.01)	298	0	145	0	191	0.01 (0.8)
P4	0.23 (0.30)	278	0 (0.06)	192	0.27 (0.41)	298	0.9 (1.1)
P5	0.33 (0.76)	288	0.33	274	0.33	277	0.77 (1.45)
P6	0.15 (0.19)	278	0.15	300	0.15 (0.19)	295	0.42 (3.67)
P7	0	184	0.07	28	0	49	0.07
P8	0	225	0	284	0	134	2.15
P9	0	146	0	235	0	178	0
P10	1.96	296	0.37	137	0.74	200	1.49
P11	0	117	0	45	0	47	1.52
P12	0.11	85	0	234	0	17	0.11
P13	0.002 (0.49)	255	0 (0.09)	299	0.002 (0.03)	250	0.002 (0.07)
P14	0 (1.5)	135	0 (1.5)	235	0.02 (0.21)	299	0.02 (0.59)
P15	0.59 (0.71)	275	0.26 (0.29)	290	0.59	299	1.23 (1.54)
P16	0.89	260	0.89	261	0.44	281	0.89 (9.21)
	Pool Strategy $s = p$						
P1	0	181	0	90	0	17	0
P2	0.44 (1.49)	210	0 (1.0)	299	0.16 (0.27)	264	0.42 (1.48)
P3	0 (0.32)	295	0	114	0	79	0.01 (0.8)
P4	0.5 (1.02)	300	0.35 (0.68)	219	0 (0.43)	258	0.9 (1.1)
P5	0.38 (0.58)	243	0	236	0	63	0.77 (1.45)
P6	0.15 (0.17)	288	0.05 (1.19)	275	0.01 (0.75)	295	0.42 (3.67)
P7	0	227	0	231	0	275	0.07
P8	0	93	0	234	0	85	2.15
P9	0	277	0	159	0	153	0
P10	1.16 (2.76)	202	0.14 (1.46)	257	0.30	83	1.49
P11	0	205	0	150	0	141	1.52
P12	0.11 (1.34)	227	0 (0.23)	295	0	17	0.11
P13	0 (0.48)	202	0.002 (0.44)	277	0.002 (0.49)	297	0.002 (0.07)
P14	0.06 (0.75)	201	0.04 (0.32)	183	0 (0.39)	128	0.02 (0.59)
P15	0 (0.07)	296	0	284	0.03 (0.54) (0.03)	296	1.23 (1.54)
P16	0.02 (3.08)	278	0.44 (3.37)	267	0.89	256	0.89 (9.21)

Table 3: p-C MPSS Results

mechanisms, is equivalent to reintroducing a subtle form of synchronization. This is reflected in the fact that the pool strategy appears to perform slightly less well in an SPDS context.

The second issue concerns the behaviour of the multiple initial points, MPSS and MPDS, strategies. In general, these paradigms are relatively easy to maintain in an asynchronous parallel environment, since each process enquires the central memory at unforeseen moments, independently of the other processes. It is particularly easy in the context of the solution pool strategies where not only the set of available solutions is modified at random moments, but also the solution selected by a particular process is randomly chosen within this set. The case of the basic central memory strategy is less straightforward, since only one solution, the current best, is available at any given time, and it is updated only when an individual search thread finds an improving one. Hence, because the central memory solution is updated less frequently as the global search advances, there exists the possibility that MPSS and MPDS searches become, eventually, SPSS and SPDS searches, respectively. The fact that, when 16 processors are used, the average gaps obtained by using the MPDS and SPDS strategies are the same seems to point towards this conclusion.

Two observations qualify, however, this last point. Note, on the one hand, that in the case of an MPDS approach, the different search strategies still ensure a wider coverage of the problem domain. This may indeed contribute to explain the superiority, apparent in Figures 3 and 4, of MPDS over MPSS. On the other hand, when comparing MPDS and SPDS solutions, one notices that when the optimal solution is not identified for a given problem, different solutions are generally reached, and in different numbers of iterations. This is especially interesting when the case of problems P13 to P16 is considered, since here all search threads start with the same initial solution, the multiple starting point (MPSS and MPDS) paradigm being enforced only through the management of the central memory.

We may thus infer that the search trajectories of the two parallelization strategies, MPDS and SPDS, are different, and that the conversion phenomenon of multiple starting points strategies, MPSS and MPDS, into single point ones, SPSS and SPDS, respectively, is not very present in our experiments. It also shows that careful algorithmic design and the unpredictability of communications inherent to asynchronous parallelization (especially in a heterogeneous computing hardware context) can enforce a parallelization strategy that works toward a broader exploration of the search space.

We have run the procedures on 4, 8 and 16 processors for 300 iterations, and the results are graphically synthesized in Figures 3 and 4 which display the evolution of the average gap for problems P1 to P12, and for the whole set of problems, respectively.

Prob	p = 4		p = 8		p = 16		Seq
	Gap	Iter	Gap	Iter	Gap	Iter	
	Basic Central Memory Mechanism						
P1	0	119	0	119	0	33	0
P2	0.32 (0.350	275	0 (0.42)	253	0.06	217	0.42 (1.48)
P3	0	73	0	72	0	73	0.01 (0.8)
P4	0.23 (0.56)	296	0	297	0	254	0.9 (1.1)
P5	0	292	0	53	0	188	0.77 (1.45)
P6	0.15 (1.24)	243	0.05 (0.93)	256	0.01 (0.02)	297	0.42 (3.67)
P7	0.07	223	0.07	154	0	216	0.07
P8	0	65	0	272	0	81	2.15
P9	0	149	0	240	0	61	0
P10	0	99	0.14 (1.79)	286	0 (2.72)	297	1.49
P11	0	289	0	135	0	72	1.52
P12	0	33	0	33	0	33	0.11
P13	0.002 (0.02)	276	0.002 (0.07)	292	0 (0.03)	244	0.002 (0.07)
P14	0.02 (0.2)	285	0.06 (1.0)	284	0.02	290	0.02 (0.59)
P15	0.55 (0.81)	263	0.26 (0.29)	291	0.26 (0.6)	275	1.23 (1.54)
P16	0.23	226	0.23	284	0.23	223	0.89 (9.21)
	Pool Strategy $s = p$						
P1	0	255	0	103	0	21	0
P2	0.21 (1.87)	165	0.07 (0.15)	163	0.21 (0.3)	256	0.42 (1.48)
P3	0	201	0	106	0	75	0.01 (0.8)
P4	0.05	228	0.23 (0.3)	240	0.05 (0.55)	235	0.9 (1.1)
P5	0.38 (0.13)	210	0	122	0	116	0.77 (1.45)
P6	0.05 (0.6)	256	0 (0.12)	257	0.05 (0.69)	196	0.42 (3.67)
P7	0	255	0.07	110	0	264	0.07
P8	0	280	0	127	0	217	2.15
P9	0	266	0	54	0	283	0
P10	0.17 (1.53)	211	0.42 (1.01)	272	0	225	1.49
P11	0	281	0	186	0	240	1.52
P12	0	57	0	33	0	33	0.11
P13	0.002 (0.07)	221	0.002 (0.05)	300	0.002 (0.06)	200	0.002 (0.07)
P14	0.02 (0.6)	299	0.06 (0.44)	256	0.06 (0.29)	263	0.02 (0.59)
P15	0.59 (0.81)	253	0.26 (0.29)	266	0	290	1.23 (1.54)
P16	0.23	226	0.23	218	.23	256	0.89 (9.21)

Table 4: p-C MPDS Results

In the first case, problems P1 to P12 form a homogeneous set, having been generated by using the same procedure and parameter settings. Furthermore, the sequential tabu search procedure has been calibrated by using eight of these twelve problems. Problems P13 to P16, on the other hand, are significantly different, both as compared to the other twelve, and as compared one to another (especially due to large differences in fixed to variable cost ratios). Consequently, computing averages over the whole problem set does not offer a very meaningful measure. We give the two graphical displays, however, because it emphasizes the robustness of the parallel algorithms. While the procedures display a more similar behaviour and a better performance on the restricted problem set (this was to be expected), the overall performance is excellent. In general, all parallelization schemes consistently allow to reach very good solutions, the average gaps falling within a very narrow interval.

With respect to the influence of the number of processors on the solution quality, our experiments indicate that increasing this number is beneficial: better solutions are found, in general, when using 8 or 16 processors than when using only 4. The strategies that appear to take the most advantage of an increase in the number of processors (as measured by the average gap) are MPDS, SPDS, and MPDS with the solution pool mechanism. Furthermore, we have also run the MPDS strategies for 600 iterations on 4 processors, without improving on the performance of the same algorithm run for only 300 iterations on 8 processors. Further study should better clarify this issue, but our initial experiments suggest that increasing the number of processes is more beneficial than simply increasing the number of iterations. This follows from the asynchronous nature of the procedures, which ensures that when more processors are used, more sub-regions of the solution space are explored simultaneously.

We also observe, however, that for tabu search, as noticed for other classes of algorithms, there are limits on the efficiency of throwing more processors at a given problem. Generally, the average gap either decreases slightly or even increases, when one passes from 8 to 16 processors. This last phenomenon may be explained by the fact that for the size of problems we consider, 16 different threads imply that some initial points and search strategies are quite far from those identified as "good" for the sequential tabu search (the parallel versions were not calibrated). This, and the increased randomness introduced by the doubling of the number of processors, may have the search be attracted toward less interesting regions of the search space. Hence, it would appear that 8 processors forms an adequate computing force for the size of problems we study. This conclusion is also corroborated by the behaviour of the average number of iterations required to reach the best solution, where going from 8 to 16 processors is hardly justified.

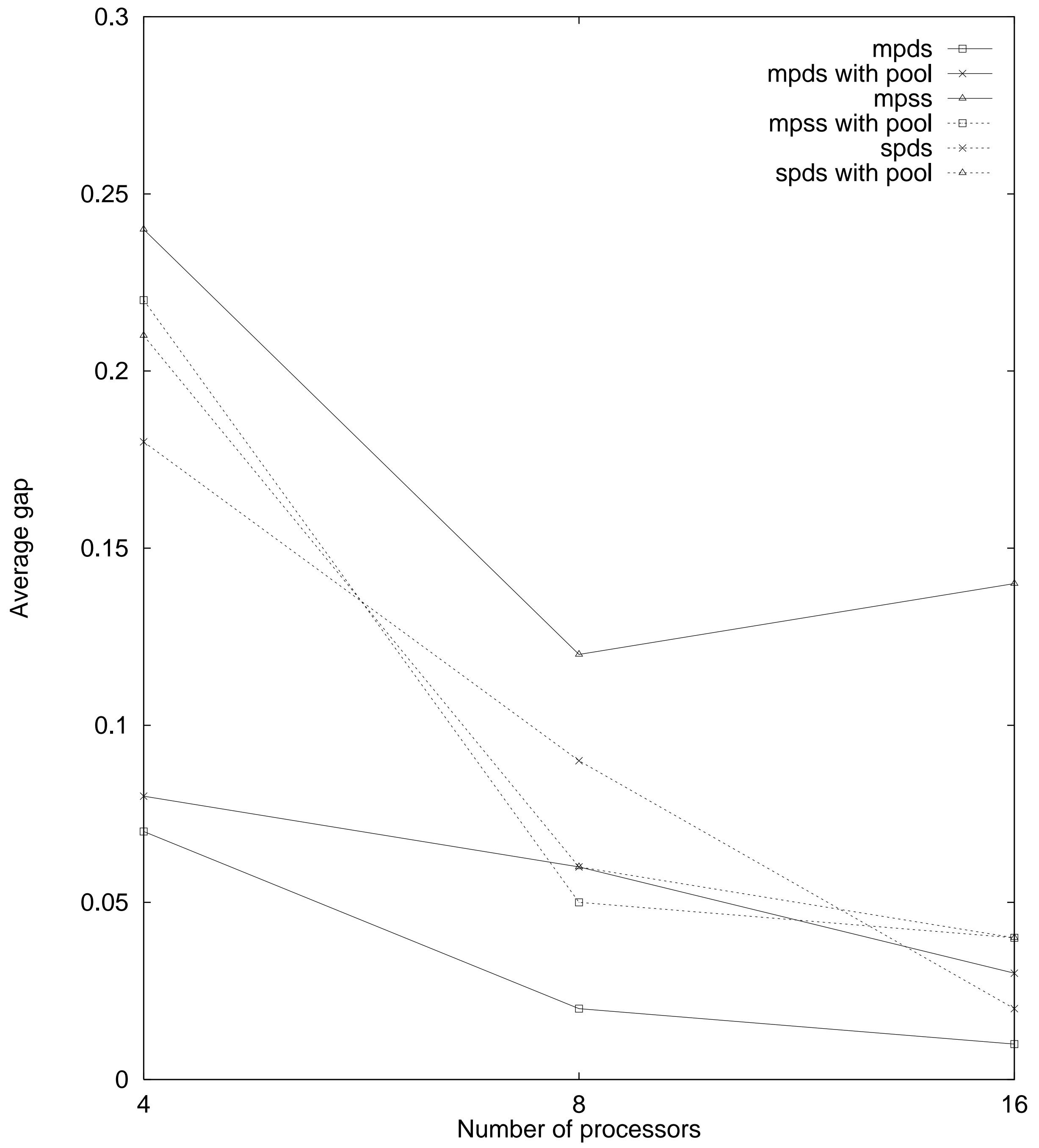


Figure 3: Gap Comparisons on Problems P1 to P12

Concerning the quantity and management of the information kept in the central memory and exchanged among processes, our results suggest that using a pool of the s best solutions found so far, and randomly allocating one of these to a particular process, increases the number of solutions that are examined and may help find better solutions. This is particularly encouraging, especially since no tuning has been performed on the procedures, and only a very simple acceptance mechanism (a solution is accepted if it is better than the worst in the pool) has been used. Finally, the procedures appear quite robust relative to the size of the pool kept by the central memory. In all the experiments reported in this paper, the size of the pool is fixed to the number of available processors. We performed a number of additional experiences with different pool sizes, and while performances varied for some individual problems, the general behaviour of the procedures did not.

Among the strategies tested, MPDS and SPDS seem to perform better, with respect to both the average gap and the effort required to reach the best solution, with MPDS slightly ahead, especially when 8 processors are used. In fact, it is interesting to note that asynchronous parallel tabu search appears remarkably robust in achieving high quality solutions for a rather wide gamut of algorithmic strategies and parameter settings. It is also interesting to compare their performance with that of synchronous implementations of the same algorithm.

The comparisons are illustrated in Figure 5, which displays the evolution with the number of processors of the average gaps obtained, on the same set of problems, P1 to P12, by the sequential procedure, and by three synchronous algorithms that are both representative of their class and offered the best performance [6] (see Section 3: p-KS MPSS, p-RS MPSS (the search threads are synchronized every 25 iterations), and 1-KS MPSS (each slave performs two tabu add/drop iterations). Note that all synchronous approaches use only the best search strategy identified for the sequential case, and thus are significantly more tuned than the asynchronous ones.

The performance of the synchronous algorithms are compared to the those of the asynchronous MPDS, MPDS with pool, and SPDS with pool. The results illustrated in Figure 5 clearly show the superiority of the asynchronous approach. A more thorough exploration of the solution space, as compared with that achieved by sequential and synchronous parallel methods, due to the asynchronous implementation of the search is the most probable cause for this result.

The results presented above also emphasize one important fact: that in designing parallel tabu search algorithms, how the knowledge gathered during the parallel exploration of the domain is exchanged and combined among processes is at least as important as how the domain is divided among, or how the tasks are allocated to, the same processes. We believe that, while important for the design of any parallel

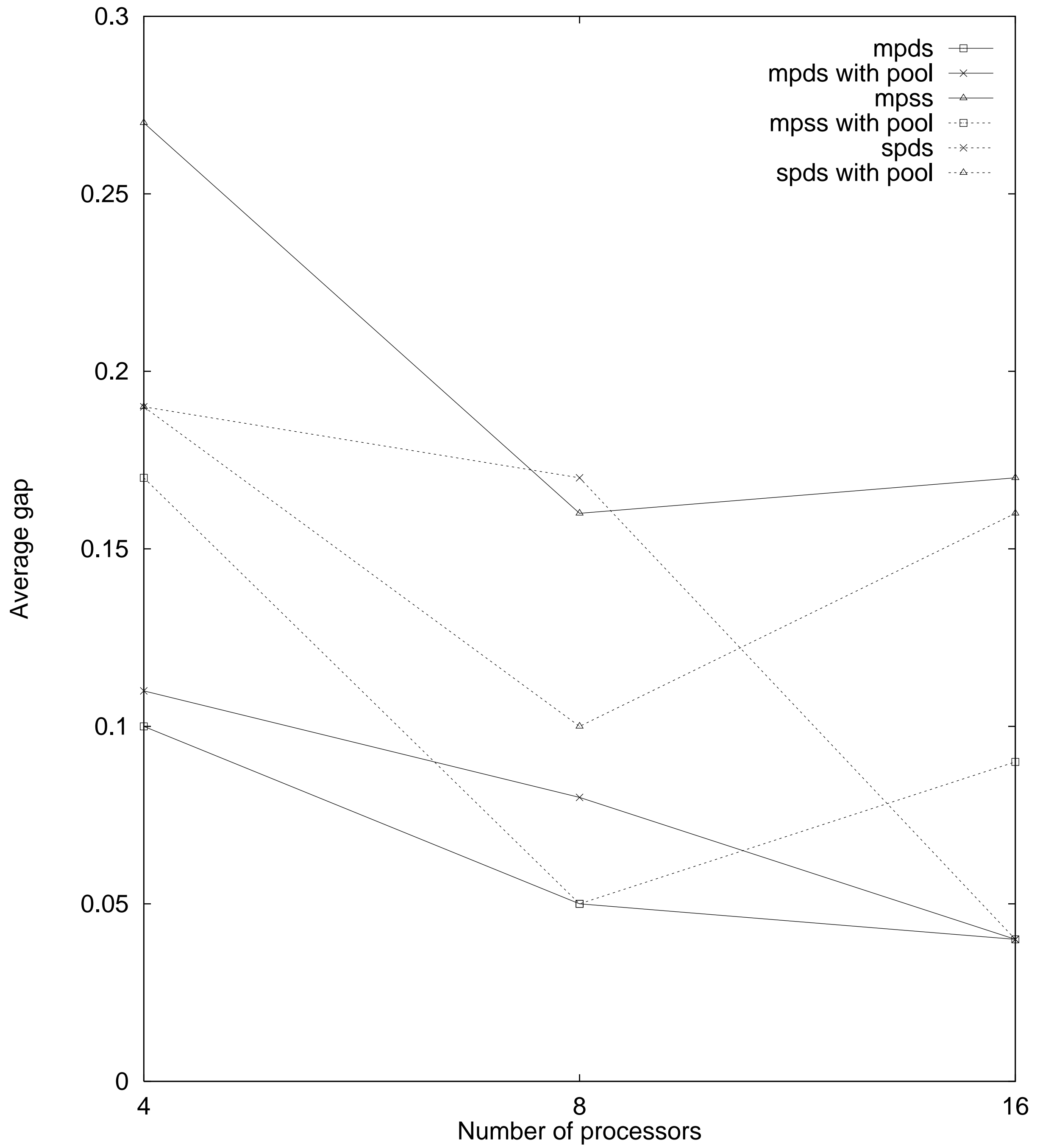


Figure 4: Gap Comparisons Between Asynchronous Implementations on all Problems

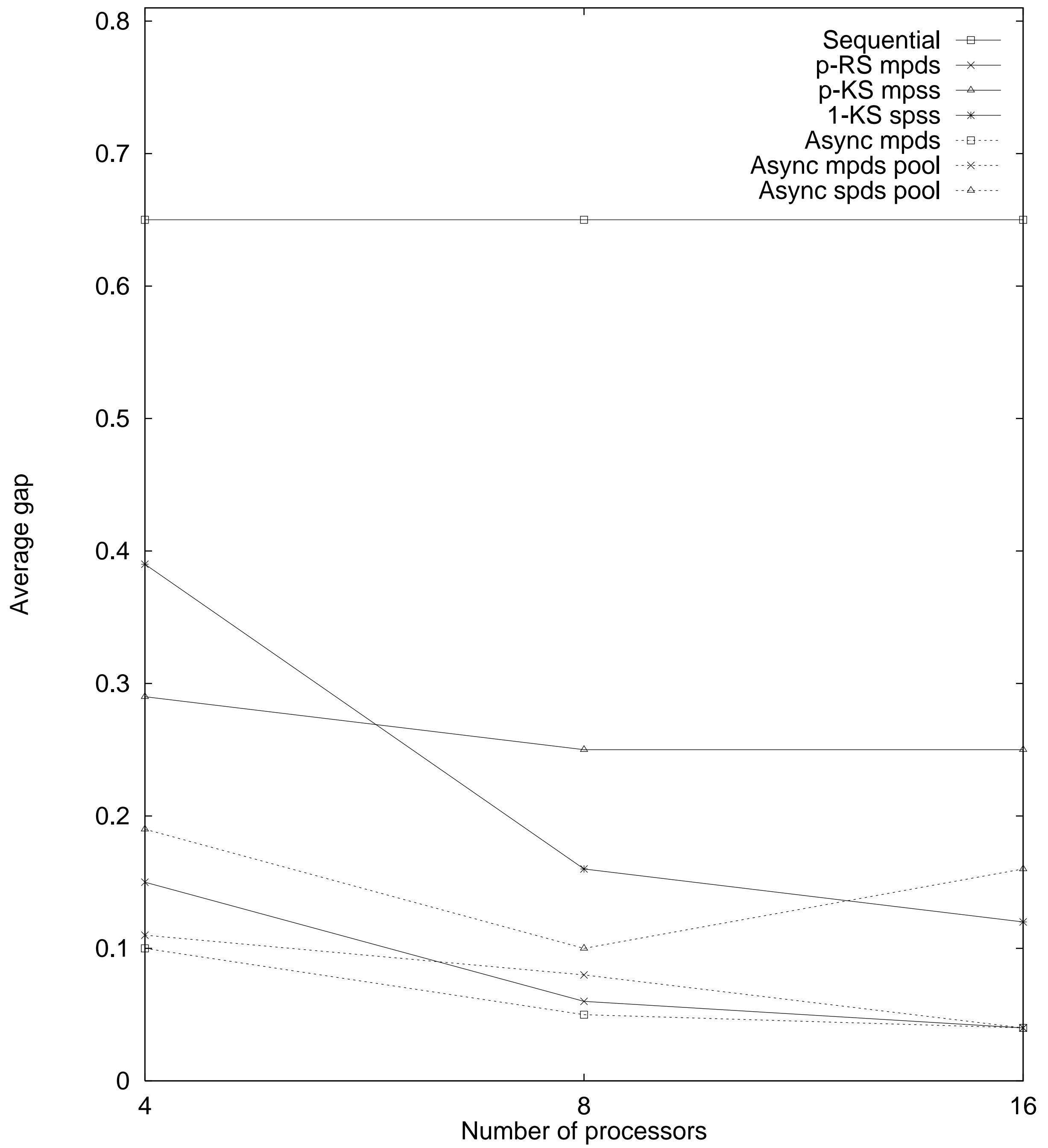


Figure 5: Gap Comparisons – Best Asynchronous and Synchronous Procedures

algorithm, this fact is particularly relevant for tabu search methods: on the one hand, tabu search makes extensive use of information gathered during the search, while, on the other hand, this information may not always be available to all processes sinvolved in parallel computation, or at least not in a way that comprehensively reflects the history of the global search. By this yardstick, asynchronous approaches offer richer search strategies than synchronous ones, but also require careful information handling.

5 Conclusions and Perspectives

We have presented a study of asynchronous parallelization strategies for tabu search. Several approaches have been compared and the influence of several important parameters on the algorithmic efficiency and the solution quality have been analyzed. Even though strictly speaking our results are valid only for the class and dimensions of the problems studied, we believe our conclusions to be valuable for tabu search methods in general.

We have shown that parallelization, and in particular asynchronous strategies, may be very profitable for tabu search. Furthermore, in general, asynchronous tabu search appears as a robust procedure: it achieves high quality results for a wide gamut of algorithmic strategies and parameter settings, and generally outperforms synchronous approaches.

Two particular conclusions are noteworthy. First, it has been shown that for asynchronous parallel tabu search, an increase in the number of available processors is generally beneficial. In fact, is appears more interesting to double, up to a point, the number of processors than the number of iterations. Second, increasing the level of accumulated knowledge about the state of the global search, as well as the degree of diversification of the search among the various threads, yields better performances. The mechanism we introduced, the central memory solution pool, achieves this quite nicely. It is also an indication that improved performances might be achieved by a more creative use of the information gathered by each process and exchanged among processes. We plan to explore these avenues in the near future.

Acknowledgments

This research has been supported by grants from the Fonds F.C.A.R. of the Province of Québec, and the Natural Sciences and Engineering Research Council of Canada. We would like to thank Mr. Pierre Bélanger for his assistance with the execution of the multiple runs required by this study.

References

[1] J. Chakrapani and J. Skorin-Kapov. A Connectionist Approach to the Quadratic Assignment Problem. *Computers & Operations Research*, 19(3/4):287–295, 1992.

[2] J. Chakrapani and J. Skorin-Kapov. Massively parallel tabu search for the quadratic assignment problem. *Annals of Operations Research*, 41:327–341, 1993.

[3] T.G. Crainic, P.J. Dejax, and L. Delorme. Models for Multimode Multicommodity Location Problems with Interdepot Balancing Requirements. *Annals of Operations Research*, 18:279–302, 1989.

[4] T.G. Crainic, L. Delorme, and P.J. Dejax. A Branch-and-Bound Method for Multicommodity Location with Balancing Requirements. *European Journal of Operational Research*, 65(3):368–382, 1993.

[5] T.G. Crainic, M. Gendreau, P. Soriano, and M. Toulouse. A Tabu Search Procedure for Multicommodity Location/Allocation with Balancing Requirements. *Annals of Operations Research*, 41:359–383, 1993.

[6] T.G. Crainic, M. Toulouse, and M. Gendreau. Synchronous Tabu Search Parallelization Strategies for Multicommodity Location-Allocation with Balancing Requirements. Publication 934, Centre de recherche sur les transports, Université de Montréal, 1993.

[7] T.G. Crainic, M. Toulouse, and M. Gendreau. Towards a Taxonomy of Parallel Tabu Search Algorithms. Publication 933, Centre de recherche sur les transports, Université de Montréal, 1993.

[8] B. Gendron and T.G. Crainic. An Exact Algorithm for Multicommodity Location with Balancing Requirements. Publication 843, Centre de recherche sur les transports, Université de Montréal, 1993.

[9] F. Glover. Tabu Search - Part I. *ORSA Journal on Computing*, 1(3):190–206, 1989.

[10] F. Glover. Tabu Search - Part II. *ORSA Journal on Computing*, 2(1):4–32, 1990.

[11] F. Glover. Tabu Search: A Tutorial. *Interfaces*, 20(4):74–94, 1990.

[12] F. Glover and M. Laguna. Tabu search. In C.R. Reeves, editor, *Modern Heuristic Techniques for Combinatorial Problems*. Blackwell Scientific Publications, London, 1993.

[13] M. Malek, M Guruswamy, M. Pandya, and H. Owens. Serial and Parallel Simulated Annealing and Tabu Search Algorithms for the Traveling Salesman Problem. *Annals of Operations Research*, 21:59–84, 1989.

[14] Grigoriadis M.D. and Hsu T. RNET – The Rutgers Minimum Cost Network Flow Subroutines. Technical report, Rutgers University, New Brunswick, New Jersey, 1979.

[15] E. Taillard. Robust Taboo Search for the Quadratic Assignment Problem. *Parallel Computing*, 17:443–455, 1991.

[16] E. Taillard. Parallel iterative search methods for vehicle routing problems. *NETWORKS*, 23:661–673, 1993.

[17] E. Taillard. Parallel taboo search techniques for the job shop sheduling problem. *ORSA Journal on Computing*, 6(2):108–117, 1994.

[18] H.W.J.M. Trienekens and A. de Bruin. Towards a taxonomy of parallel branch and bound algorithms. Report EUR-CS-92-01, Department of Computer Science, Erasmus University Rotterdam, 1992.