PARALLEL SCATTER SEARCH ALGORITHMS FOR EXAM TIMETABLING

by

GHIA ABDULLAH SLEIMAN HAIDAR

B.S., Computer Science, Beirut University College, 1996

Thesis submitted in partial fulfillment of the requirements for the Degree of Master of Science in Computer Science

Department of Computer Science and Mathematics

LEBANESE AMERICAN UNIVERSITY

June 2009

LEBANESE AMERICAN UNIVERSITY

Thesis Approval Form

Student Name: Ghia Abdullah Sleiman Haidar I.D.: 199205410

Thesis Title : Parallel Scatter Search Algorithms for Exam Timetabling

Program : M.S. in Computer Science

Division/Dept: Computer Science and Mathematics

School : Arts and Sciences - Beirut

Approved/Signed by:

Thesis Advisor	Dr. Nashaat Mansour	[redacted]
Member	Dr. Haidar Harmanani	[redacted]
Member	Dr. Abbass Tarhini	[redacted]

Date: June 29, 2009

Plagiarism Policy Compliance Statement

I certify that I have read and understood LAU's Plagiarism Policy. I understand that failure to comply with this Policy can lead to academic and disciplinary actions against me.

This work is substantially my own, and to the extent that any part of this work is not my own I have indicated that by acknowledging its sources.

Name: Ghia Abdullah Sleiman Haidar

Signature: 

Date: June 29, 2009

I grant to the LEBANESE AMERCIAN UNIVERSITY the right to use this work, irrespective of any copyright, for the University's own purpose without cost to the University or its students and employees. I further agree that the University may reproduce and provide single copies of the work to the public for the cost of reproduction.

To the person who was always with me

And will always remain in me

My father

Acknowledgment

I would like to express my sincere gratitude to Dr. Nashaat Mansour, my Advisor whose guidance and support throughout my thesis work made it all possible.

I would like to thank Dr. Haidar Harmanani and Dr. Abbas Tarhini for being in my thesis committee.

Finally, I would like to thank my family for their long support.

Abstract

University exam timetabling is an important activity for scheduling exams into predefined days, time periods and rooms. Given a set of constraints, exam timetabling is an NP-Hard problem that requires heuristic techniques to be solved adequately within reasonable execution time. For large numbers of exams and students, sequential algorithms are likely to be very time consuming. The purpose of this work is to design and implement a parallel scatter search meta-heuristic algorithm for producing good sub-optimal exam timetables in a reasonable time. Scatter search is a population-based approach that generates solutions over a number of iterations and aims to combine diversification and search intensification. We propose a parallel scatter search that is based distributing the population of candidate solutions over a number of processors in a cluster environment. The main components of scatter search are computed in parallel and an efficient communication technique is employed. Empirical results show that our proposed parallel scatter search algorithm yields good speed-up. Also, they show that the parallel scatter search algorithm improves solution quality since the algorithm explores larger parts of the search space within reasonable time.

Keywords: Meta-heuristics; parallel algorithm; scatter search; timetabling; exam timetabling.

Contents

List of Figures

List of Tables

Chapter 1
Introduction

1. Timetabling

Timetabling is the assignment of timeslots to events subject to defined constraints. In order to be able to allocate time and place to a specified event, we need to have sufficient resources of timeslots and rooms, as it is indicated by (Burke, Eckersley, McCollum, Petrovic, and Qu, 2005). Wren (1999) defined "timetabling as the allocation, subject to constraints, of given resources to objects being placed in space time, in such a way as to satisfy as nearly as possible a set of desirable objectives".

The University timetabling problem is of two types. These types are exam timetabling and course timetabling. In our work, we will concentrate on the first part, i.e. the exam timetabling problem.

2. Exam Timetabling

Exam timetabling consists of allocating dates, periods and rooms to a defined set of exams. In our case, we consider a limited number of periods (timeslots) and a limited number of rooms as resources for allocation. In a more general definition, the exam timetabling problem consists of allocating a sequence of exams a fixed period of timeslots and space resources, in a way to satisfy a set of constraints. These constraints often consist of both hard constraints and soft constraints. Hard constraints such as room capacity are to be fully respected. Soft constraints include

number of students having simultaneous exams, consecutive exams, or multiple exams. Both hard and soft constraints are used to evaluate the solution quality through an objective function. For the exam timetabling problem, the cost function is the objective function that measures the exam solution quality in terms of room capacity, number of students having simultaneous exams, consecutive exams, or multiple exams.

"The exam timetabling problem is a well known NP-hard optimization problem" (Schaerf, 1999) and (Cheng, Kleinberg, Kruk, Lindsey, and Steffy, 2003). In optimization problems, our attempt is to come out with a best solution, a solution with best obtained value relative to the scoring function adopted. Maximization problems where we try to get a higher scoring value and minimization problems where we try to lower the scoring value are both optimization problems.

3. Previous Work on Exam Timetabling

Some heuristic algorithms have been developed to solve the exam timetabling problem. Some have investigated a case-based reasoning technique called CBR and applied it to a set of algorithms previously used successfully to solve a set of problems. This was addressed via an inductive assumption stating that if an algorithm was successfully applied to a problem, it can also be applied to a similar problem. This work was done by Burke et al. in 2005 by using a simple simulated annealing algorithm. Other technical work has been done to address the exam timetabling problem. The Tabu Search was used by Di Gaspero and Schaerf in 2000, and White and Xie in 2001. Mansour, Tarhini, and Ishakian (2003) used a three-phase simulated annealing approach to solve the exam timetabling problem. Another evolutionary meta-heuristic technique, based on scatter search algorithm, has been

developed by Mansour and Ishakian (2007) for finding good suboptimal solutions for exam timetabling.

Another approach for exam timetabling has been developed by (Lofti and Cerveny, 1991) and implemented at some universities. This approach consists of multi-phase exam timetabling. The multi-phase exam timetabling process consists of 1) grouping the final exams into sets called `blocks' in order to minimize the number of students having simultaneous exams, 2) assigning exam days to courses while minimizing the number of students having more then one exam per day, 3) arranging the exam dates in a way to minimize the number of students having consecutive exams, and 4) assigning classrooms to exams as to maximize the space utilization.

In 2001, Erben has developed an alternative approach, based on some grouping criteria of the graph coloring problem in association with a fitness function defined on the set of partitions of vertices to guide the grouping genetic algorithm. This approach has been successfully applied to a choice of hard-to-color graph instances, and thus has been also extended to the application of timetabling problems.

Another approach has investigated the use of local search techniques based on various combinations of neighborhood functions, and has been applied to timetabling problem by Di Gaspero and Schaerf in 2003. They have proposed a set of generic operators that automatically compose neighborhood functions, giving rise to more complex ones by relying on some limiting constraint techniques to prune the list of candidates, and thus selecting the most effective search technique through a systematic analysis of all possible combinations built upon a set of basic, human-defined, neighborhood functions.

El-Sayed, Abd El-Wahed, and Ismail (2008) combined both scatter search and genetic algorithm and developed a hybrid genetic scatter search algorithm (GSS) by replacing two phases in scatter search (combination and improvement) with two phases in genetic (crossover and mutation). GSS starts with the diversification and initialization of start points (population) without applying the improvement method. Reference set and subsets creation follows scatter search approach. Then as a

substitution to the combination method, a crossover process is performed for each subset along with a crossover probability to yield new offspring solutions. Instead of the improvement method, offspring solutions are replaced with a mutation probability at each element position called lotus. In the last step, the reference set is updated (if applicable) and GSS repeats until the stopping criterion is met. This GSS algorithm achieved consistent solutions but it took a longer execution time then scatter search technique.

In another study, a heuristic ordering based method, very similar to those used for graph coloring problems, has been combined with a backtracking technique to cope with the possible unsuitability of a heuristic. This heuristic ordering based method has been applied successfully to exam timetabling problem by Carter and Johnson in 2001.

Burke and Petrovic in 2004 have presented overviews of recent research conducted on university exam timetabling including hybrid evolutionary algorithms, meta-heuristics, multi-criteria approaches, case-based reasoning techniques and adaptive approaches. A research on sequential and clustering constraint-based techniques and meta-heuristic methods has been applied to solve timetabling problems by Qu, Burke, McCollum, Merlot, and Lee (2007).

4. Previous Work on Parallel Scatter Search

Different parallel implementations have addressed different strategies and variant types of metaheuristics to solve large scale problems yet not much work on parallel implementation on scatter search can be found in the literature. Adenso-Diaz, Garcia-Carbajal, and Lozano (2006) have emulated a set of parallelization strategies for scatter search algorithm on the 0-1 knapsack problem. They carried out a group of parallel scatter search algorithm threads by dividing scatter search algorithm into 2

phases. Phase I includes tasks related to initial population generation and reference set creation. Phase II includes all tasks related to Subsets creation and combination, solution improvement reference set update. For each one of these phases, they implemented three different ways of processors communication they called them parallelization strategies. These strategies are single walk strategy where every processor work independent from other processors, multiple walk with independent threads strategy which is not more than a simple replication on processors, and multiple walk with cooperative threads strategy where processors communicate together and share information generated. For both phases, they carried out two different ways of updating reference set, static reference set update where all reference set solutions are updated together at the end of subset combination process, and dynamic reference set update where reference set is checked for update on every solution generated by subset combination. Emulation results showed a low average in the parallel efficiency, and no clear out-performance in the implementation of different strategies regarding solution quality.

Lopez, Batista, and Moreno-Perez (2003) have developed a parallel scatter search algorithm to solve the p-median problem. They proposed three different parallel techniques to reach 1) less execution time or 2) more space exploration. These types are 1) synchronous parallel scatter search SPSS, 2) replicated combination scatter search RCSS and 3) replicated parallel scatter search RPSS. They proposed a low-level parallelism SPSS algorithm and replaced the local search in the basic algorithm by a parallel search to reduce the execution time. They divided the neighborhood among *n_pr* subsets where *n_pr* denotes total number of running processors. The replicated combination scatter search RCSS consists of generating and distributing multiple subsets from the reference set on the number of processors, where every subsets set is combined and improved. Thus, the RPSS algorithm is not more than a natural replication of the scatter search algorithm on the processors. It consists of a multi-start search where the local search on every processor is replaced by an entire scatter search with different population for every processor.

Bozejko and Wodecki (2008) have presented a parallel algorithm based on the scatter search and the path re-linking methods to solve a NP-hard flow shop scheduling problem with the constraints of: 1) jobs are to be routed on a group of machines, 2) every job has to be processed on all machines following a unique sequence, 3) the job order must be the same on every machine, 4) one job can be processed per machine, and 5) each job can be handled at exactly one machine at a given time. In this algorithm, the root processor creates and controls the starting solutions set S of size n, while calculations of path re-linking procedures are executed by all processors on local data. The path re-linking idea is based on a stochastic local search, starting from solution1, to find a new good solution2 where the other solution1 is used as a reference point. Communication takes place only to create a common set of new starting solutions. Once S is created on the root processor, it is broadcasted among all the processors. Then on every processor, a set of randomly chosen n/2 pairs from S is applied to path re-linking algorithm to produce another set S' of n/2 solutions. The non root processors send back solutions from the set S' to the root processor to be evaluated and applied to the set S. This procedure is repeated until a defined iteration number is reached.

Another parallel scatter search algorithm has been also developed by Lopez, Torres, Batista, Moreno-Perez, and Moreno-Vega (2006) to solve the classification subset selection feature. This algorithm is a replication of the scatter search algorithm except that it replaces the combination method by two greedy methods and runs these methods simultaneously on every processor. The two proposed greedy methods for combining solutions are the greedy combination (GC) and the reduced greedy combination (RGC). In both methods, the subset generation method produces two new solutions, S'_1 and S'_2 from subset S_1 and S_2 by adding the features common to S_1 and S_2. Only the reduced method considers those features with the highest accuracy percentages of previously found solutions. This proposed parallel scatter search runs sequentially and uses the parallelism only when it applies its two-greedy-strategy combination method.

5. Thesis Objectives

In this work, we looked for a parallel scatter search meta-heuristic algorithm that achieves a decrease in time execution and increase in solution quality. We developed a full communication parallel scatter search algorithm, Parallel Scatter Search (PSS). In PSS, we implemented a tree-based bottom-up processor communication for parallelizing the reference set generation and update methods. All other phases of the scatter search algorithm are distributed evenly on all running processors. This approach has lead to increase in performance and increase in quality. We also experimented with two other parallel implementations, Master-Slave Parallel Scatter Search (MSPSS) and Embarrassingly Parallel Scatter Search (EPSS). MSPSS consists of assigning the root processor the role of a master processor that is responsible of dividing the scatter search tasks and delegating them slave processors, and the role of slave processors to all other processors. EPSS is a replication version of the scatter search where every processor works independently.

6. Thesis Organization

Chapter 2 describes the exam timetabling problem. A brief description of the sequential scatter search algorithm is given in Chapter 3. Chapter 4 explains the parallel scatter search algorithm defined for exam timetabling. In Chapter 5, we list the empirical results. Concluding remarks are given in Chapter 6.

Chapter 2

Exam Timetabling Problem Definition

The student exam timetabling problem is a standard event scheduling problem. Exam timetabling consists of assigning dates, time slots and rooms to a given set of exams subject to minimizing the number of students having simultaneous exams, the number of students having consecutive exams per one day, the number of students having three multiple exams per day and the number of students having four multiple exams per day. In addition to the above mentioned constraints, rooms can only be allocated to exams if their capacities fit all students attending the scheduled exams. In our exam scheduling problem, we will defined for every room its capacity and for every exam the number of students attending it.

For number of exam days, we have considered exam days ranging from 8 to 9 days. Exam periods or time slots will be arranged into 4 periods per day.

To measure the feasibility of all scheduled exams, we have derived an objective function from all constraints that computes all constraints violated. Since not all constraints have the same critical level, we have associated a penalty or a coefficient to every constraint and assigned higher penalties to more critical ones. So a better schedule is a schedule that has a lower objective function value and thus violates less the following constraints:

- Avoid students having simultaneous exams
- Avoid students having consecutive exams
- Avoid students having 3 multiple exams per day
- Avoid students having more than 3 multiple exams per day

- Do not assign rooms with capacity less than number of students attending the exam

In order to implement the exam timetabling constraints, we defined S_{SE} as the number of students having simultaneous exams, S_{CE} as the number of students having consecutive exams within the same day, S_{3E} as the number of students having 3 multiple exams per day and S_{4E} as the number of students having 4 multiple exams per day. ρ_{xy} is the assignment of room y to exam x subject to room capacity Ψ_y and number of students enrolled in exam x.

Other limiting constraints are as follows:

(1) Predefined number of classrooms R.

(2) Limited number of Exam Periods $\prod$ ($\prod$ = D * E, where D is the number of exam days and E is the number of exam period per day).

(3) Predefined Classroom Capacities Ψ (Ψ_1, Ψ_2,..., Ψ_n).

The objective function, OF1, is defined by Mansour and Isahakian (2007) as follows:

$$OF1=\alpha * S_{SE} + \varphi * S_{CE}+\sigma * S_{3E}+ \beta * S_{4E}+\gamma * (\Sigma_{1<=x<=\Pi} \Sigma_{1<=y<=R} \rho_{xy}) \quad (1)$$

Where α, = user-defined weight for students having simultaneous exams

φ = user-defined weight for students having consecutive exams

σ = user-defined weight for students having multiple below 4 exams

β = user-defined weight for students having multiple 3 exams

γ = user-defined weight for room capacity violations.

The inner summation in ($\Sigma_{1<=x<=\Pi} \Sigma_{1<=y<=R} \rho_{xy}$) gives the total room violation number for a selected timeslot while the outer summation gives the total room violation number in all timeslots.

Assigning values to these defined weights is flexible and falls under the user's particular choices and requirements. Thus, different instances of the problem can be reached by assigning different values to these weights.

Chapter 3

Sequential Scatter Search Meta-heuristic

In 2003, Laguna and Marti defined the sequential scatter search as a population-based meta-heuristic algorithm that generates solutions out of other combined ones. To achieve this, we adapt the controlled diversification strategy which consists of choosing random options by controlled manner to ensure diversification and better exploration of the solution space. Once these solutions are generated, they become input to a heuristic improvement method to render them more feasible. Then after the reference set is generated, we select solutions subsets and apply, on every subset, both the combination and improvement methods to generate new solutions.

The basic template of implementing the scatter search algorithm consists of five methods described by Laguna and Marti (2003) as follows:

1- Diversification Generation Method: generates a set of diverse solutions by exploring the solution space. This set of diverse solutions is called the initial population set.

2- Improvement Method: transforms a trial feasible or not feasible solution into a new more feasible one.

3- Reference Set Update Method: creates and maintains a reference set of size b consisting of two smaller sets called high quality set and diversity set of size $b1$ and $b2$ respectively, where $b= b1+ b2$.

4- Subset Generation Method: generates a subset from the reference set solutions.

5- Solution Combination Method: combines subset solutions to yield new solution solutions.

Figure 3.1 shows a graphic representation of the sequential scatter search algorithm.

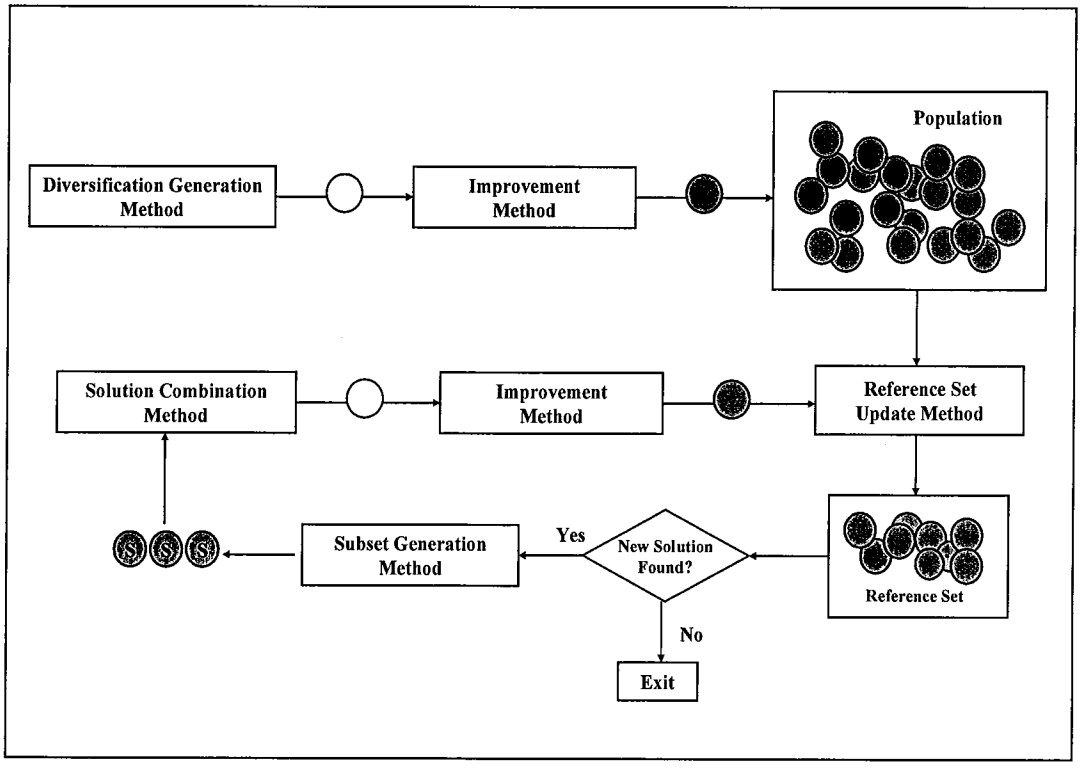

Figure 3.1: Schematic representation of a basic sequential scatter search design.

Scatter search algorithm starts by applying the population generation method to create a starting population set. Once the improvement method is applied to all population set solution, the reference set creation method starts creating a reference set by choosing b high quality solutions. The size of b is usually 20. The search procedure starts by assuming that there is a better new solution, i.e. assigning RefSetImproved = True. Then subsets are combined in step 7 and improved in step 8 to yield new solutions. If any of the new solutions has smaller objective function value or bigger diversity value than the worse reference set solution, the worse solution in the reference set is then replaced by the new solution. This process of subsets creation and solution combination and improvement is repeated until no better solution is found, i.e. RefSetImproved = False.

Figure 3.2 shows a high level pseudocode of a basic sequential scatter search algorithm.

1. Set Population Set $P = 0$.

   while $P < PopSize$ **do**

   Construct a solution x by using the diversification generation method.

   Apply the improvement method on solution x.

   If $(x \notin P)$ **then**

   $P = P \cup x$.

   end if

   end while

2. Create *ReferenceSet* = $\{x1, \ldots, xb\}$ with the "best" b solutions in P by using reference set update method.
3. Set *RefSetImproved* = TRUE.

   while (*RefSetImproved*) **do**

   4. Generate *Subsets* with the subset generation method.
   5. Set *RefSetImproved* = FALSE.

   while (*Subsets* $\neq \emptyset$) **do**

   6. Select the next subset s in *Subsets*.
   7. Apply the solution combination method to s.
   8. Apply the improvement method to solution s.
   9. Apply the reference set update method.

      if (*ReferenceSet* changed) **then**

      Set *RefSetImproved* = TRUE.

      end if

   10. Delete s from *Subsets*.

   end while

end while

Figure 3.2: Basic sequential scatter search pseudocode.

Chapter 4

Parallel Scatter Search Algorithms for Exam Timetabling

4.1 Overview of Parallel Scatter Search Algorithms

We propose three different parallel scatter search algorithms as illustrated in Figures 4.1, 4.2 and 4.3 respectively. These algorithms are Master-Slave Parallel Scatter Search (MSPSS), Parallel Scatter Search (PSS) and Embarrassingly Parallel Scatter Search (EPSS). In addition, Figures 4.4, 4.5 and 4.6 show the communication features of MSPSS, PSS and EPSS among all *n_pr* processors where *n_pr* is the total number of processors in execution. Parallel scatter search algorithm phases are described in details in the following sub-sections.

4.2 Parallel Scatter Search Algorithm PSS

The Parallel Scatter Search (PSS) is a fully parallel algorithm that applies a tree-based communication system among all processors $Processor_p$, where $1 \leq p \leq n\_pr$. First, $Processor_0$ or *RootProcessor* broadcasts the input data files to all other processors and defines the initial population size $ProcessorPopSize_p$ for each $Processor_p$.

Once the $ProcessorPopSize_p$ is defined each processor $Processor_p$ ($0 \leq p \leq n\_pr$), $Processor_p$ starts the diversification generation method to create its local

population size. The improvement method follows every creation of every population solution. The tree-based communication system in PSS reduces the communication time and allowed us to adopt more to the scatter search algorithm by using a lager global reference set size on every processor, $ProcessorGlobalRefSet_p$. After the creation of $ProcessorPopSize_p$, every $Processor_p$ starts generating its complete local reference set $ProcessorGlobalRefSe_p$. Once all global reference sets are defined, PSS starts merging all $ProcessorGlobalRefSe_p$ in a buttom-up tree-based communication way starting from leaf processors and moving up toward *RootProcessor*. Each $Processor_p$ ($0 \leq p \leq n\_pr$) creates a local $ProcessorGlobalRefSet_p$ of size equal to *GlobalReferenceSet*. Then every 2 processors start unifying their corresponding *GlobalReferenceSet* in a bottom-up tree-based communication toward the *RootProcessor*. After having a complete unified *GlobalReferenceSet* from all $ProcessorGlobalRefSet_p$, the *RootProcessor* broadcasts it to all other processors.

The subset generation method follows the reference set generation method. Each $Processor_p$ ($0 \leq p \leq n\_pr$) creates it local *ProcessorSubsets* of size $SubsetSize_p$. The subset generation method adapts the creation of 2-element subsets. After $SubsetSize_p$ *ProcessorSubsets* are determined, combination and improvement methods are applied to subsets in an attempt to yield more feasible solutions. The reference set update method is then applied on every generated solution to check whether this new candidate solution can be added to the processor reference set *ProcessorGlobalRefSet*.

This procedure is applied on every processor. If $ProcessorGlobalRefSet_p$ is changed at any processor, all processors re-unify all $ProcessorGlobalRefSet_p$ in bottom-up tree-based communication towards the *RootProcessor*, where the *RootProcessor* at it turn re-generates a *GlobalReferenceSet* and broadcasts it to all processors to create new local *ProcessorSubsets* and generate new combined candidate solutions. This loop of re-generating *GlobalReferenceSet* and creating new local *ProcessorSubsets* is repeated unless no $ProcessorGlobalRefSet_p$ was improved

at any processor. At the end, PSS outputs the best optimum solution in *GlobalReferenceSet.*

Using the bottom-up tree-based communication method, we were able to reduce the communication needed to update *GlobalReferenceSet* from *n_pr -1* times required to unify all $ProcessorGlobalRefSet_p$ to log(*n_pr*) times, and thus increase the size of *GlobalReferenceSet* to highly conform to sequential scatter search.

4.3 Master-Slave Parallel Scatter Search Algorithm MSPSS

The Master-Slave Parallel Scatter Search (MSPSS) differs from PSS by its master-slave technique where the master processor denoted by *RootProcessor* is responsible of dividing the jobs and delegating tasks to remaining processors denoted by $Processor_p$, where $1 \leq p \leq n\_pr$. MSPSS applies the same diversification, subset generation method, and combination and improvement methods used in PSS. For the reference set generation method, MSPSS generates a partial reference set at each $Processor_p$ denoted by $ProcessorLocalRefSet_p$ in a way that the summation of all $ProcessorLocalRefSet_p$ is equal to the complete requested *GlobalReferenceSet*. Once all partial local reference sets are defined, the non-root processors send $ProcessorLocalRefSet_p$ to *RootProcessor*. After receiving *n_pr*-1 $ProcessorLocalRefSet_p$, *RootProcessor* creates *GlobalReferenceSet* broadcasts it to all other processors.

4.4 Embarrassingly Parallel Scatter Search Algorithm EPSS

Embarrassinly Parallel Scatter Search (EPSS) applies a replication algorithm on all processors where every processor creates its own initial population size, generates its related $ProcessorGlobalRefSet_p$, and applies combination and improvement methods to all subsets generated from corresponding $ProcessorGlobalRefSet_p$. in the EPSS model, no communication is encountered unless in the first beginning where *RootProcessor* broadcasts the input data files to all other processors and at the end in a master slave way, where it gathers all $ProcessorGlobalRefSet_p$ from n_pr-1 processors to unify $ProcessorGlobalRefSet_0$ and select the most optimum solution out of it. The communication cost is reduced in EPSS model but more overhead work is requested by all processors.

1- Broadcast input data from root processor to all processors;
2- Determine $ProcessorPopSize_p$ from n_pr processors;
for *processor* {0, ..., n_pr - 1} **do**
    3-Use the Diversification generation method to construct $ProcessorPopSize_p$ solutions;
    4- Apply the Improvement method on $ProcessorPopSize_p$ solutions;
    5-Use the ReferenceSet generation method to create $ProcessorLocalRefSet_p$;
end for;
if (*RootProcessor*) **then**
    6- Receive $ProcessorLocalRefSet_p$ from $Processor_p$ where p = 1,2,.., n_pr-1;
    7- Create *GlobalReferenceSet*;
    8- Broadcast *GlobalReferenceSet* to all processors;
else
    9- Send $ProcessorLocalRefSet_p$ to *RootProcessor*;
    10- Receive *GlobalReferenceSet* from *RootProcessor*;
end if;
repeat
    for *processor* {0, ..., n_pr - 1} **do**
        11- Generate *SubsetSize Subsets* with the Subset generation method;
        12- Determine $SubsetSize_p$ *ProcessorSubsets*;
        while (*ProcessorSubsets* $\neq \emptyset$) **do**
            13- Select next *subset s* from *ProcessorSubsets*;
            14-Apply the Combination method on subset s to generate candidate solution x;
            15- Apply the Improvement method on candidate solution x;
            16-Apply the ReferenceSet Update method to update *ProcessorLocalRefSet*;
        end while;
    end for;
    if (at least 1 *ProcessorLocalRefSet* was improved) **then**
        if (*RootProcessor*) **then**
            17-Receive n_pr-1 *ProcessorLocalRefSet* from n_pr-1 processors;
            18- Update *GlobalReferenceSet*;
            19- Broadcast *GlobalReferenceSet* to all processors;
        else
            20- Send *ProcessorLocalRefSet* to *RootProcessor*;
            21- Receive *GlobalReferenceSet* from *RootProcessor*;
        end if;
    end if;
until (no improvement in *GlobalReferenceSet*);

Figure 4.1: Master-Slave Parallel Scatter Search Algorithm (MSPSS).

1- Broadcast input data from root processor to all processors;

2- Determine $ProcessorPopSize_p$ from *n_pr* processors;

for *processor* {0, ..., *n_pr* - 1} **do**

3-Use the Diversification generation method to construct $ProcessorPopSize_p$ solutions;

4- Apply the Improvement method on $ProcessorPopSize_p$ solutions;

5-Use the ReferenceSet generation method to create $ProcessorGlobalRefSet_p$;

end for;

6- Receive all $ProcessorGlobalRefSet_p$ in *Button-up Tree-based Communication* way to create *GlobalReferenceSet* at *RootProcessor*;

if (*RootProcessor*) **then**

7- Broadcast *GlobalReferenceSet* to all processors;

end if;

repeat

for *processor* {0, ..., *n_pr* - 1} **do**

8- Generate *SubsetSize Subsets* with the Subset generation method;

9- Determine $SubsetSize_p$ *ProcessorSubsets*;

while (*ProcessorSubsets* $\neq \emptyset$) **do**

10- Select next *subset s* from *ProcessorSubsets*;

11-Apply the Combination method on subset *s* to generate candidate solution *x*;

12- Apply the Improvement method on candidate solution *x*;

13-Apply the ReferenceSet Update method to update *ProcessorGlobalRefSet*;

end while;

end for;

if (at least 1 *ProcessorGlobalRefSet* was improved) **then**

14-Receive all $ProcessorGlobalRefSet_p$ in *Button-up Tree-based Communication* way to create *GlobalReferenceSet* at *RootProcessor*;

if (*RootProcessor*) **then**

15- Broadcast *GlobalReferenceSet* to all processors;

end if;

end if;

until (no improvement in *GlobalReferenceSet*);

Figure 4.2 Parallel Scatter Search Algorithm (PSS).

1- Broadcast input data from root processor to all processors;

2- Determine $ProcessorPopSize_p$ from n_pr processors;

for *processor* {0, ..., n_pr - 1} **do**

3-Use Diversification generation to construct $ProcessorPopSize_p$ solutions;

4- Apply the Improvement method on $ProcessorPopSize_p$ solutions;

5-Use the ReferenceSet generation method to create $ProcessorGlobalRefSet_p$;

end for;

repeat

for *processor* {0, ..., n_pr - 1} **do**

11- Generate *GlobalSubsets* with the Subset generation method from $ProcessorGlobalRefSet_p$;

while (*GlobalSubsets* $\neq \emptyset$) **do**

12- Select next *subset s* from *GlobalSubsets*;

13-Apply the Combination method on *s* to generate candidate *x*;

14- Apply the Improvement method on candidate solution *x*;

15-Apply ReferenceSet Update method to update *GlobalSubsets*;

end while;

end for;

until (no improvement in $ProcessorGlobalRefSet_p$);

if (*RootProcessor*) **then**

16-Receive n_pr-1 $ProcessorGlobalRefSet_p$ from n_pr-1 processors;

17- Update *Root* $ProcessorGlobalRefSet_p$;

else

18- Send $ProcessorGlobalRefSet_p$ to *RootProcessor*;

end if;

Figure 4.3: Embarrassingly Parallel Scatter Search Algorithm (EPSS).

Figure 4.4: Communication features of MSPSS.

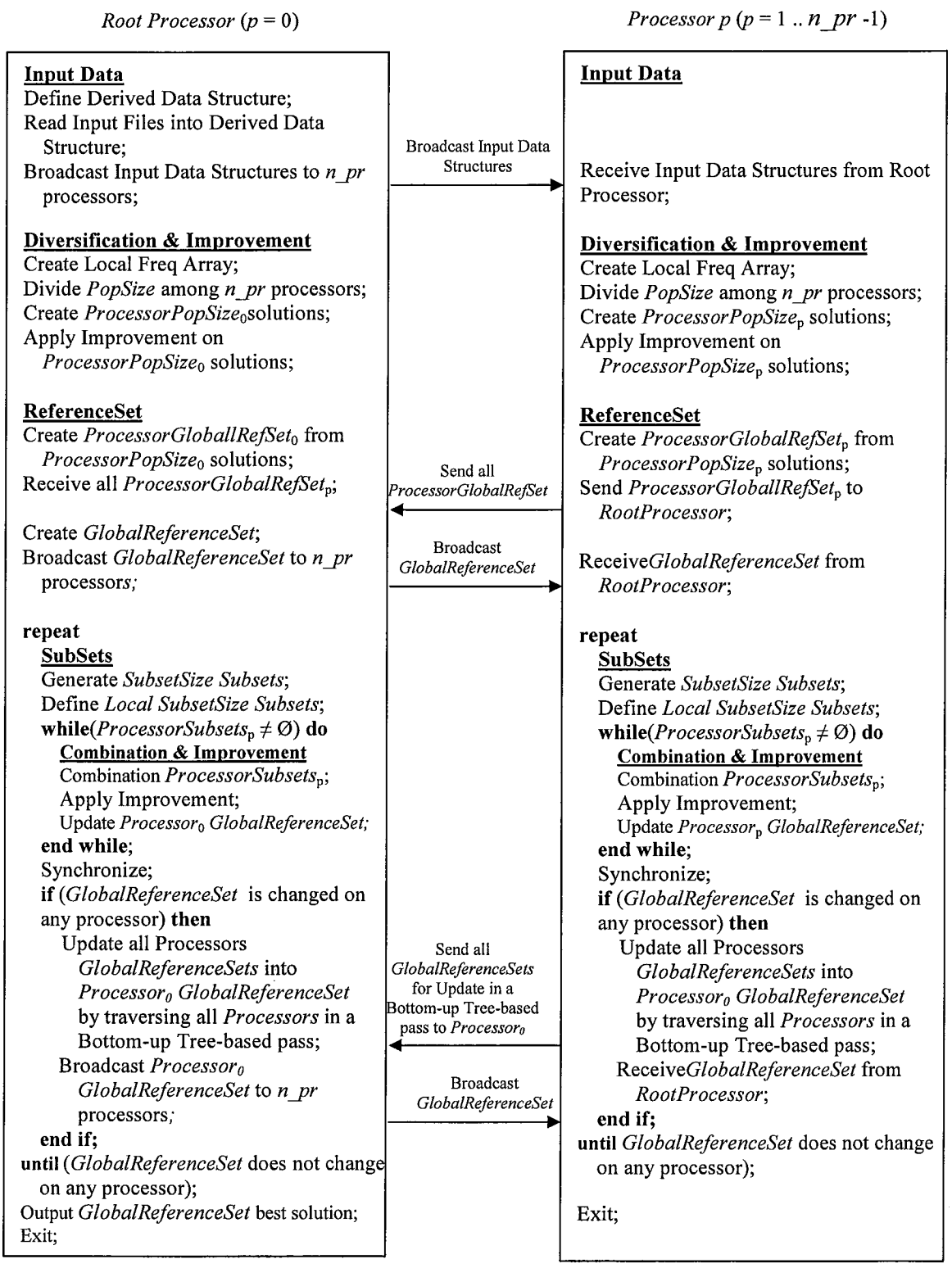

Figure 4.5: Communication features of PSS.

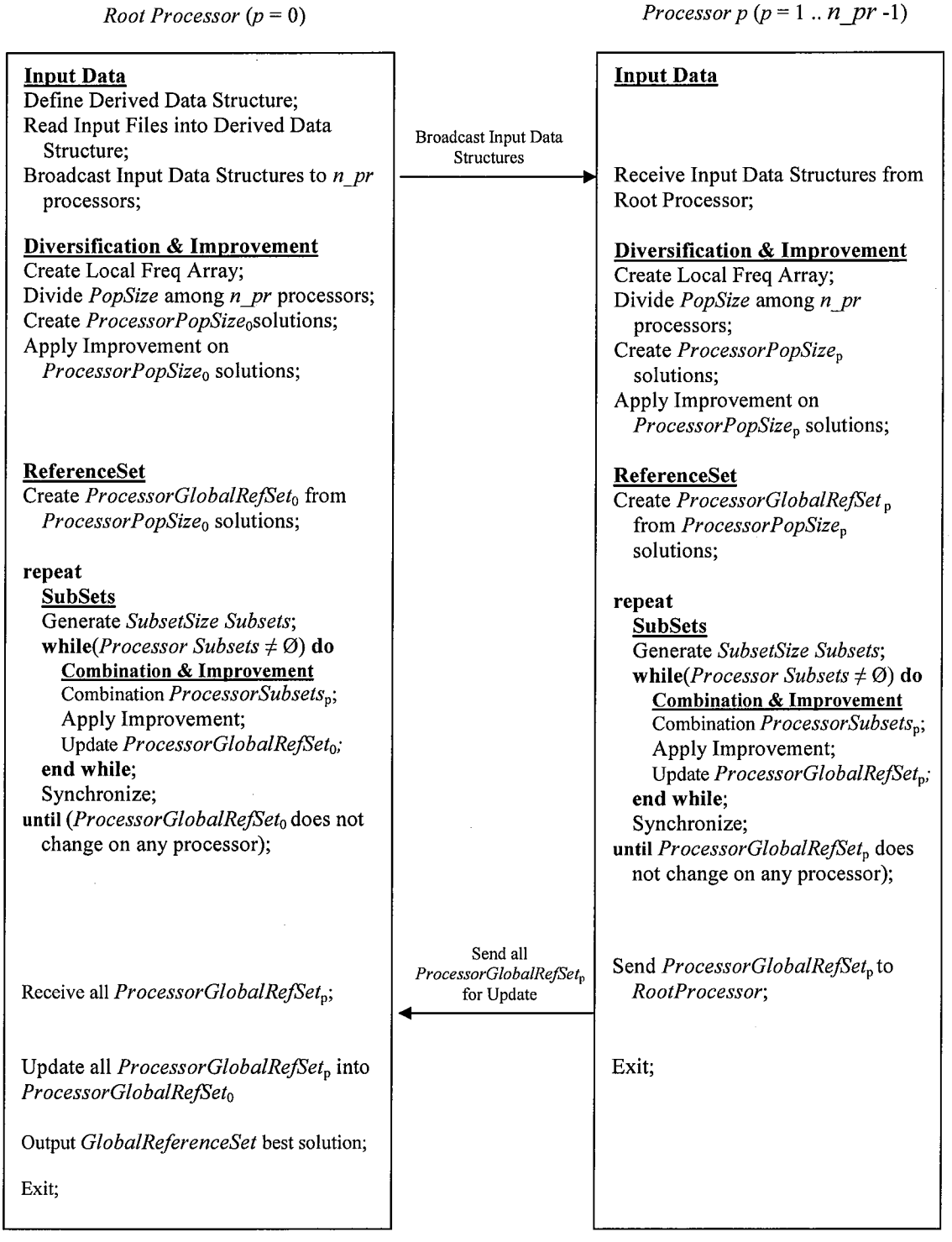

Figure 4.6: Communication features of EPSS.

4.5 Parallel Diversification Generation Method

The parallel diversification generation method is designed to achieve a high speedup rate and use of full parallel computation by reducing the differences of processing time among running processors. For this reason, the parallel diversification generation method partitions the initial population size (*PopSize*) into *n_pr ProcessorPopSize*, where *n_pr* denotes the number of processors running in parallel, such that:

$$PopSize = \sum_{p=0..n\_pr-1} ProcessorPopSize_p \quad (2)$$

Since in most of the cases *PopSize* is not a multiple of the running processors, we divide *PopSize* equally into the *n_pr* processors and distribute the remaining set of the initial population solutions one by one on the running processors starting from processor 0 and moving forward until no more solution number is left in the remaining set. In other words, we define *ProcessorPopSize* as a function of *PopSize*, number of processors *n_pr* and processor rank/id *p* as follows:

$$\text{If } PopSize \geq [\text{int}(PopSize / n\_pr) * n\_pr] + (1 + p) \text{ then}$$
$$ProcessorPopSize = (PopSize / n\_pr) + 1. \quad (3)$$
$$\text{Otherwise, } ProcessorPopSize = PopSize / n\_pr.$$

Our parallel diversification generation method emphasizes diversification over randomization. For this reason, we use a controlled randomization and a frequency-based memory technique to generate a set of diverse solutions. In our implementation, we start by dividing the range of $\prod$ ($\prod$ = D * E, where D is number of Exam Days and E is number of Exam Day Periods, usually = 4) into 4 sub-ranges.

To assign a period to a given exam, we first select the sub-range that correspond to the least previously selected one, and then select the period value within this sub-range. To maintain a high rate of parallel computation and reduce unnecessary communication load, we replicate the frequency-based technique of the sub-ranges count on all the *n_pr* processors, so that each processor is responsible for managing its frequency based count accordingly.

On every running processor p, the parallel diversification generation method starts generating initial solutions until their count reaches $ProcessorPopSize_p$. For every initial solution, we generate random rooms and allocate exam periods according to the controlled randomization method to every course in the list of courses to be scheduled. Figure 4.7 shows an outline of the parallel diversification generation method.

```
for processor {0, ..., n_pr -1} do
    counter = 0;
    CreateControlRandomArray();
    if (PopSize ≥ [int(PopSize / n_pr) * n_pr] + (1 + processor)) then
        ProcessorPopSize = (PopSize / n_pr) + 1;
    else
        ProcessorPopSize = PopSize / n_pr;
    end if;
    while (counter < ProcessorPopSize)
        InitiateEmptySolution(Sol);
        for course (0 ... MaxCourseNum – 1 in Sol) do
            Range = GetSubRange();
            ExamPeriod = (∏ / 4 * Range) + Random(0, ∏/4);
            ExamRoom = Random(1,R);
            AdjustFreqCount of ControlRandomArray;
        end for;
        if not (SolutionExists) then
            Add Sol to ProcessorPopSize Solutions;
            counter ++;
        end if;
    end while;
end for;
```

Figure 4.7: Parallel diversification generation method pseudocode

4.6 Improvement Method

The Improvement method of the parallel scatter search follows both the parallel diversification generation method and the combination method. The purpose of this Improvement method is to generate more feasible solutions, i.e. solutions with minimum objective function values.

Two types of improvement methods exist. The first type is based on the concept of First Move, where we move all elements in the input solution to the first locations that improve the solution objective function value. Another type or option is to apply the Improvement method on the input solution elements in an attempt to render the input solution as optimum as possible. This consists of applying the Improvement method a multiple of times until we reach a more optimum solution with minimum objective function value. This concept uses the Best Move method, where we move all solution elements to the best locations that improve its solution objective function value. In the parallel scatter search algorithm, we adapt the second option which is Best Move, since its main drawback, time execution, is decreased by the parallelization feature. We compute total *PopSize* Improvement Cost as follows:

$$\text{Total } PopSize \text{ Improvement Cost} = (1/\ n\_pr) * C_{PopSize} + C_s \qquad (4)$$

Where $C_{PopSize}$= cost needed to run the Improvement method on *PopSize* solutions, and $C_1 = 0$ if *PopSize* is divisible by *n_pr*, or C_s = cost needed to run the Improvement method on one solution 0 if *PopSize* is not divisible by *n_pr*.

Figure 4.8 illustrates the design of the Improvement method.

```
SolutionImproved = TRUE;
while (SolutionImproved)
    for course (0 ... MaxCourseNum – 1) do  /* loop courses in solution Sol */
        SolutionImproved = FALSE;
        for period (0 ... MaxPeriodRandNum – 1) do
            PrevCourseObjVal = ObjectiveFunction(Sol, course,  period);
            SelectRandExamPeriod(course);
            NewCourseObjVal = ObjectiveFunction(Sol, course, period);
            if (PrevCourseObjVal > NewCourseObjVal ) then
                AssignFeasibleExamPeriod(course);
                SolutionImproved = TRUE;
            end if;
        end for;
    end for;
end while;
SolutionImproved = TRUE;
while (SolutionImproved)
    for course (0 ... MaxCourseNum – 1) do  /* loop courses in solution Sol */
        SolutionImproved = FALSE;
        for room (0 ... MaxRoomRandNum – 1) do
            PrevCourseObjVal = ObjectiveFunction(Sol, course, room);
            SelectRandRoom(course);
            NewCourseObjVal = ObjectiveFunction(Sol, course, room);
            if (PrevCourseObjVal > NewCourseObjVal ) then
                AssignFeasibleRoom(course);
                SolutionImproved = TRUE;
            end if;
        end for;
    end for;
end while;
```

Figure 4.8: Improvement method pseudocode

4.7 Parallel Reference Set Update Method

The Parallel Reference Set Update method covers two reference set generation and update methods. The first method, Local Reference Set Update method, consists of creating and maintaining a local reference set (*ProcessorLocalRefSet*) on every running processor from corresponding *ProcessorPopSize*. The second method, Global Reference Set Update method, is responsible for creating and updating a global reference set (GlobalReferenceSet) at the root processor.

Once a *ProcessorPopSize* is created on processor p, the Local Reference Set Update method generates $ProcessorLocalRefSet_p$. When all *n_pr ProcessorLocalRefSets* are created, the Global Reference Set Update method gathers all *n_pr ProcessorLocalRefSets* on the root processor to 1) generate the global reference set GlobalReferenceSet, and 2) broadcast GlobalReferenceSet to all processors as a pre-requisite stage required by the parallel subset generation method.

All reference sets, GlobalReferenceSet and *n_pr ProcessorLocalRefSets* are constructed by adding two smaller sub reference sets. The first sub reference set consists of a set of best quality solutions called *HQRefSet*, and the second sub reference set consists of the farthest diverse solutions and called *DivRefSet*.

The generation of the *ProcessorLocalRefSet* of size p_b at processor p is done by creating $HQRefSet_p$ by selecting $RefSetSize_{pb1}$ solutions from the best solutions found in the *ProcessorPopSize*, i.e. $RefSetSize_{pb1}$ solutions with lowest objective functions values, and $DivRefSet_p$ by selecting $RefSetSize_{pb2}$ farthest solutions of the *ProcessorPopSize* from the $RefSetSize_{pb1}$ solutions already chosen as high quality solutions, and combining both $HQRefSet_p$ and $DivRefSet_p$ so that $p_b = p_{b1} + p_{b2}$. The

definition of the *ProcessorLocalRefSet* size p_b is a function of *GlobalReferenceSet* size b (b is usually equal to 20), number of processors n_pr and processor rank/id p as follows:

$$\text{If } b \geq [\text{int}(b / n\_pr) * n\_pr] + (1 + p) \text{ then}$$
$$p_b = (b / n\_pr) + 1. \quad (5)$$
$$\text{Otherwise, } p_b = b / n\_pr.$$

To determine the $DivRefSet_p$ solutions, we sort the $RefSetSize_{pb1}$ solutions in the $HQRefSet_p$ reference set in ascending order according to their objective functions values. After that, we compute the Minimum Distance between all solutions z in the *ProcessorLocalRefSet* and the remaining solutions z' in $HQRefSet_p$ as follows:

$$\partial_{\min}(z) = \text{Min } (\text{Dist}(z, z')) \quad (6)$$

The reason behind choosing the Diversity Minimum Distance equation could be justified by our intention to select the farthest solution z from *ProcessorPopSize* to the complete solutions set $HQRefSet_p$ and not to every solution z' in $HQRefSet_p$. Therefore, we select the closest solution z' from the set $HQRefSet_p$ to z, i.e. just the minimum distance to include solution z to the set $HQRefSet_p$.

For maintaining *ProcessorLocalRefSet*, we follow the subsequent steps:

1) Order $HQRefSet_p$ in ascending order according to the objective function values of its solutions.
2) Order $DivRefSet_p$ in descending order according to the distance values of its solutions.
3) If the objective function value of the candidate solution z is greater than the objective function value of the worst solution z_{worse} in $HQRefSet_p$., replace

z_{worse} by candidate solution z and reorder $HQRefSet_p$ according to their objective function values.

4) If the objective function value of the candidate solution z is greater than the objective function value of z_{worse} and the distance $\partial_{min}(z)$ to all the solutions in the $HQRefSet_p$ is smaller that of the worst, least diverse solution in $DivRefSet_p$, then replace the worst solution in $DivRefSet_p$ by the candidate solution z and reorder $DivRefSet_p$ based on the Distance.
5) Disregard solution z if the distance $\partial_{min}(z)$ to all *HQRefSet* solutions is smaller than the least diverse solution in $DivRefSet_p$.

The same approach applies to the generation and the maintenance of the *GlobalReferenceSet* where *HQRefSet* and *DivRefSet* follow the same techniques adapted for managing *ProcessorLocalRefSet*s.

4.8 Parallel Subset Generation Method

Parallel Subset Generation method generates replicated subsets of reference set solutions on *n_pr* processor. Basic scatter search algorithm considers the generation of subsets of size 2, 3 and 4, and subset sizes larger than 4, by choosing the best *i* elements of *ReferenceSet* and varying *i* from 5 to the *RefSetSize*. Thus, for subset size 2, there is a maximum of $SubSetSize = (RefSetSize^2 - RefSetSize)/2$ as mentioned by (Laguna and Marti, 2003). In our approach, we create subsets of size 3 by augmenting 2-elements subset with to include the best solution not in this subset.

Thus, for subset type 1, we combine all possible solutions (x, x') from the *GlobalReferenceSet* generated by the parallel reference set update method. For subset type 2, we augment the 2-element subsets previously created by adding the best solution x'' in *GlobalReferenceSet* with the best objective function value, such that $x'' \notin \{x, x'\}$.

Since we are concerned about reducing the communication overload and since the subset generation method is relatively a fast operation and does not require a long processing time, we adapt replicating the subset generation method on all *n_pr* processors.

4.9 Solution Combination Method

The solution combination method combines the elements in each subset that gives higher partial objective function value in order to yield more optimum solution. The solutions combined may be infeasible as stated by Laguna and Marti (2003). For this reason, we apply the improvement method on every combined solution.

Our combination method approach adapts a voting technique depending on the objective function values, and applies a randomization selection of Room and Exam Period in case we have the same scores. First, we initiate an empty solution *s* with all courses ordered in decreasing order according to the number of students attending them. Then, for each course *c* in the ordered solution, we consider the partial objective function values for the identical course in all of the 3 solutions x, x' and x" in the 3-element subset. After, we select the Room and the Exam Period that correspond to lowest objective function value. Then, we apply the improvement method on the combined solutions.

For parallel processing and time reduction, we divide the number of Subset Size into the processors in execution. This way, each processor combines only the subsets assigned to it.

The definition of *ProcessorSubsets* size is a function of *Subsets SubSetSize*, number of processors *n_pr* and processor rank/id *p* as follows:

If $SubSetSize \geq [\text{int}(SubSetSize / n\_pr) * n\_pr] + (1 + p)$ then

$$ProcessorSubsets \text{ size} = (SubSetSize / n\_pr) + 1. \quad (7)$$

Otherwise, $ProcessorSubsets \text{ size} = SubSetSize / n\_pr$.

Chapter 5
Empirical Results

5.1 Experimental Procedure

We apply our parallel scatter search algorithm to four real-world subject problem instances, as it is shown in Table 5.1.

Table 5.1: Subject problems.

Subject Problem	Semester	# of Rooms	# of Exams	# of Students	# of Enrolments
SP1	Fall 2006	38	473	3652	13662
SP2	Spring 2007	38	472	3704	13455
SP3	Fall 2007	38	537	3911	15392
SP4	Spring 2008	38	634	3794	15858

The solution quality will be evaluated by objective function value based on the following components:

1) Number of students having simultaneous exams
2) Number of students having consecutive exams per day
3) Number of students having multiple 3 exams per day
4) Number of students having multiple exams (above 3 exams)
5) Number of rooms with violated capacity

These five components will be tested on different numbers of processors, different semesters, number of periods ($\Pi = D * E$) and populations sizes. To apply

our scatter search algorithm in parallel, we set the number of running processors to be 1, 2, 4, 8 and 16. For number of periods, we consider $\Pi = 32$ for SP1, SP2 and SP3, and $\Pi = 36$ for SP4. The Semesters are defined in Table 6.1. As for the population size PopSize and reference set size RefSize, we choose PopSize to be 100, 200 and 1024 solutions, and RefSize to be 10, 20 and 32 solutions. For the Objective Function user-defined weights or coefficients, we use $\alpha = 200$, $\varphi = 1$, $\sigma = 10$, $\beta = 100$, and $\gamma = 100$ respectively.

We run our parallel scatter search program on a cluster of PCs, each has 2.33 GHz CPU and 2 RAM memory. Parallelization is implemented using C++ and MPI-2 software on Linux platform.

The objective function value, execution time in minutes, and speedups are presented for each number of processors. Speedup, the ratio of sequential time to parallel time is given below:

$$\text{Speedup} = \frac{\text{Sequential Execution Time on One Processor}}{\text{Parallel Execution Time on } n\_pr \text{ Processors}} \quad (8)$$

To give a more accurate measure of the true efficiency of the parallel scatter search program, we measured the parallel efficiency as it is defined by:

$$\text{Parallel Efficiency} = \frac{\text{Speedup} * 100}{n\_pr} \quad (9)$$

5.2 Experimental Results

Tables 5.2 and 5.3 show the results of executing PSS and MSPSS on SP1, SP2, SP3, and SP4 over 1-16 processors respectively using PopSize = 1024, RefSize = 32.

Table 5.2: Results for PSS, PopSize=1024, RefSize=32.

Semester	Processor#	S_{SE}	S_{CE}	S_{3E}	S_{4E}	R_V	Objective Function	Iteration#	Execution Time (Mins)	Parallel Efficiency
SP1	1	0	357	16	0	0	517	11	919	-
	2	1	405	11	0	0	715	17	704	65.25
	4	1	398	22	0	0	818	15	314	73.2
	8	0	352	10	0	0	452	13	139	82.65
	16	0	403	8	0	0	483	6	37	155.25
SP2	1	0	329	12	0	0	449	6	545	-
	2	0	314	6	0	0	374	13	562	48.5
	4	0	213	0	0	0	213	4	143	95.3
	8	0	203	1	0	0	213	3	56	121.65
	16	0	398	14	0	0	538	21	117	29.2
SP3	1	1	328	7	0	0	598	6	812	-
	2	1	224	6	0	0	484	23	1527	26.6
	4	1	268	3	0	0	498	15	476	42.6
	8	1	266	7	0	0	606	9	157	64.65
	16	1	219	6	0	0	479	16	145	35
SP4	1	0	276	2	0	0	296	10	1929	-
	2	0	272	1	0	0	282	10	990	97.4
	4	0	260	1	0	0	270	17	836	57.7
	8	0	260	2	0	0	280	31	754	32
	16	0	261	1	0	0	271	10	134	90

- Comparing PSS parallel efficiency for the four subject problems in Figure 5.1, we found the following:
    - For SP1, SP2 and SP3, the parallel efficiency increases as *n_pr* increases up to 8 processors and number of iterations decreases.

- When n_pr = 16, SP1 gives the highest efficiency rate where the number of iterations decreases, while both SP2 and SP3 decrease in efficiency where both numbers of iterations increase.
- For SP4, we deduced that parallel efficiency decreases when n_pr $\leq$ 8, but when n_pr = 16, the parallel efficiency increases as the number of iteration decreases.
- These differences in the number of search iterations and the parallel efficiency rates are due to the differences in the reference sets generated on the multiple processors.
- Examining SP1 and SP4 over 16 processors, we noticed a remarkable increase in the parallel efficiency, which makes it possible to run large problems over 16 processors within reasonable time.

- The communication performance of tree-based PSS could be more promising for appropriate interconnection networks that support tree communication than the cluster results.

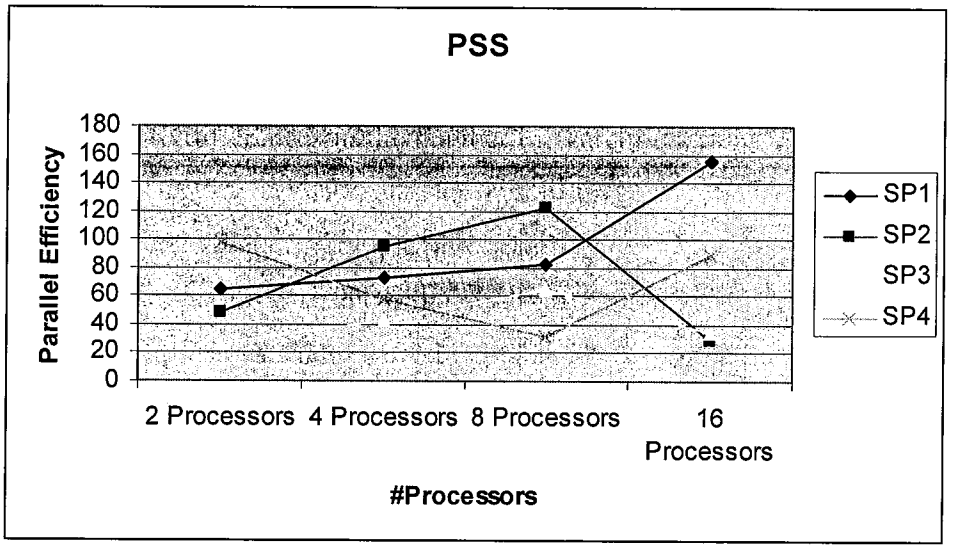

Figure 5.1: PSS parallel efficiency for SP1, SP2, SP3 and SP4.

Figure 5.2 gives an illustration for the objective functions obtained by running PSS on SP1, SP2, SP3 and SP4 over 1-16 processors.

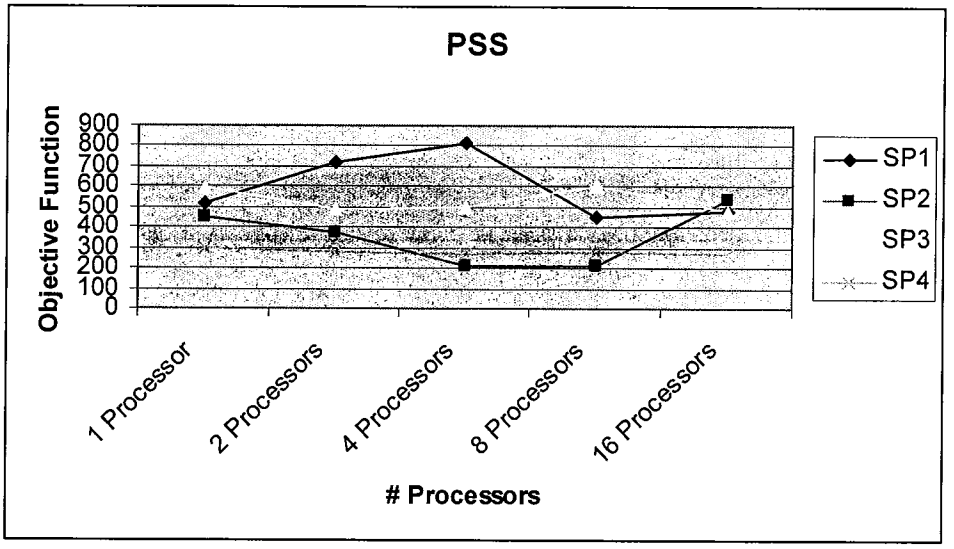

Figure 5.2: PSS objective functions for SP1, SP2, SP3 and SP4.

- Regarding PSS objective function results, we infer the following remarks:
  - SP1 gave lower quality solutions than the sequential scatter search when the number or processors is 2 or 4, while it gives higher quality solutions when the number of processors reaches 8 and 16.
  - SP2 gave higher quality solutions than the sequential scatter search when the number of processors is between 1 and 8. On 16 processors, the solution quality decreases and the objective function reaches a higher number than the sequential scatter search but still it is within a close range.

- SP3 and SP4 gave higher quality solutions than the sequential scatter search, and the objective functions obtained in general reached lower rates when the number of processors increased.
- We can deduce that PSS gives higher solutions quality than the sequential scatter search algorithm.

Table 5.3: Results for MSPSS, PopSize=1024, RefSize=32.

Semester	Processor#	S_{SE}	S_{CE}	S_{3E}	S_{4E}	R_V	Objective Function	Iteration#	Execution Time (Mins)	Parallel Efficiency
SP1	1	0	357	16	0	0	517	11	919	-
	2	0	363	9	0	0	453	4	192	239.3
	4	0	385	15	0	0	535	11	234	98.2
	8	0	368	12	0	0	488	20	209	55
	16	0	399	13	0	0	529	16	90	63.8
SP2	1	0	329	12	0	0	449	6	545	-
	2	0	311	7	0	0	381	9	398	68.5
	4	0	311	7	0	0	381	4	98	139
	8	0	313	9	0	0	403	10	112	60.8
	16	0	314	8	0	0	394	10	60	113.5
SP3	1	1	328	7	0	0	598	6	812	-
	2	1	242	4	0	0	482	10	690	58.8
	4	1	235	4	0	0	475	18	609	33.3
	8	1	212	5	0	0	462	13	222	45.7
	16	1	176	0	0	0	376	18	162	31.3
SP4	1	0	276	2	0	0	296	10	1929	-
	2	0	271	1	0	0	281	9	915	105.4
	4	0	244	1	0	0	254	11	557	86.6
	8	0	269	1	0	0	279	11	279	86.5
	16	0	261	1	0	0	271	18	235	51.3

Figure 5.3 shows a graphical illustration of MSPSS parallel efficiency for SP1, SP2, SP3 and SP4.

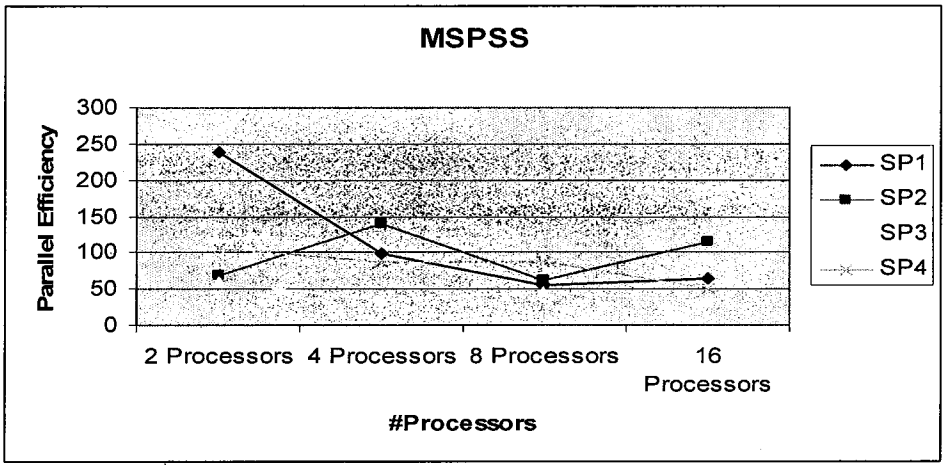

Figure 5.3: MSPSS parallel efficiency for SP1, SP2, SP3 and SP4.

- Examining MSPSS parallel efficiency, we deduce the following:
  - MSPSS gives lower efficiency rates for SP1, SP3 and SP4 when the number of processors increases.
  - For SP2, MSPSS gives lower efficiency rate when the number of processors is 8, but it yields higher efficiency rates over 4 and 16 processors.
  - We can deduce that MSPSS gives less efficiency rates when the number of processors increases.

Figure 5.4 illustrates a graphical representation of the objective functions obtained by running MSPSS for SP1, SP2, SP3 and SP4.

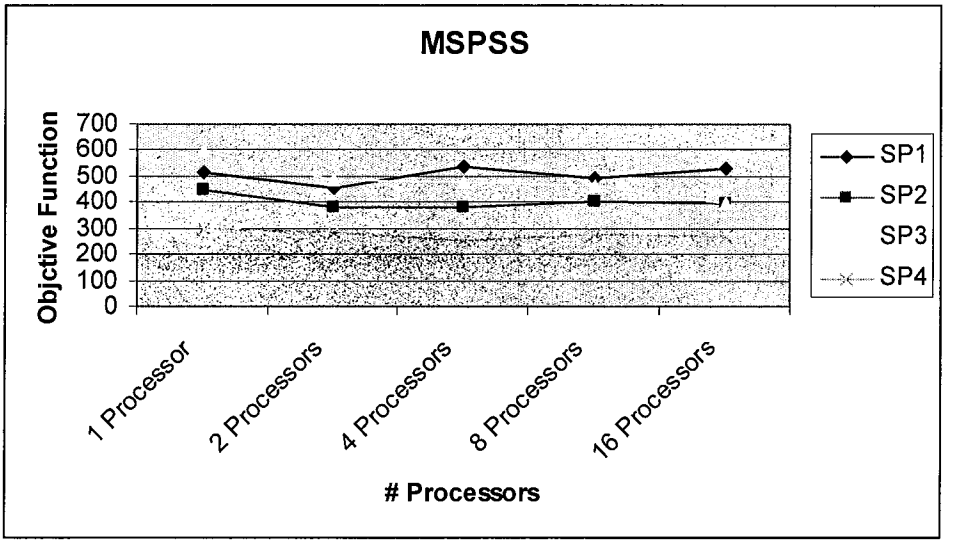

Figure 5.4: MSPSS objective functions for SP1, SP2, SP3 and SP4.

- Regarding the objective function obtained by MSPSS, we deduce the following findings:
    - SP3 and SP4 gave relatively higher quality solutions when the number of processors increases, but the objective function rates remained very close to the rates obtained by running the sequential scatter search algorithm.
    - For SP1 and SP2, MSPSS gave lower quality solutions when the number of processors increases.
    - We can say that MSPSS did not produce better quality results than the sequential scatter search algorithm.

Table 5.4 shows the results of executing EPSS for SP2 over 1-16 processors where we set PopSize = 1024, RefSize = 32.

Table 5.4: Results for EPSS, PopSize=1024, RefSet=32.

Semester	Processor#	S_{SE}	S_{CE}	S_{3E}	S_{4E}	R_V	Objective Function	Iteration#	Execution Time (Mins)	Parallel Efficiency
SP2	1	0	329	12	0	0	449	6	545	-
	2	0	314	7	0	0	384	13	622	43.8
	4	0	128	0	0	0	128	4	105	129.8
	8	3	432	15	0	0	1182	3	41	166.2
	16	4	418	17	0	0	1388	21	128	26.7

Figure 5.5 illustrates the parallel efficiency results obtained by running EPSS on PS2 over 1-16 processors based on wall clock time.

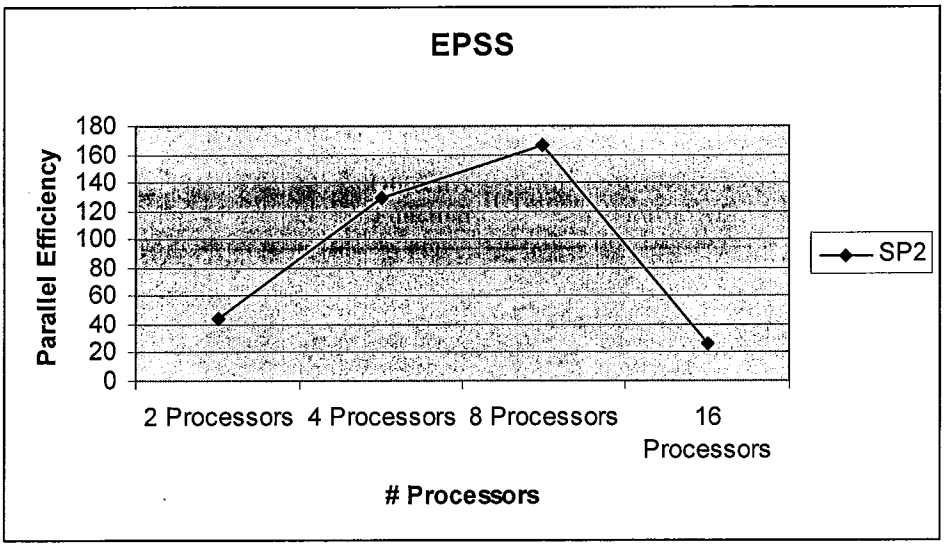

Figure 5.5: EPSS parallel efficiency.

- Studying the parallel efficiency rates, EPSS experimental results showed the following:

- EPSS parallel efficiency rates increased when the number of processors reached 4 and 8.
- The lowest parallel efficiency rate was recorded when the number of processor is 16, where the number of iterations increased dramatically.

Figure 5.6 illustrates the behavior of the objective function by running EPSS on PS2 over 1-16 processors.

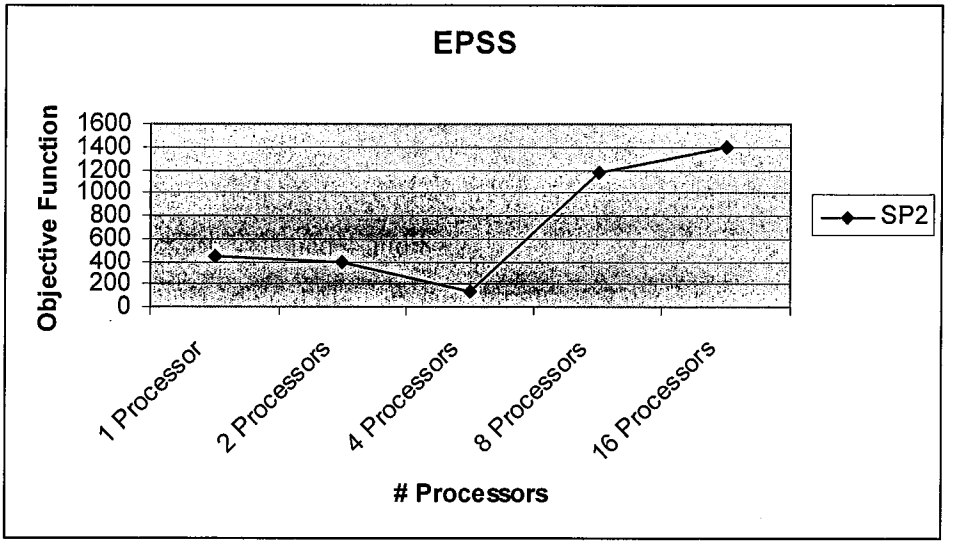

Figure 5.6: EPSS objective function for SP2.

- Regarding EPSS objective function results for SP2, we deduced the following:
  - EPSS solution quality increased when the number of processors ranged from 2 to 4 processors.

- For 8 and 16 processors, the solution quality decreased dramatically and the objective function reached its highest rate over 16 processors.
- Comparing sequential scatter search and EPSS, we found that sequential scatter search gives higher quality results than EPSS.

Figure 5.7 shows a graphical representation of the parallel efficiency rates obtained by running PSS, MSPSS and EPSS on SP2.

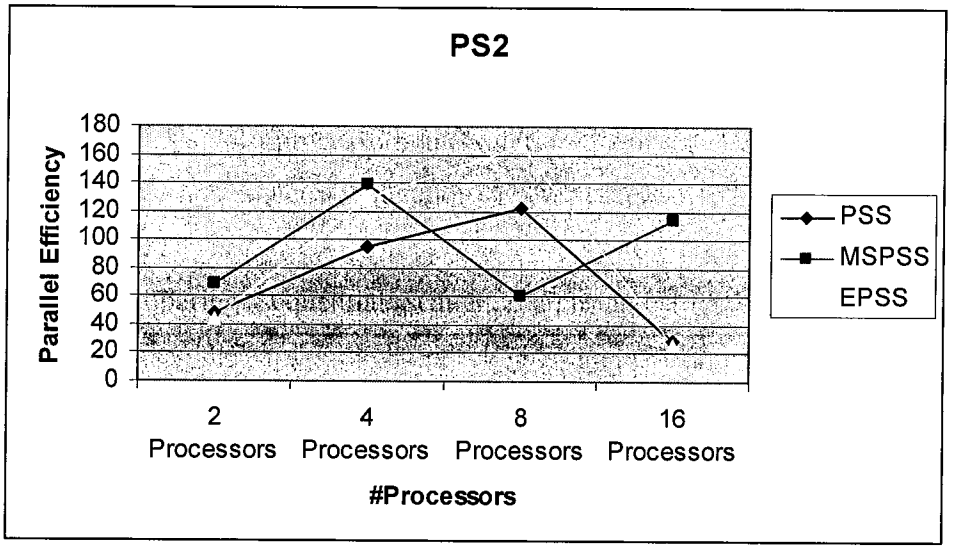

Figure 5.7: Parallel efficiency of PSS, MSPSS and EPSS for SP2.

Figure 5.8 shows a graphical illustration of the objective function rates obtained by running PSS, MSPSS and EPSS on SP2.

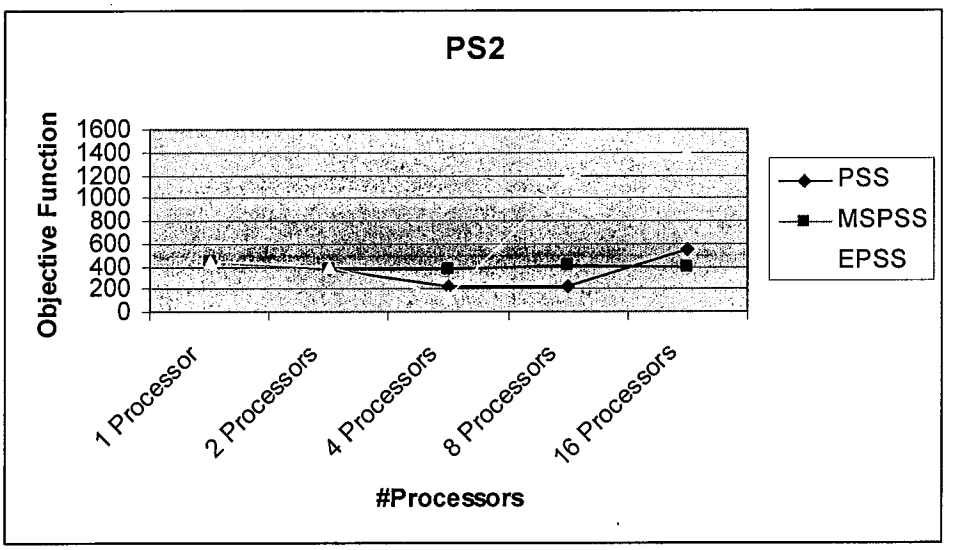

Figure 5.8: Objective function of SP2 by PSS, MSPSS and EPSS.

- Comparing PSS, MSPSS and EPSS, we found the following:
    - Both PSS and EPSS gave higher parallel efficiency rates when the number of processors ranges from 2 to 8 processors and lower parallel efficiency rates over 16 processors.
    - EPSS gave a lower solution quality than PSS and MSPSS when the number of processors increases, until it reached a very low quality rate over 16 processors.
    - Both PSS and MSPSS gave a solution quality relatively close compared to each other.

Figures 5.9 and 5.10 give a graphical illustration for PSS and MSPSS speed-up respectively.

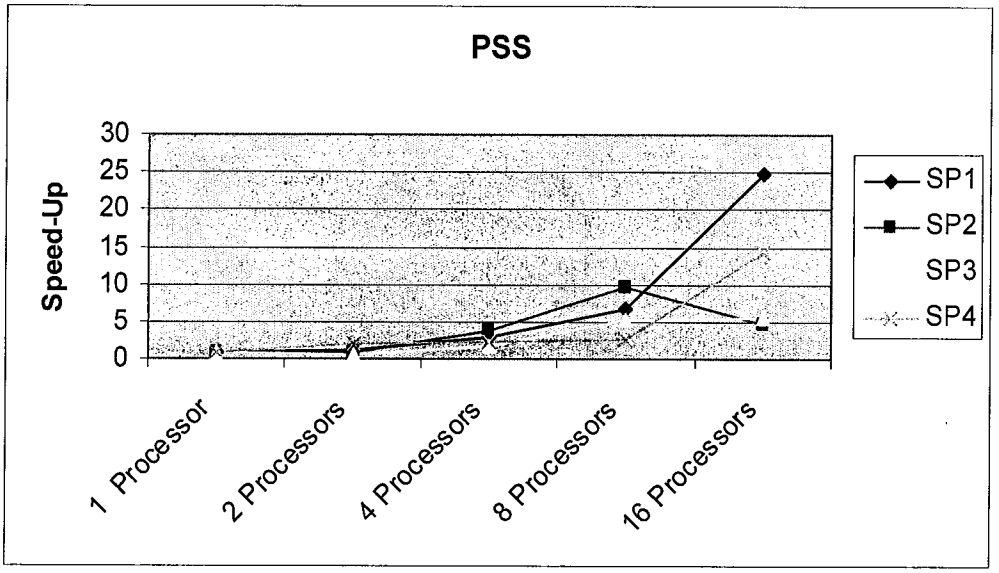

Figure 5.9: PSS speed-up.

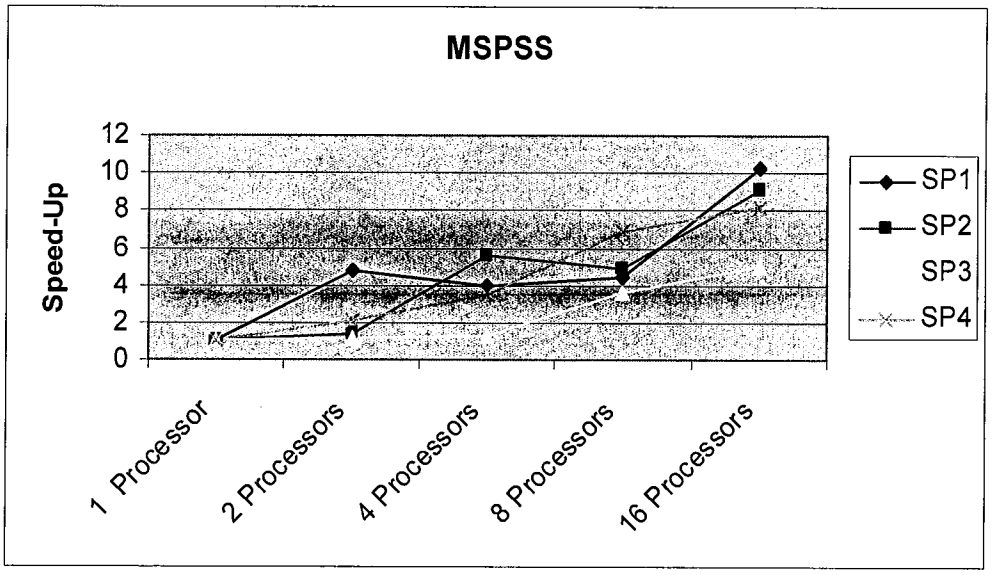

Figure 5.10: MSPSS speed-up.

Comparing PSS and MSPSS speed-up in figures 5.9 and 5.10, we deduce that, in general, both PSS and MSPSS speed-up tend to increase as the number of processors increase.

Table 5.5 illustrates both PSS and MSPSS average parallel efficiency rates for SP1, SP2, SP3 and SP4 over 2, 4, 8 and 16 processors.

Table 5.5: Average parallel efficiency rates for SP1, SP2, SP3 and SP4.

Program	2 Processors	4 Processors	8 Processors	16 Processors
PSS	59.4	67.2	75.23	77.36
MSPSS	118	89.3	62	65

PSS and MSPSS average parallel efficiency results of executing parallel scatter search are shown in Figure 5.11. This figure illustrates a graph for the average parallel efficiency obtained by running PS1, PS2, PS3 and PS4 over 1-16 processors based on wall clock time.

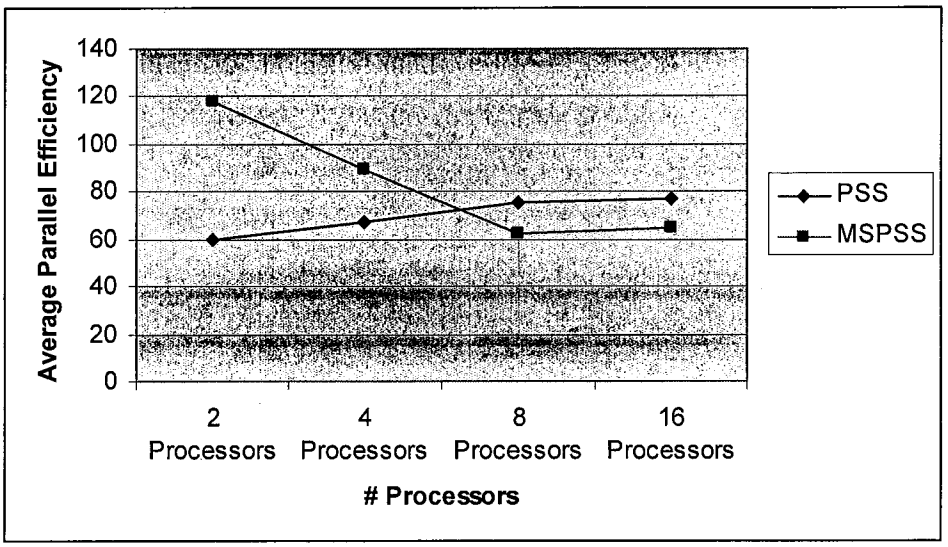

Figure 5.11: PSS and MSPSS average parallel efficiency.

- Comparing PSS and MSPSS average parallel efficiency, we deduced the following results:
    - MSPSS gives higher average rates than PSS for parallel efficiency when the number of processors is 2.
    - For 4 processors, MSPSS gives higher average rates than PSS but the difference in rates between PSS and MSPSS over 4 processors becomes remarkably smaller than the difference in rates between PSS and MSPSS over 2 processors.
    - PSS gives higher average rates than MSPSS for parallel efficiency when the number of processors ranges from 8 to 16 processors.
    - We can conclude that PSS gives better efficiency results than MSPSS when the number of processors increases.
    - PSS average parallel efficiency increases as the number of processors increases.
    - MSPSS average parallel efficiency decreases as the numbers of processors increases.

Table 5.6 shows the results of executing MSPSS and PSS respectively on SP1 over 1-16 processors where we set PopSize = 200 and RefSize = 20.

Table 5.6: Results for PSS & MSPSS on SP1, PopSize=200.

Prog	Processor#	Ref Set	S_{SE}	S_{CE}	S_{3E}	S_{4E}	R_V	Objective Function	Iteration#	Execution Time (Mins)	Parallel Efficiency
PSS	**1**	**20**	**0**	**496**	**16**	**0**	**0**	**656**	**11**	**321**	-
	2	**20**	**1**	**543**	**18**	**0**	**0**	**923**	**10**	**149**	**107.8**
	4	**20**	**0**	**444**	**20**	**0**	**0**	**644**	**18**	**134**	**60**
	8	**20**	**0**	**498**	**13**	**0**	**0**	**628**	**16**	**61**	**65.8**
	16	**32**	**0**	**210**	**1**	**0**	**0**	**220**	**2**	**18**	**111.5**
MSPSS	**1**	**20**	**0**	**496**	**16**	**0**	**0**	**656**	**11**	**327**	-
	2	**20**	**1**	**443**	**9**	**0**	**0**	**733**	**8**	**118**	**138.5**
	4	**20**	**1**	**507**	**21**	**0**	**0**	**917**	**10**	**79**	**103.5**
	8	**20**	**0**	**467**	**15**	**0**	**0**	**617**	**16**	**120**	**34**

	16	32	0	181	1	0	0	191	17	143	14.3

Figures 5.12 and 5.13 illustrate a graphical representation for MSPSS and PSS parallel efficiency and objective function of SP1 respectively over 1-16 processors.

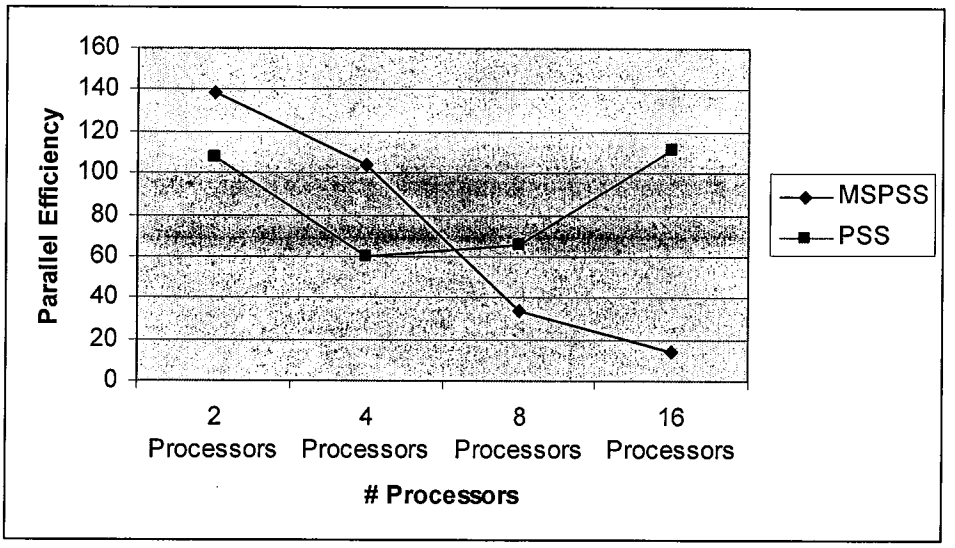

Figure 5.12: PSS and MSPSS parallel efficiency for SP1.

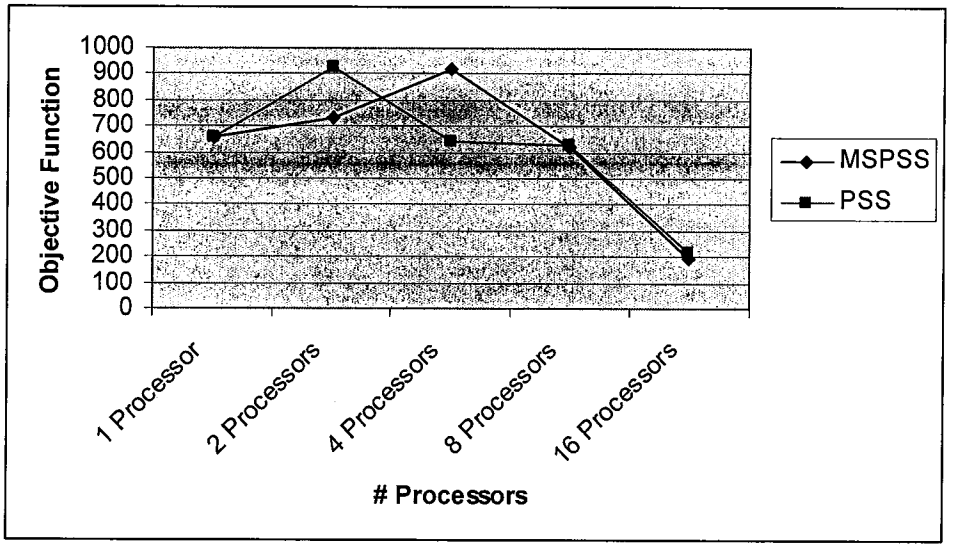

Figure 5.13: PSS and MSPSS objective function for SP1.

- Comparing PSS and MSPSS on PopSize = 200 and RefSize = 20, we found that:
    - PSS gives better efficiency than MSPSS over 8 and 16 processors yet MSPSS gives better efficiency than PSS over 2 and 4 processors.
    - PSS gives higher solution quality than MSPSS over 2 processors, lower solution quality than MSPSS over 4 processors and relatively close range of solution quality as MSPSS over 8 and 16 processors.
    - We deduce that on higher number of processors, 8 and 16, PSS takes less time than MSPSS and generates lower solution quality yet very close to those generated by MSPSS.

Table 5.7 illustrates the results of executing PSS for SP1 over 1-4 processors where we set PopSize = 100 and RefSize = 10.

Table 5.7: Results for PSS on SP1, PopSize =100, RefSet=10.

Semester	Processor#	S_{SE}	S_{CE}	S_{3E}	S_{4E}	R_V	Objective Function	Iteration#	Execution Time (Mins)	Parallel Efficiency
SP1	1	2	489	15	0	0	1039	6	45	-
	2	1	484	16	0	0	844	3	13	173
	4	2	416	15	0	0	966	8	16	70.3

Table 5.8 illustrates the results of executing PSS for SP1 over 1-8 processors where we set PopSize = 200 and RefSize = 20.

Table 5.8: Results for PSS on SP1, PopSize =200, RefSet=20.

Semester	Processor#	S_{SE}	S_{CE}	S_{3E}	S_{4E}	R_V	Objective Function	Iteration#	Execution Time (Mins)	Parallel Efficiency
SP1	1	0	496	16	0	0	656	11	321	-
	2	1	543	18	0	0	923	10	149	107.8
	4	0	444	20	0	0	644	18	134	60
	8	0	498	13	0	0	628	16	61	65.8

Table 5.9 illustrates the results of executing PSS for SP4 over 1-4 processors where we set PopSize = 100 and RefSize = 10.

Table 5.9: Results for PSS on SP4, PopSize =100, RefSet=10.

Semester	Processor#	S_{SE}	S_{CE}	S_{3E}	S_{4E}	R_V	Objective Function	Iteration#	Execution Time (Mins)	Parallel Efficiency
SP4	1	0	350	3	0	0	380	5	89	-
	2	0	300	2	0	0	320	24	191	23.3
	4	0	319	1	0	0	329	13	56	40

Table 5.10 illustrates the results of executing PSS for SP4 over 1-8 processors where we set PopSize = 200 and RefSize = 20.

Table 5.10: Results for PSS on SP4, PopSize =200, RefSet=20.

Semester	Processor#	S_{SE}	S_{CE}	S_{3E}	S_{4E}	R_V	Objective Function	Iteration#	Execution Time (Mins)	Parallel Efficiency
SP4	1	0	295	3	0	0	325	14	900	-
	2	0	320	1	0	0	330	13	426	105.6
	4	0	324	3	0	0	354	12	199	113
	8	0	349	5	0	0	399	19	156	72.1

Figures 5.14 and 5.15 show a graphical illustration of PSS objective function on SP1 and SP4 respectively using PopSize = 100, 200 and 1024, and RefSize = 10, 20 and 32.

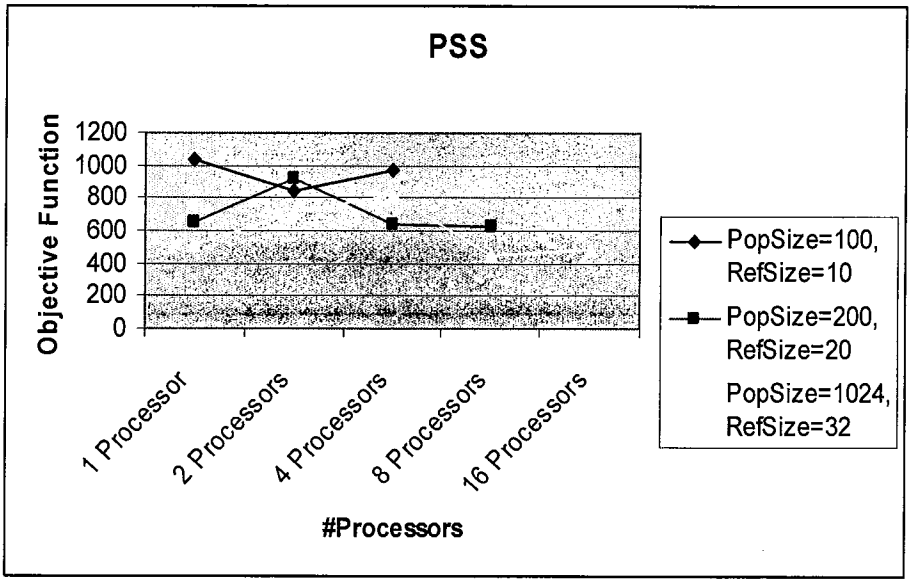

Figure 5.14: PSS objective function on SP1, PopSize=100,200,1024, RefSize=10,20,32.

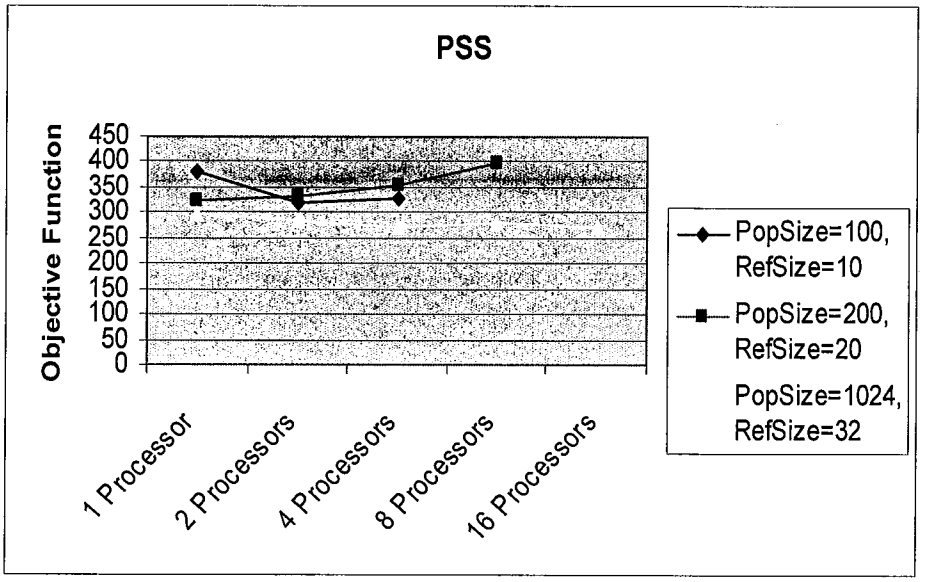

Figure 5.15: PSS objective function on SP4, PopSize=100,200,1024, RefSize=10,20,32.

- Studying the behavior of PSS objective function over different population sizes and reference set sizes, we deduced the following findings:
    - SP1: on 1 and 4 processors, PSS gave higher solution for PopSize=200 and RefSize=20 than for PopSize=100 and RefSize=10.
    - SP1: on 1, 2 and 4 processors, PSS gave higher solution for PopSize=1024 and RefSize=32 than for PopSize=200 and RefSize=20.
    - SP4: on 2 and 4 processors, PSS gave higher solution for PopSize=100 and RefSize=10 than for PopSize=200 and RefSize=20.
    - SP4: on 1, 2, 4 and 8 processors, PSS gave higher solution for PopSize=1024 and RefSize=32 than for PopSize=200 and RefSize=20.
    - Thus, except for SP1 on 4 processors, PSS generated higher solution quality for PopSize=1024 and RefSize=32.
    - As conclusion, PSS yielded higher solution quality for larger population size and reference set size than on smaller population and reference set sizes.

5.3 Experimental Limitations

A number of experimental limitations encountered us. These limitations are as follows:

1. PSS, MSPSS and EPSS do not fully respect the reference set generation and update methods of the sequential scatter search. This is due to the parallelization of the scatter search algorithm where the population size and subsets are divided among *n_pr* processors and thus different reference sets are generated on every processor. Moreover, MSPSS generates partial reference sets such that gathering all *n_pr* reference sets constitutes the total reference set size.
2. Limited scalability of PSS, MSPSS and EPSS. Over larger number of processors, 32 or 64 processors, we need to increase both population size and reference set size which is not consistent with the sequential scatter search since it takes long time for running large problems. Besides, increasing the number of processors from 8 processors to 16 processors, we where obliged to increase the size of the population size and to set the reference set size equal to 32 instead of 20.
3. Due to some random selection cases, PSS runs more iterations than sequential scatter search in an unpredictable manner, which leads to reduction in the parallel efficiency.
4. Sometimes we obtain some unexpected peculiar results that fall outside pattern implied by the other encountered results, and lead to reduction in solution quality and/or efficiency rates. For example in Table 5.2, PSS generated, for SP2, 4 iterations over 4 processors, 3 iterations over 8 processors, and the objective function was 213 for

both runs, while over 16 processors, PSS generated 21 iterations and gave a higher objective function value = 538. Also in Figure 5.6, PSS generated, for SP2, 16 iterations over 8 processors and gave an objective function value = 628, while over 16 processors, PSS generated only 2 iterations and gave an objective function value = 220.

5. We encounter a threat to validity of results due to the limited number and size of subject problems. We were able to get only 4 subject problems with limited size of courses and student enrollments as described in Table 5.1.

Chapter 6
Conclusion

In this paper, we implemented three different parallel scatter search algorithms to solve the exam timetabling problem. These algorithms are 1) a fully parallel algorithm based on a bottom-up tree-based communication system called Parallel Scatter Search (PSS), 2) a master- slave algorithm called Master-Slave Parallel Scatter Search (MSPSS) and 3) a replicated algorithm called Embarrassingly Parallel Scatter Search (EPSS). The experimental results proved that the proposed parallel scatter search algorithms gave higher solution quality in less processing time since they explored larger parts of the search space within reasonable time.

Concluding remarks are detailed as follows:

1. Parallel scatter search algorithms PSS, MSPSS, EPSS reduced the execution time with respect to the sequential scatter search algorithm
2. Parallel scatter search algorithms PSS, MSPSS, EPSS allowed employing higher demanding features of scatter search such as combining subsets with 3 elements.
3. PSS is more scalable than MSPSS since we were able to investigate larger reference set size on 16 processors.
4. Clearly, there is no steady behavior or a clear pattern for PSS and MSPSS as far as efficiency and objective function, although over 16 processors, PSS tends to give higher efficiency and solution quality.
5. In general, PSS gave highest parallel efficiency rates and highest solution quality for larger population size and reference set size over 8 and 16 processors.

6. PSS yielded higher solution quality for larger population size and reference set size than on smaller population and reference set sizes.
7. Both PSS and MSPSS speed-up tend to increase as the number of processors increases.
8. PSS average parallel efficiency increases as the number of processors increases.
9. MSPSS average parallel efficiency decreases as the numbers of processors increases.
10. PSS generated higher solution quality for larger population size and reference set size.
11. Comparing sequential scatter search and EPSS, we found that sequential scatter search gives higher quality results than EPSS.
12. PSS and MSPSS give higher solutions quality than the sequential scatter search algorithm.
13. Due to parallelization and increase in population size, we have the opportunity to explore larger areas of search space and thus, to find better quality timetabling solutions.
14. Thus, parallel scatter search algorithms are necessary for handling/solving large timetabling problem within reasonable time.

References

Adenso-Diaz, B., Garcia-Carbajal, S. & Lozano, S. (2006). An empirical investigation on parallelization strategies for scatter search. *European Journal of Operational Research,* 169(2), 490-507.

Alba, E., Luque, G. & Luna, F. (2006). Workforce planning with parallel algorithm. *Journal of Mathematical Modelling and Algorithms, 6(3), 509-528.*

Bozejko, W. & Wodecki, M. (2008). Parallel Scatter Search Algorithm for the Flow Shop Sequencing Problem. *Parallel Processing and Applied Mathematics* (Vol. 4967, pp. 180-188). Heidelberg: Springer Berlin.

Burke, E., Bykov, Y., Newall, J. & Petrovic, S. (2004). A time-predefined local search approach to exam timetabling problems. *IIE Transactions on Operations Engineering*, 36(6), 509-528.

Burke, E.K., Bykov, Y. & Petrovic, S. (2001). A Multicriteria approach to examination timetabling. In: E. Burke, & W. Erbe, (Eds.) *Practice and Theory of Automated Timetabling* III, 2079, pp. 118-131. Berlin Heidelberg New York: Springer-Verlag.

Burke, E.K., Eckersley, A.J., McCollum, B., Petrovic, S. & Qu, R. (Eds.). (2005). Similarity measures for exam timetabling problems. *proceedings of the 1st multidisciplinary conference on scheduling: theory and applications,* (MISTA 2003), Nottingham, August 13-16, 2003, pp. 120-136.

Burke, E., Elliman, D., Ford, P. & Weare, B. (1996). Examination timetabling in british universities: the practice and theory of automated timetabling. *Lecture Notes in Computer Science*, 1153, pp. 76-90, Springer-Verlag.

Burke, E., Hart, E., Kendall, G., Newall, J., Ross, P. & Schulenburg, S. (2003). *Handbook of meta-heuristics*. Kluwer Academic Publishing Group.

Burke, E., Kendall, G. & Soubeiga, E. (2003). A tabu-search hyperheuristic for timetabling and rostering. *Journal of Heuristics*, 9(6), 451-470.

Burke, E., MacCarthy, B., Petrovic, S. & Qu, R. (2006). Multiple-retrieval case based reasoning for course timetabling problems. *European Journal of Operational Research*, 57(2), 148-162.

Burke, E.K., Newall, J.P. & Weare, R.F. (1996). A memetic algorithm for university exam timetabling. In: E. Burke, & P. Ross, (Eds.) *Practice and Theory of Automated Timetabling*, 1153, pp. 241-250. Berlin Heidelberg New York: Springer-Verlag.

Burke, E. & Petrovic, S. (2002). Recent research directions in automated timetabling. *European Journal of Operational Research*, 140, pp. 266-280.

Carter, M. & Johnson, D.G. (2001). Extended clique initialization in examination timetabling. *Journal of the Operational Research Society*, 52(5), 538-544.

Carter, M. & Laporte, G. (1996). Recent developments in practical examination timetabling. In: E. Burke, P. Ross (Eds.) *Practice and Theory of Automated Timetabling*, 1153, (pp. 3-21). Berlin Heidelberg New York: Springer-Verlag.

Cheng, E., Kleinberg, R.P., Kruk, S.G., Lindsey, W.A. & Steffy, D.E. (2004). A strictly combinatorial approach to a university exam scheduling problem. *Congr. Numer.*, 167, 121 – 132.

Crainic, T. (2005). *Parallel Computation, Co-operation, Tabu Search.* US: Springer.

David, P. (1998). A constraint-based approach for examination timetabling using local repair techniques. In: E. Burke, & M. Carter, (Eds.) *Practice and Theory of Automated Timetabling II*, 1408, (pp. 169-186). Berlin Heidelberg New York: Springer-Verlag.

Di Gaspero, L. & Schaerf, A. (2003). Multi-neighbourhood local search with application to course timetabling. In: E. Burke, & P. DeCausmaecker, (Eds.) *Practice and Theory of Automated Timetabling IV*, 2740, (pp. 262-275). Berlin Heidelberg New York: Springer-Verlag.

Di Gaspero, L. & Schaerf, A. (2001). Tabu search techniques for examination timetabling. In: E. Burke, & W. Erben, (Eds.) *Practice and Theory of Automated Timetabling III*, 2079, (pp. 104-117). Berlin Heidelberg New York: Springer-Verlag.

El-Sayed, S. Abd El-Wahed, W. & Ismail, N. (2008). *A hybrid genetic scatter search algorithm for solving optimization problems.* Master's Thesis. Cairo University.

Erben, W. (2001). A grouping genetic algorithm for graph colouring and exam timetabling. In: E. Burke, & W. Erben, (Eds.) *Practice and Theory of Automated Timetabling III*, 2079, (pp. 132-156). Berlin Heidelberg New York: Springer-Verlag.

Glover, F., Laguna, M. & Marti, R. (2000). fundamentals of scatter search and path relinking. *Control and Cybernetics*, 39, pp. 575-589.

Johnson, D. (1990). Timetabling university examinations. *Journal of the Operational Research Society*, 140, pp. 266-280.

Laguna, M. (2002). Scatter search. In: P.M. Pardalos, & M.G.C. Resende (Eds.) *Handbook of Applied Optimization* (pp. 182-193). Oxford: University Press.

Laguna, M. & Marti, R. (2003). *Scatter search: methodology and implementations in C*. London: Boston Dordrecht.

Lofti, V. & Cerveny, R. (1991). Final-exam-scheduling package. *Journal of the Operational Research Society*, 42, pp. 205-216.

Lopez, F.G., Batista, B.M & Moreno-Perez, J.A. (2003). Parallelization of the scatter search for the p-median problem. *Parallel Computing*, 29, pp. 575-589.

Lopez, F.G., Torres, M.G., Batista, B.M., Moreno-Perez, J.A. & Moreno-Vega, J.M. (2006). Solving features subset selection problem by a parallel scatter search. *European Journal of Operational Research*, 196, pp. 477-489.

Mansour, N., and Isahakian, V. (2007). Evolutionary algorithm for exam scheduling. Int. Symp. On Innovations in Intelligent Systems and Applications, Istanbul, June 20-23, pp. 283-287.

Mansour, N., Tarhini, A. & Isahakian, V. (2003). *Three-phase simulated annealing algorithms for exam scheduling.* Proceeding of the ACS/IEEE International Conference on computer systems and applications. 14-18 July 2003.

Marti, R. (2006). Scatter search-wellsprings and challenges. *European Journal of Operational Research*, 169, pp. 351-358.

Marti, R., Laguna, M. & Campos, V. (2005). Scatter search vs. genetic algorithms: an experimental evaluation with permutation problems. In C. Rego, B. Alidaee (Eds.) *Meta-heuristic Optimization Via Adaptive Memory and Evolution: Tabu Search and Scatter Search*, (pp. 263-282). Kluwer: Academic Publishers.

Marti, R., Laguna, M., Campos, V. & Lourenço, H. (2000). Assigning proctors to exams with scatter search. In M. Laguna, J. L. González-Velarde (Eds.) *Computing Tools for Modeling, Optimization and Simulation: Interfaces in Computer Science and Operations Research*, (pp. 215-227). Kluwer: Academic Publishers.

Marti, R., Laguna, M. & Glover, F. (2006). Principles of scatter search. *European Journal of Operational Research*, 169, pp. 359-372.

Nepal, T., Melville, S.W. & Ally, M.I. (1998). A brute force and heuristics approach to tertiary timetabling. In: E. Burke, & M. Carter (Eds.) *Practice and Theory of Automated Timetabling III*, 1408, (pp. 254-265). Berlin Heidelberg New York: Springer-Verlag.

Paechter, B., Rankin, R.C. & Cumming, A. (1998). Improving a lecture timetabling system for university-wide use. In: E. Burke, & M. Carter (Eds.) *Practice and Theory of Automated Timetabling II*, 1408, (pp. 156-165). Berlin Heidelberg New York: Springer-Verlag.

Qu, R., Burke, E. K., McCollum, B., Merlot, L.T.G. & Lee, S.Y. (2007). A survey of search methodologies and automated system development for examination timetabling. *automated scheduling, optimization and planning (ASAP) group*, School of Computer Science, University of Nottingham, U.K.

Rich, D.C. (1996). A smart genetic algorithm for university timetabling. In: E. Burke, & P. Ross (Eds.) *Practice and Theory of Automated Timetabling*, 1153, (pp. 182-197). Berlin Heidelberg New York: Springer-Verlag.

Schaerf, A. (1999). A survey of automated timetabling. *Artificial Intelligence Review*, 13, pp. 87-127.

White, G.M. & Chan, P.W. (1979). Towards the construction of optimal examination timetables. *INFOR*, 17, pp. 219-229.

White, G.M. & Xie, B.S. (2001). Examination timetables and tabu search with longer term memory. In: E. Burke, & M. Carter (Eds.) *Practice and Theory of Automated Timetabling III*, 2079, (pp. 85-103). Berlin Heidelberg New York: Springer-Verlag.

Wren, A. (1999). Scheduling, timetabling and rostering - a special relationship? In: E. Burke, & P. M. Ross (Eds.) *Practice and Theory of Automated Timetabling*, 1153, (pp. 46-75). Berlin Heidelberg New York: Springer-Verlag.

Appendix A: Table of Symbols

Symbol	Description
S_{SE}	Number of students having simultaneous exams
S_{CE}	Number of students having consecutive exams per day
S_{3E}	Number of students having multiple 3 exams per day
S_{4E}	Number of students having multiple 4 exams per day
R_v	Number of classrooms violated
Ψ	Predefined classroom capacity
D	Number of exam days
E	number of exam period per day
II	number of exam periods ($\prod$ = D * E)
α,	user-defined weight for students having simultaneous exams
φ	user-defined weight for students having consecutive exams per day
β	user-defined weight for students having multiple 3 exams per day
σ	user-defined weight for students having multiple 4 exams per day
γ	user-defined weight for room capacity violations
ρ_{xy}	Assignment of room y to exam x subject to room capacity Ψ_y and number of students enrolled in exam x
PopSize	Population size
RefSize	Reference set size
n_pr	Total number of processors in execution