Parallel Numerics '05, 97-107 M. Vajteršic, R. Trobec, P. Zinterhof, A. Uhl (Eds.)
Chapter 5: Optimization and Classification ISBN 961-6303-67-8

Sequential and Parallel Algorithms for the Shortest Common Superstring Problem

Xuan Liu[1,*], **Ondrej Sýkora**[2]

[1] [2]Department of Computer Science, Loughborough University, Loughborough, Leicestershire LE11 3TU, The United Kingdom

We design sequential and parallel genetic algorithms, simulated annealing algorithms and improved greedy algorithms for the shortest common superstring problem(SCS), which is to find the shortest string that contains all strings from a given set of strings. The SCS problem is NP-complete [7]. It is even MAX SNP hard [2] i.e. no polynomial-time algorithm exists, that can approximate the optimum to within a predetermined constant unless P=NP. We compare the above mentioned algorithms applied to several randomly generated test cases. The test results show the superiority of the parallel island genetic algorithm.

1 Introduction

The shortest common superstring problem has applications in both computational biology and data compression [11, 12, 14, 16]. DNA sequencing is the task of determining the sequence of nucleotides in a molecule of DNA. These nucleotides are one of adenine, cytosine, guanine, and thymine, and are typically represented by the alphabet {a,c,g,t}. Current laboratory procedures can directly determine the nucleotides of a fragment of DNA up to about 600 nucleotides long. Once the nucleotides of all of the fragments have been determined, the sequence assembly problem is the computational task of reconstructing the original molecule from the overlapping fragments. The shortest superstring problem is an abstraction of this problem [1].

Let $s_1 = a_1 \ldots a_r$ and $s_2 = b_1 \ldots b_s$ be strings over some finite alphabet Σ. We say that s_1 is a substring of s_2 if there is an integer $i \in [0\ ,\ s-r]$ such

*Corresponding author. E-mail: [1]x.liu@lboro.ac.uk,[2]o.sykora@lboro.ac.uk

that $a_j = b_{i+j}$ for $1 \leq j \leq r$. We also say in this case that s_2 is a superstring of s_1.

An instance of the *shortest common superstring problem* (SCS) is a set of strings $S = \{s_1,...,s_n\}$ over a finite alphabet Σ and the problem is to find a minimum length string that is a superstring of every $s_i \in S$.

Several linear approximations for the SCS problem have been proposed. Blum et al. [2] were the first to introduce an approximation algorithm that produces a solution within a factor of 3 from the optimum. This was improved by Jiang et al. [9] who have obtained a $2\frac{2}{3}$ approximation and by Z Sweedyk [17] with $2\frac{1}{2}$ approximation. Anyway these approximation algorithms are not eaily implementable and in practise we use either heuristics or some kind of machine learning algorithms.

Holland originated genetic algorithms (GAs) in the 1960s. GAs are based on the principles of natural selection and adaptation and are claimed to be able to explore good solutions relatively quickly in a large and complicated search space. The power of the algorithms comes from the mechanism of evolution, which allows searching through a huge number of possibilities for solutions.It is that chromosomes are the information carriers and that the evolution process works at the chromosome level through reproduction. The reproduction can be made by either combining chromosomes from the parents to produce offspring, a process called crossover, or by a random change occurring in the chromosome pattern, termed mutation.

A GA creates an initial population of solutions at beginning. Then, the GA evaluates fitness function to all members of population, to characterise them from the most fit to the least fit. Afterwards, genetic operators transform the parent chromosomes to their offspring according to the criteria of fitness. The GA repeats the processes of selection, crossover and mutation to artificially simulate genetic operations.

Zaritsky et al. [19] recently achieved good results by using the GA approach. We do not know about any other sequential or parallel GA application for the shortest common superstring problem.

In this paper, we focus on design parallel island GA and compare another three sequential and parallel approaches, which are Greedy algorithm, Sort and Merge algorithm and Simulated Annealing algorithm. The experimental results show that the parallel island GA produces the best results out of those four parallel algorithms.

2 Description of the Algorithms

2.1 The Sequential and Parallel Genetic Algorithm (GA) for SCS

Given a set of strings as the input to the SCS problem, the GA generates an initial population of random candidate solutions $ind_1, \ldots, ind_m$ after preprocessing, which is elimination of the substrings from the input. Let m be the population size. Each individual of initial population gets the order of the set of string blocks. The *fitness* function is defined as *Maximal Total Length* − *Current Total Length*, where *Maximal Total Length* $= \sum_{i=1}^{n} |s_i|$. To minimise the length of the superstring we need to maximise the fitness function.

GAs are complex algorithms controlled by many parameters and the results produced by them heavily depend on setting these parameters correctly. In our GA we use the following *crossover* operation for the SCS: In every generation (iteration), the GA randomly, with a certain probability p_{cross}, (according to fitness value) chooses two parental individuals, ind_1 and ind_2. Then a part of their strings at a random position is exchanged to form two offspring. Afterwards, the *mutation* is executed by exchange a block on each individual with some probability p_{mut} in the new population. This procedure is repeated until m new offspring are generated.

The algorithm terminates when the number of generations n_g reaches a preset value. In this algorithm, we set n_g equal to 3000.

Parallel computing has been a valuable tool for improving running time and enlarging feasible sizes of problems and it is an economic and strategic issue. Strong efforts have been put into developing standards for parallel programming environments, such as PVM (Parallel Virtual Machine) [15], MPI (Message Passing Interface) [13] and HARNESS [6], etc. Probably the most common is PVM.

In this article we used PVM, as it is standard and it frees the algorithm designer from load balancing, resource control, fault tolerance and other problems with parallel software.

One of the frequently cited advantages of using GAs is their "natural" parallelism. *Global Model* PGAs use parallel techniques to speed up the operation of the GA without changing the basic operation of the sequential GA. In this model, a single global population is always employed without locality considerations. In fact, the *Global Model* does not achieve good results due to machine dependency considerations.

Another famous parallelized GAs approach is *Cellular Model* [18]. The *Cellular Model* seeks to exploit fine grained parallel architectures. In general, the *Cellular Model* assigns one chromosome per processor and limits selection

and crossover to local neighborhood. It achieves similar effect of isolation as found in the *Island Model* where the isolation is set by distance.

In this paper we use the *Island Model* to exploit a coarse-grain parallelism, which can be easily extended to a distributed system. The basic idea of *Island Model* is to distribute the total population among the available processors, and to execute a classical GA in each sub-population. Each processor is using same GA but independently of the others. Cantú-Paz [3, 4] also calls it *multiple-population coarse grained GA.*

Each GA is usually started with a different random seed. Therefore, every few generations, sub-populations could swap few individuals. We call this process *migration.*

In our PVM GA for SCS problem, each processor executes the sequential algorithm. After each generation there is a possibility that two processors will share some "genetic material". We chose 1% opportunity that two of the available processors, randomly chosen, will swap the migrants. *Migrants* were selected with the GA's selection function. Also, *migrants* accepted from the other processor replaced the worst fit individuals with the ones just received. These parameters were achieved in experimental way.

2.2 The Sequential and Parallel SA for SCS

Simulated Annealing (SA) is an advanced Local Search method which finds its inspiration from the physical annealing process studied in statistical mechanics [10]. The SA algorithm repeats an iterative procedure that looks for the better configurations while offering the possibility of accepting worse configurations. The SA algorithm provides opportunities to jump out from local minima. All candidate solutions of the problem are modelled as possible configurations of a thermal system. Therefore, the parameter space S becomes the space of all configurations. The energy E of the system, relies on the current configuration. The optimal solution corresponds to the minimum energy configuration. According to the Boltzmann distribution, given a temperature T, the probability of the system being in a certain configuration u is:

$$\pi_T(u) = \frac{e^{\frac{-E(u)}{kT}}}{\sum_{v \in S} e^{\frac{-E(v)}{kT}}} \tag{1}$$

where k is the Boltzmann's constant, and sum is taken over all configurations.

There are several important components in our SA algorithms. Let $f(s)$ stand for the length of the current superstring s. Our implementation of temperature reduction was proposed by Huang et al. [8]. It reduces the tem-

perature according to the length of Markov chains that the length of a chain is redetermined at each temperature level. The new temperature T_{new} is set to $T_{old}e^{-\frac{\lambda T_{old}}{\sigma}}$, where λ is randomly chosen constant from $(0, 1)$. The parameter σ controls the reduction ratio. Parameter L stands for the number of iterations with a temperature. The pseudocode of our sequential SA algorithm for SCS follows.

Algorithm 1 [Sequential SA algorithm for SCS]

```
1: randomly generate an initial string list s_c from the string set S
2: set an initial T = T_max
3: set λ and σ
4: while termination = false do
5:   for 0 ≤ i < L do
6:     randomly choose string list s_new
7:     Δ = f(s_new) − f(s_c)
8:     if Δ ≤ 0 or ((Δ > 0) and ( e^(−Δ/T) ) is verified ) then
9:       s_c ← s_n
10:    end if
11:    if T < T_min then
12:      termination = true
13:    end if
14:  end for
15:  T ← T(e^(−λT/σ))
16: end while
17: output
```

Our parallel SA algorithm is implemented in PVM. The idea of parallelism is borrowed from the *Island Model* genetic algorithm. Due to the temperature plays a crucial role in SA, we decided to run the sequential SA algorithm independently on each worker. After a while, all workers halt the computations and the master randomly chooses two of them to exchange their current temperatures. As long as they receive the temperature from the other, they continue working. Each worker stops working and returns its best result to the master when it reaches the termination-condition. Otherwise, the workers should exchange their current temperatures with others and carry on working. Here, the number of iterations in each stage is randomly generated.

2.3 The Sequential and Parallel Greedy algorithms for SCS

The *Greedy* algorithm is a simple and fast sequential approximation of a shortest superstring. Given a non-empty set of strings $S = \{s_1, ..., s_n\}$, repeat

the following steps until S contains only one string: Select a pair of strings $s', s'' \in S$ that maximizes overlap between s' and s'', remove s' and s'' from S replacing them with the merge of s' and s''.

We improved the *Greedy* Algorithm for SCS as follows: *Sort and Merge* (SM) algorithm: Let overlap between two strings s_x and s_y be of the length r. Produce superstring and r for any pair of distinct strings in the set of strings S. Sort the pairs in nonincreasing order according to r; if the superstrings s_{xy} and s_{wz} have the same length overlap, we prefer the superstring with smaller length. Combine the strings of S according to this order to create a superstring for S.

In our experiments, the *SM* algorithm always produced shorter common superstring than the *Greedy* algorithm. Moreover the *SM* algorithm was faster than the *Greedy* algorithm.

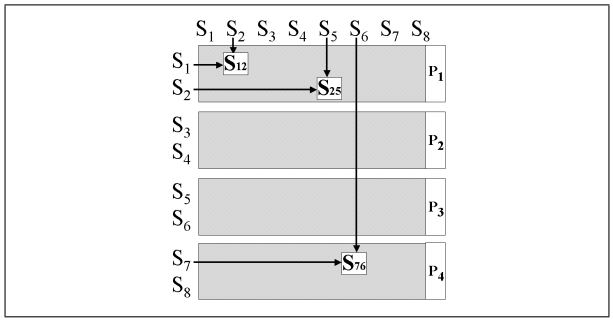

Figure 1: Arrange the combination of each two strings into processors.

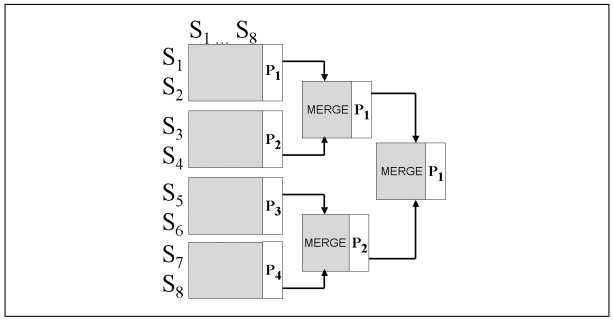

Figure 2: Merge the results into one list until a whole sorted list is formed.

As we described above, our *SM* algorithm has three main parts: produce the superstring s_{xy} for each two distinct strings s_x, s_y, sort the pairs of strings according to length of s_{xy} and combine them. Only the first and the second parts of this algorithm are suitable for parallelisation. Given a non-empty set

Strings	Greedy Results	SM Results	GA Results	SA Results
10	127bits	125bits	120bits	130bits
20	247bits	245bits	243bits	250bits
30	391bit	387bits	390bits	408bits

Table 1: The average test results for different problem sizes.

Strings	Greedy Time	SM Time	GA Time	SA Time
10	0.25ms	0.15ms	3593ms	70ms
20	0.9ms	0.65ms	14118ms	599.32ms
30	2ms	1.89ms	32831ms	2117.8ms

Table 2: The average running time for different problem sizes.

of strings $S = s_1, ..., s_n$, e.g. $n = 8$, arrange the combinations of each two strings into processors as in Figure 1. In this example, we have 4 processors.

After this, the processors will merge their results into one list until a whole sorted list is formed(see Figure 2).

Afterwards, the processor which contains the whole list creates the superstring using the idea which was described in the sequential algorithm.

3 Test Results and Discussion

The Table 1 and Table 2 show our experiments for the four sequential algorithms run on different number of strings. The strings were randomly generated by our testing platform.

We tested sets of 10, 20 and 30 strings, each string with length from 10 to 20 bits. For each of the randomly generated problem instances each algorithm was run 5 times. We discarded the worst results and got the average results from others. The configurations of the sequential GA were: the total population size was 30. The probability of crossover was 0.01, the probability of mutation probability was 0.6, the number of generations was 3000. The parameters of the sequential SA were: initial temperature was 100, the temperature reduction factor was 0.9. the minimum temperature was 0.001. The Table 1, contains the best average results, which is the achieved superstring length, and the corresponding running time is in Table 2.

From experiments for our sequential algorithms we can see that our GA can get the best results when the input strings are not more than 30. While the problem size increased, the population size and total generation number

String Blocks	PVM Greedy	PVM SM	PVM GA	PVM SA
10	130bits	127bits	123bits	131bits
20	249bits	247bits	243bits	253bits
30	385bits	381bits	375bits	382bits

Table 3: The PVM test results and running time for different problem sizes.

were not changed which meant that the GA's searching abilities decreased. Our results for GA confirmed the results obtained by [19]. Our SM algorithm can get very good results from the efficiency point of view. On the other side, the Greedy and SA did not perform very well. For longer strings one can first use SM-algorithm and then using the achieved solution in the population apply GA. Similar strategy can be used for parallel genetic algorithm.

Our PVM implementation of the algorithms have been run on a cluster of 20 Sun ULTRAsparc 5 workstations running Debian GNU/Linux. They are connected with 100Mbit Ethernet using Cisco 2950 switches. We tested the sequential and parallel version Greedy, SM, GA and SA algorithms. Both parallel algorithms used 8 processors. In the parallel island model GA algorithm, we set the migration rate to 1% and kept the same configurations as the sequential GA. We used the same testing methods as for the sequential algorithms.

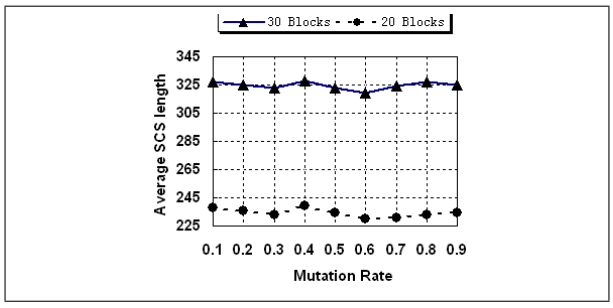

Figure 3: Mutation rate efficiency.

From our test (see Table 3) results one can see that the parallel GA produced the best results among these four parallel implementations. The parallel SA algorithm gets better results than its sequential counterpart due to the parallel algorithm enlarges their total searching space. The parallel Greedy and SM algorithms can not achieve better results than their sequential versions. On the other side, the running time we achieved for both of them was not better. The reason is that the communication time increased the total running

time.

One can also see that mutation rate influences the results of parallel and sequential GA algorithm (a known fact, see [4]).

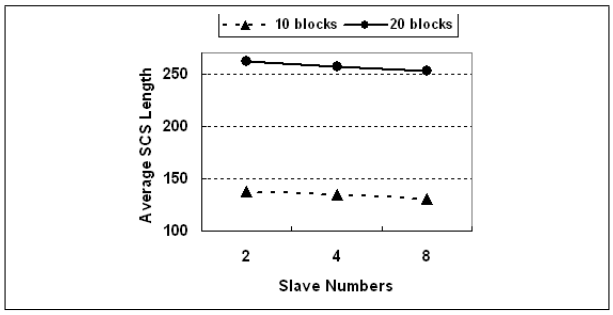

Figure 4: Worker numbers and the results.

From the Figure 3, we can see the better results always happen when we have mutation rate equal to 0.6.

We also observed that the worker numbers can affect the results of the parallel SA algorithm (see Figure 4). The more workers, the shorter SCS was produced. Possible explanation is that the parallel SA algorithm has bigger searching space than sequential SA algorithm and this ability is used efficiently. Another explanation could be that the exchange of current temperatures on different processors helps the searching process.

4 Conclusions

In this paper we designed sequential Greedy, GA and SA algorithms for the SCS problem. We also created a sequential SM algorithm and its parallel version, parallel island GA and parallel SA algorithms for the SCS problem. Comparison of all four parallel algorithms shows the superiority of the parallel island GA algorithm. Our results suggest application of the following simple strategy to get good results: first, use SM algorithm sequentially and the result include into the starting population and then apply our PVM GA. In the journal article we will discuss how the number of processors influences the quality of results as well as possible improvements of the GA for longer strings.

References

[1] Armen, C., Stein, C., A $2\frac{2}{3}$-approximation algorithm for the shortest superstring problem, *CPM*, 1996, 87-101.

[2] Blum, A., Jiang, T., Li, M., Tromp, J., Yannakakis M., Linear approximation of shortest superstrings, *JACM* **41** (1994), 634-647.

[3] Cantú-Paz, E., Implementing fast and flexible parallel genetic algorithms. *Handbook of Practical Genetic Algorithms, Volume 3.* Editor: Chalmers, L., CRC Press, 1999.

[4] Cantú-Paz, E., A survey of parallel genetic algorithms, *Calculateur Paralleles, Reseaux et Systems Repartis* **10**, 141–171.

[5] Corcoran, A.L., Wainwright, R.L., A parallel island model genetic algorithm for the multiprocessor scheduling problem, in: Proc. *ACM/SIGAPP Symposium on Applied Computing*, 1994, 483-487.

[6] Dongarra, J., Geist, A., Kohl, J.A., Papadopoulos, P.M., Sunderam, V., HARNESS: Heterogeneous Adaptable Reconfigurable Networked Systems, *Oak Ridge National Laboratory*, `http://www.csm.ornl.gov/harness/`.

[7] Garey, M., Johnson, D., *Computers and Intractability.* Freeman, New York, 1979.

[8] Huang, M.D., Romeo, F., Sangiovanni-Vincentelli, A., An efficient general cooling schedule for simulated annealing, in: Proc. *IEEE INT. Conf. on Computer Aided Design*, 1986, 381-384.

[9] Jiang, T., Jiang, Z., Breslauer, D., Rotation of periodic strings and shorst superstrings, in: Proc. *Third South American Conference on String Processing*, 1996.

[10] Kirkpatrick, S., Gelatt, C., Vecchi, M., Optimization by simulated annealing, *Science*, **220** (1983), 671-680.

[11] Lesk A.,(editor), *Computational Molecular Biology, Sources and Methods for Sequence Analysis.* Oxford University Press, 1988.

[12] Li, M., Towards a DNA sequencing theory (learning a string), in: Proc. *31st Annual Symposium on Foundations of Computer Science*, 1990, 125–134.

[13] Message Passing Interface, `http://www-unix.mcs.anl.gov/mpi/`, 29 June 2004

[14] Peltola, H., Soderlund, H., Tarhio, J., Ukkonen, E., Algorithms for some string matching problems arising in molecular genetics, in: Proc.*IFIP Congress*,(1983),53–64.

[15] Parallel Virtual Machine, `http://www.csm.ornl.gov/pvm/`, 29 June 2004

[16] Storer, J., *Data Compression: Methods and Theory.* Computer Science Press, 1988.

[17] Sweedyk, Z., A $2\frac{1}{2}$-approximation algorithm for shortest superstring, *SIAM Journal of Computing* **29** (1999), 954-986.

[18] Whitely, D., Cellular genetic algorithms, in: Proc. *Fifth International Conference on Genetic Algorithms*, 1993.

[19] Zaritsky, A., Sipper, M., Coevolving solutions to the shortest common superstring problem, *Biosystems* **76** (2004), 209-216.