Distributed Simulated Annealing with MapReduce

Atanas Radenski
Chapman University, Orange 92865, USA
Radenski@Chapman.edu

Abstract. Simulated annealing's high computational intensity has stimulated researchers to experiment with various parallel and distributed simulated annealing algorithms for shared memory, message-passing, and hybrid-parallel platforms. MapReduce is an emerging distributed computing framework for large-scale data processing on clusters of commodity servers; to our knowledge, MapReduce has not been used for simulated annealing yet. In this paper, we investigate the applicability of MapReduce to distributed simulated annealing in general, and to the TSP in particular. We (i) design six algorithmic patterns of distributed simulated annealing with MapReduce, (ii) instantiate the patterns into MR implementations to solve a sample TSP problem, and (iii) evaluate the solution quality and the speedup of the implementations on a cloud computing platform, Amazon's Elastic MapReduce. Some of our patterns integrate simulated annealing with genetic algorithms. The paper can be beneficial for those interested in the potential of MapReduce in computationally intensive nature-inspired methods in general and simulated annealing in particular.

Keywords: simulated annealing, MapReduce, traveling salesperson (TSP)

1 Introduction

Simulated annealing is a metaheuristic that is used to find near-optimal solutions for various hard combinatorial optimization problems; it does so by imitating the physical process by which melted metal is cooling slowly to form a frozen structure with minimal energy. Simulated annealing is computationally intensive and this has stimulated the exploration of a variety of high-performance simulated annealing algorithms based on popular paradigms: shared memory [9], message-passing [6], and hybrid-parallel [2, 4].

MapReduce (MR) is an increasingly popular distributed computing framework for large-scale data processing that is amenable to a variety of data intensive tasks. Users specify serial-only computation in terms of a *map* method and a *reduce* method, and the underlying implementation automatically parallelizes the computation, tends to machine failures, and schedules efficient inter-machine communication [3]. MR was first implemented as a proprietary platform by Google. Soon afterwards, Apache offered Hadoop MR [15] as open source, and cloud computing providers offer MR platforms on a cost-effective pay-per-use basis.

By design, MR supports fault-tolerance, load-balancing, and scalability. This is in contrast to well understood but lower level high-performance frameworks, such as MPI and OpenMP, in which users - rather than the frameworks - need to tend to

machine failures and scheduling. Such advantages of MR to more traditional frameworks have motivated us to explore its suitability for high-performance simulated annealing and to our knowledge, this is the first study of its kind. MR is known to work well on large datasets. MR's applicability to computationally intensive problem domains with smaller datasets - such as simulated annealing and TSP - poses challenges, primarily because of the lack of direct control over tasks and data allocation. This paper makes contributions towards better understanding of MR's potential in computationally intensive problem domains with smaller datasets in general, and simulated annealing and TSP in particular.

The remainder of the paper is organized as follows. Section 2 introduces MR and then specifies algorithmic patterns for simulated annealing with MR. Section 3 describes a conversion of the patterns into MR implementations for the TSP; it also evaluates the solution quality and performance (execution time and speedup) in the Amazon cloud. Section 4 reviews related work and Section 5 offers conclusions.

2 Placing Simulated Annealing on MapReduce

The MapReduce Framework. Excellent general introductions of the MR framework [3, 8] and its implementation within the Hadoop platform [15] are available to the interested reader. In this paper, we offer only a brief description of MR features needed for the understanding of our simulated annealing algorithm design.

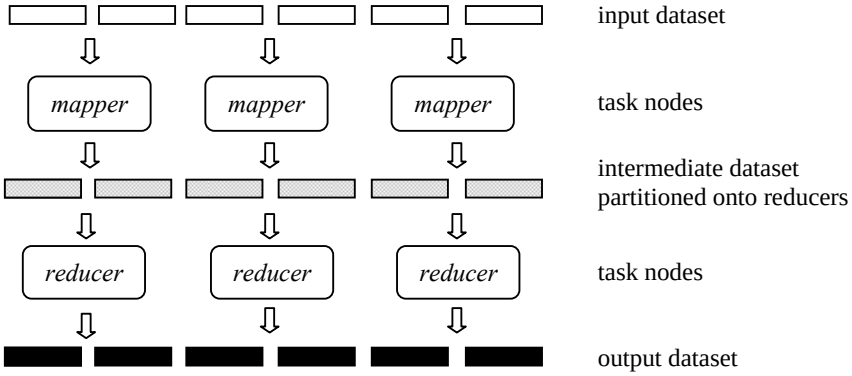

Fig. 1. A simplified representation of an MR job. The size of the output dataset can be different from the size of the input data set. An MR cluster consists of a number of *tasks nodes* and a single *main node* that controls tasks nodes. All mappers and reducers run on task nodes.

The MR framework consists of a programming model and runtime behavior. In the programming model, users specify serial map and reduce methods (one of each kind) that transform key-value records into new key-value records. The run-time environment transforms an input set of records into an output set in two principal stages. First, a user-defined *map* method is applied over all records from the input dataset - in parallel, in a number of separate map tasks, or simply *mappers* – to

produce intermediate outputs from all *map* methods. All intermediate records are then shuffled, sorted, and submitted for final processing by a user-defined *reduce* method. In general, the reduce method can be executed in parallel in several reducer tasks, or simply *reducers*, to produce several output sets of records. MR uses intermediate records' keys to *partition* records between reducers. In that, all intermediate records with the same key are always assigned to the same reducer; yet the same reducer may possibly handle a number of different keys. The MR framework assigns records to mappers and reducers, guided by record keys and without direct user participation.

The map and reduce stages form a single MR *job* (Fig. 1). It is possible to *pipeline* several MR jobs so that the output from one job is used as the input for the next one (Fig. 2). Input and output data sets for MR jobs are stored in a distributed file system.

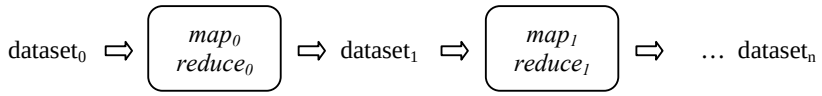

Fig. 2. An MR job pipeline. The output of job *k* becomes the input of job *k+1*. In some MR implementations, such pipelines are referred to as job flows.

We use the following notation for MR pipelines in this paper:

- *A1 + A2 +… + Am* is the pipeline of jobs *A1, A2,… Am*
- *mA* is an abbreviation for a pipeline *A + A +… + A* of length *m*

In addition to the primary *map* and *reduce* methods, the MR framework includes two methods that can be optionally used to *initialize* and *finalize* mappers and/or reducers. Initialization can create objects that persist during *map (reduce)* invocations within the same mapper (reducer); these objects are also available in finalization.

Pure and Hybrid Annealing Patterns. The rest of this section introduces two MR algorithmic patterns for pure simulated annealing and four algorithmic patterns for hybrid simulated annealing. *Hybrid* simulated annealing patterns use genetic operations, such as crossover, to enhance the annealing process, as opposed to *pure* patterns which employ simulated annealing alone. For readability, we present all patterns in Python-like pseudo code, instead of our actual Java implementations.

Data Representation and MR Tasks. Recall that logically, MR input and output datasets are collections of records. In the general case, an MR record is a key-value pair: *record = <key, value>*. *Empty* keys can be used to make records equivalent to values; this option is employed in our simulated annealing algorithmic patterns. Each value represents, in textual form, a possible *solution* to a problem (such as a TSP route, for example). An input/output dataset defines a *population* of candidate-solutions. Our simulated annealing patterns transform *input populations* of candidate-solutions into *output populations* of possibly better candidate solutions.

At the file system level, MR datasets are collections of one or more files. The MR framework uses the number and size of files in the input dataset to determine the number of spawned mappers, without direct user control. In general, each file from a multi-file dataset will be assigned to at least one mapper, with larger files being split by MR and assigned to multiple mappers for the same large file. In particular, a relatively small single-file dataset (such as an input population of candidate solutions

for the TSP for example) will be assigned to a single mapper, regardless of the number of available task nodes in the MR cluster; other nodes will remain idle. In contrast to mappers, the number of reducers can be explicitly defined by the user. In particular, it is possible to define a MR job with *zero reducers*, in which case the output dataset is produced by the mappers alone. Without user specification, a default number of reducers are spawned uniformly on each task node.

Single-Job Simulated Annealing Patterns: SA0 & SA. In practice, it is often difficult to assess the accuracy obtained with a single simulated annealing run. In order to find a better solution, a frequently used strategy is to run simulated annealing a number of times and select the best solution from the independent runs [16, 6]. We have adopted this idea in two single-job MR patterns: a *special* simulated annealing pattern, *SA0*, and a *general-purpose* simulated annealing pattern, *SA*.

With the *special pattern*, *SA0*, annealing runs are performed by distributed mappers in a single MR job with zero reducers, in which mappers simply invoke an annealing algorithm over their assigned candidate-solutions (Fig. 3). The *SA0* pattern is termed *special* because it works well only in the special case of single-record input files. Recall that the number of mappers is implicitly determined by MR as a function of the number and sizes of input files. For good *SA0* performance, each candidate-solution must be preloaded in its own file; in this case, each candidate-solution will be assigned by MR to a dedicated mapper. Grouping all candidate-solution in a single file would be detrimental to *SA0*'s performance because all solutions will most likely be assigned by MR to a single mapper and in fact annealed serially.

```
class Mapper:
  method map(key, value):
    solution = parse(value)
    annealer.anneal(solution)
    emit(empty, solution)
```

Fig. 3. Mapper for the *SA0* algorithmic pattern. MR splits the input set of candidate-solutions between mappers and feeds candidate-solutions as "values" into map method invocations. Each map invocation anneals its assigned candidate-solutions then emits the annealed solution with an empty key in the output population. The number of mappers – and the level of parallelism – is determined by MR based on the number and the sizes of input files with candidate-solutions.

With the *general-purpose pattern*, *SA*, annealing runs are simultaneously performed on distributed reducers rather than on mappers (Fig. 4). The mappers themselves are only used to replace default input keys with new uniformly distributed random keys; even a single mapper can be efficient enough in this process. The updated records are then further submitted by MR to available reducers. Note that the initial default MR key for each record is simply the position in the record in its corresponding file. Such default keys may cause MR to partition records onto a small number of reducers and therefore result in non-uniform reducer workloads. By using uniformly distributed random keys, *SA* provides uniform distribution of records onto mappers. This approach provides good performance regardless of the physical representation of the input set of candidate-solution - either as a single file or as a collection of multiple files.

```
class Mapper:
  method map(key, value):
    randomKey = random();
    emit(randomKey, value)

class Reducer:
  method reduce(key, values):
    for value in values:
      solution = parse(value)
      annealer.anneal(solution)
      emit(empty, solution)
```

Fig. 4. Mapper and reducer for the *SA* algorithmic pattern. MR splits mappers' output onto reducers based on keys emitted by mappers; in this pattern mappers emit random keys to provide uniform distribution onto available reducers and more balanced reducer load. Each reduce invocation anneals all candidate-solutions with the same random key and emits possibly improved solutions in the output population.

Genetic Annealing Pattern: GA+SA. It has been recognized that enhanced initial candidate-solutions for simulated annealing can improve both the quality of the final solution and also the annealing execution time [12]. Such enhanced solutions can be obtained by first applying a genetic algorithm, *GA* (Fig. 5) on a randomly generated initial population of candidate-solutions and then applying simulated annealing, *SA* (Fig. 4) on the genetically evolved population.

```
class Mapper [or Reducer]:
  method initialize: subpopulation = ∅
  method map [or reduce] (key, value):
    subpopulation.add(parse(value))

method finalize():
  genetic.evolve(subpopulation)
  for solution in subpopulation:
    emit(empty, solution)
```

Fig. 5. Mapper and reducer for *GA*, the genetic algorithmic pattern incorporated in the *GA+SA* algorithmic pattern. The *GA*'s mapper and reducer are nearly the same. The initial population of candidate-solutions is split by MR in sub-populations among distributed mappers. Each mapper runs a genetic algorithm on its own subpopulation. All evolved subpopulations are then merged by MR onto a single reducer (in contrast to mappers, the number of reducers can be explicitly controlled programmatically). The reducer then runs the same genetic algorithm to further evolve the entire population. The map/reduce method invocations simply accumulate sub-populations, while the actual genetic computation occurs during finalization.

Thus, *GA+ SA* is a two-job MR pipeline (Fig. 2) in which the first job, *GA* is a basic multi-population genetic algorithm [1]. The second job in *GA+SA* is the general purpose simulated annealing, *SA* (Fig. 4). Note that *SA0*, the special purpose simulated annealing (Fig. 3) must not be used after *GA* because *GA* uses a single reducer and produces a single-file output dataset; recall that *SA0* degrades to serial execution for a single-file input dataset.

Genetic Annealing Pipeline Pattern: m(GA+SA). After a genetic algorithm is trapped in a local minimum, the application of simulated annealing can generate uphill jumps to higher costs solutions thus avoiding premature convergence to a local minimum [5]. This computation can be defined as an MR job pipeline *m(GA+SA)* of *2m* jobs, which consecutive applies *GA+SA* over the dataset produced by the previous application. Again, all intermediate datasets are available, together with the final dataset for the selection of the best solution.

Annealing Genetic Pattern: SA+GA. A genetic algorithm can be used to recombine and possibly improve solutions produced by individual simulated annealing processes [11]. With MR, such computation can be defined in the MR framework as a two-job pipeline in which the first job, *SA* is simulated annealing (Fig. 4) and the second job, *GA* is multi-population genetic evolution (Fig. 5).

Annealing Genetic Pipeline Pattern: m(SA+GA). Genetic recombination can enhance the annealing process by running "simulated annealing followed by genetic recombination" a number of times to gradually obtain a better solution [16]. Such iterative computation can be defined as an MR job pipeline similar to the previously discussed *m(GA+SA)* pipeline, but with *SA* executing before *GA*.

Pipelines in MR. Job pipelines cannot be expressed in the pure MR model proper; such pipelines are often implemented as applications that schedule and run sequences of individual MR jobs. Section 3 contains details on our implementation of pipelines.

3 Implementation and Experimental Evaluation

Annealing the TSP on Amazon's Elastic MapReduce Cloud. The simulated annealing pure and hybrid algorithmic patterns defined in Section 2 can be instantiated to solve specific problems. To instantiate the algorithmic patterns, it suffices to develop *serial-only* domain-specific annealing and genetic algorithms, with no direct involvement of the MR API.

We illustrate this instantiation process with the traveling salesperson problem. We chose TSP because (i) it is known to be computationally intensive for larger problem sizes and because (ii) it is arguably the most popular combinatorial optimization problem that is well studied and well-applied to various specific tasks.

We developed TSP annealing implementations for the Amazon's Elastic MR cloud - a member of Amazon Web Services (AWS). We chose AWS because (i) it is a large and versatile cloud computing platform and because (ii) Amazon supports research through special grants, within its cost-effective pay-per-use business model.

The principal Elastic MR API is for Java. The goal of our proof-of-concept implementation was more to illustrate and evaluate our generic MR algorithmic patterns (Section 2) rather than develop new TSP algorithms. This is why we adopted some features of known serial TSP algorithms to fit the MR Java API.

The TSP aims to find the shortest way to visit each of n points once and return to the initial point. A candidate-solution is an array of different points, referred to as a *tour*. The length of a tour is the sum of the Euclidean distances between its points. In the special case of a *square city grids* of $n = s^2$ points, where s is even, an optimal tour of length n is known to exist [6]. This special case of the TSP problem offers an opportunity to directly assess the solution quality of annealing algorithms.

To instantiate MR algorithmic patterns into TSP implementations, we adapted in Java a proven serial annealing method, originally described by Hansen in SuperPascal [6]. We developed an *Annealer* class with an *anneal* method (Fig. 6) which we plugged into our *SA0* and *SA map* and *reduce* methods (Fig. 3 and Fig. 4).

```
class Annealer:
  method anneal(tour):
    temp = tempMax
    for k = 1, 2, ..., reductions:
      search(tour, temp)
      temp = alpha * temp
```

```
method search(tour, temp):
  na = 0; nc = 0
  while (na < attempts && nc < changes):
    tour1 = swap2RandomPoints(tour)
    if tour1.length() – tour.length < temp:
      tour = tour1
```

Fig. 6. Simulated annealing for the TSP problem. Annealing is implemented by swapping two randomly chosen tour points, *p* and *q* and reversing the tour path between *p* and *q*. The *search* method uses a simple deterministic tour acceptance criterion that has been proven to work just as well as the standard stochastic criterion [10].

For our hybrid annealing TSP implementations, we developed in Java a genetic algorithm with a proven serial crossover method, originally described by Sengoku and Yoshihara [13]. We developed a *Genetic* class with an *evolve* method (Fig. 7) which we plugged into the *GA*'s genetic *map/reduce* methods (Fig. 5).

```
class Genetic:
  method evolve(population):
    for k = 1, 2, ..., generations:
      select(population, m)
      mutate(population, mutatProb)
      cross(population, m)
```

```
method cross(population, m):
  parents = roulette(population, m)
  for  i = 1, 3, 5, ..., m:
    tour1 = crossover(population, i, i+1)
    population.add(tour1)
```

Fig. 7. Basic genetic algorithm for the TSP problem. Proof-of-concept evolutionary computation involves deleting *m* tours from the current population, applying mutation stochastically on the remaining tours, selecting *2*m* parents to crossover, producing a single offspring form every pair of parents, and adding the offspring to the population. The *crossover* method is based on the longest sub-tour crossover operator [13]. Our *mutate* method swaps two random tour points like in simulated annealing.

Pipeline Implementation. AWS's Elastic MR cloud permits the direct implementation of MR job pipelines (Fig. 2) in the form of the so called job flows. At present, to define a job flow in Elastic MR the user must employ Amazon's proprietary lower-level API. We preferred to follow a platform-independent approach, for which we implemented MR job pipelines in Java proper, using reflection: we developed a Java MR utility that reads all classes to be pipelined as command-line arguments then uses a loop to configure and run MR jobs accordingly. In addition, another Java MR utility of ours extracts and sorts all intermediate and final solutions produced by a pipeline and identifies the best solution.

Experimental Evaluation of Solution Quality. We tested experimentally the solution quality of our TSP implementations by means a serial model program; submission and evaluation of Elastic MR jobs is time-consuming and the use of a serial model program helped simplify the evaluation. (We did, however, measure execution times/performance by actually running programs on the Amazon's Elastic MR cloud, as discussed later in this section.) For simulated annealing, we used the same control parameters as in [6]. For genetic computations, we performed

*25*sqrt(tour-size)* generations with crossover probability of 50% and mutation probability of 20%. The population size for these experiments was 16.

Table 1 Pure simulated annealing *SA/SA0* solution quality

Tour Size	100	400	900	1600
Solution	100	405.30	918.72	1648.48
Error	**0%**	**1.33%**	**2.08%**	**3.03%**
Min error	0%	0.82%	1.84%	2.80%
Max error	0%	1.66%	2.30%	3.31

We tested TSP solution quality with pure simulated annealing, *SA* over *square city grids* of *n* points with known optimal tour length of *n* [6]. Table 1 shows averages of all best solutions obtained in 10 trials over tours of various sizes. The solution quality of *SA0* is the same as the solution quality of *SA* because the two methods differ only in the way they distribute the same annealing process between mappers and reducers. The highest average solution error is about 3% for larger tours.

Table 2 Solution quality of hybrid simulated annealing

Algorithm	SA/SA0	GA+SA	4(GA+SA)	SA+GA	4(SA+GA)
Improvement	Base	**0.28%**	**0.61%**	**0.06%**	**0.94%**
Min improv.	Base	0%	0%	0.01%	0.05%
Max improv.	Base	1.30%	2.01%	0.15%	1.89%

We also tested TSP solution quality with hybrid simulated annealing over *random grids* for which no optimal tours are known a priori. Table 2 shows average solution improvements by each of the hybrid methods, *GA+SA*, *4(GA+SA)*, *SA+GA*, and *4(SA+GA)*, relatively to the best solution obtained by pure annealing methods, *SA/SA0*. Solution improvements were measured in 10 trials over various randomly generated tours of 900 cities and populations of size 16. The hybrid *4(SA+GA)* pipeline (Table 2) provides nearly 1% of improvement compared to pure *SA/SA0* and therefore can reduce almost in half the estimated 2% error of *SA* (Table 1).

Experimental Evaluation of Performance on Elastic MR. On the Elastic MR cloud, we tested experimentally TSP solution performance with pure and hybrid simulated annealing: *SA*, *SA+GA*, *4(SA+GA)*, and *SA0*. We did not test performance of *GA+SA* and *4(GA+SA)* because they are comparable performance-wise to *SA+GA* and *4(SA+GA)* correspondingly. Table 3 shows average execution times *T(p)* and speedups *S(p)* on *p* task nodes as obtained in 3 trials over a randomly generated tour of size 900 and populations of sizes 16 and 32. (In Table 3 population sizes are appended in brackets to algorithm designators.) As nodes, we used AWS 32-bit small instances with 1.7 GB memory, 1 virtual core, and moderate I/O performance.

Table 3 shows that special simulated annealing, *SA0* achieves better speedup than general purpose simulated annealing, *SA*. However, *SA0* achieves this speedup for special single-record input files only, as explained in Section 2. It is an advantage of general purpose *SA* that it can be combined with *GA* to form hybrid pipelines that achieve better quality solutions, while *SA0* cannot be combined with *GA*.

Table 3 Elastic MR execution times (in minutes) and speedup

Algorithm	SA[16]		SA[32]		SA0[16]		SA+GA[16]		4(SA+GA)[16]	
Nodes (n)	T(n)	S(n)	T(n)	S(n)	T(n)	S(n)	T(n)	S(n)	T(n)	S(n)
1	11.7	**1.0**	22.6	**1.0**	11.6	**1.0**	13.8	**1.0**	52.0	**1.0**
8	3.0	**3.9**	5.1	**4.4**	2.3	**5.0**	5.3	**2.6**	20.1	**2.6**
16	2.5	**4.7**	3.7	**6.1**	1.5	**7.7**	4.9	**2.8**	16.9	**3.1**

Despite of the use of random keys in *SA*'s mappers, some reducers are assigned by MR more work than others; such imbalances result in relatively moderate speedups when the population size is equal to the number of task nodes (16 nodes in Table 3). Load imbalances can be reduced by using populations of size $k*nodes$ with $k \geq 2$. In general, the scalability of standalone and pipelined *SA* is limited by the population size.

4 Related Work

The serial components of our implementations are based on work from others [6, 13], as already discussed in the preceding section. To our knowledge, we are the first to parallelize simulated annealing with MR, but there are numerous non-MR parallel simulated annealing algorithms, such as, for example, message passing [6], shared memory [9], message passing combined with shared memory [4], and GPGPU-based [2]. Others have proposed self-contained MR-based genetic algorithms [14, 7] and MR has been used for fitness function calculation in evolutionary algorithms [17]; in contrast, our MR genetic algorithm is not intended as standalone but to be incorporated as a job in hybrid annealing pipelines.

5 Conclusions

In this paper, we investigate the applicability of MapReduce to distributed simulated annealing in general, and to the TSP in particular. The specific technical contributions of this paper are as follows: (i) we propose six MR algorithmic patterns for distributed simulated annealing; (ii) we instantiate the MR patterns into TSP implementations; (iii) we evaluate the MR implementations in cloud computing environment.

A significant advantage of our MR simulated annealing patterns to traditional parallel algorithms is that these patterns provide *fault-tolerant MR parallelism* without user intervention. With the use of MR, we trade some speedup for fault-tolerance and robustness. The lack of direct user control on parallelism however can also be a limitation when the programmer wants to explicitly declare some MR parameters, such as the total number of mappers. A benefit from our annealing MR patterns is that they can be instantiated into MR applications with the addition of serial-only domain code, such as code to represent, anneal, and evolve the TSP for example. Our hybrid annealing patters are slower than the pure annealing patterns but are more precise. In future work, the genetic component of hybrid patterns can be

fine-tuned to make then even more precise. The Amazon's Elastic MR cloud offers the advantages of instant cluster provisioning and pay-per-use cost efficiency for users who do not have access to dedicated MR clusters on the premises.

Acknowledgement. This work was supported by an AWS in Education 2011 research grant award from Amazon.

References

[1] Cantú-Paz, E.: Efficient and Accurate Parallel Genetic Algorithms. Kluwer, Boston (2000)
[2] Choong, A., Beidas, R., Zhu, J.: Parallelizing Simulated Annealing-Based Placement using GPGPU. In: Field Programmable Logic and Applications, pp. 31--34, IEEE. New York (2010)
[3] Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. CACM 51, No. 1, 107--113 (2008)
[4] Debudaj-Grabysz, A., Rabenseifner, R.: Nesting OpenMP in MPI to Implement a Hybrid Communication Method of Parallel Simulated Annealing on a Cluster of SMP Nodes. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface 2005. LNCS, vol. 3666, pp. 18--27. Springer, Heidelberg (2005)
[5] Elhaddad, Y., Sallabi, O.: A New Hybrid Genetic and Simulated Annealing Algorithm to Solve the Traveling Salesman Problem. In: World Congress on Engineering (WCE 2010), vol. 1, pp 11--14. International Association of Engineers, Taipei (2010)
[6] Hansen, P.-B.: Studies in Computational Science. Prentice Hall, Englewood Cliffs (1995).
[7] Huang D.-W, Lin, J.: Scaling Populations of a Genetic Algorithm for Job Shop Scheduling Problems Using MapReduce. In: 2010 IEEE 2nd International Conference on Cloud Computing Technology and Science, pp. 78--785. IEEE, New York (2010)
[8] Lin, J., Dyer, C.: Data-Intensive Text Processing with MapReduce. Morgan and Claypool, San Francisco Bay Area (2010)
[9] Ma, J., Li, K., Zhang, L.: The Adaptive Parallel Simulated Annealing Algorithm Based on TBB. In: 2nd International Conference on Advanced Computer Control, pp. 611—615. IEEE, New York (2010)
[10] Moscato, P., Fontanari, J.: Stochastic versus Deterministic Update in Simulated Annealing. Physics Letters A, 146, No. 4, pp. 204--208. Elsevier, Amsterdam (1990)
[11] Ohlídal M., Schwarz J.: Hybrid Parallel Simulated Annealing Using Genetic Operations. In: Mendel 2004, 10th International Conference on Soft Computing, pp. 89--94. University of Technology, Brno (2004)
[12] Ram, J. D., Sreenevas, T.T., Subramaniam K.G.: Parallel Simulated Annealing Algorithms. J. Par. Distr. Computing 37, pp. 207--212 (1996)
[13] Sengoku, H., Yoshihara, I.: A Fast TSP Solver Using GA on Java. In: 3rd Int. Symp. Artif. Life and Robot., pp.283—288. Springer Japan, Tokio (1998)
[14] Verma, A., Llorà, X., Goldberg, D. E., Campbell, R. H.: Scaling Genetic Algorithms Using MapReduce. In: 9th International Conference on Intelligent Systems Design and Applications, pp. 13--18. IEEE, New York (2009)
[15] White, T.: Hadoop: The Definitive Guide (2nd ed.). O'Reilly Media, Sebastopol (2009)
[16] Yao, X.: Optimization by Genetic Annealing. In: 2nd Australian Conf. Neural Networks, pp. 94--97. Sidney University, Sidney (1991)
[17] Zhou C.: Fast Parallelization of Differential Evolution Algorithm Using MapReduce. In: 12th Annual Conference on Genetic and Evolutionary Computation, pp. 1113--1114. ACM, New York (2010)