# Savant: Automatic Parallelization of a Scheduling Heuristic with Machine Learning

Frédéric Pinel, Pascal Bouvry
FSTC/CSC/ILIAS
University of Luxembourg
Email: frederic.pinel@uni.lu
pascal.bouvry@uni.lu

Bernabé Dorronsoro
LIFL
University of Lille
France
Email: bernabe.dorronsoro_diaz@inria.fr

Samee U. Khan
Department of Electrical and Computer Engineering
North Dakota State University
Fargo, USA
Email: samee.khan@ndsu.edu

***Abstract*—This paper investigates the automatic parallelization of a heuristic for an NP-complete problem, with machine learning. The objective is to automatically design a new concurrent algorithm that finds solutions of comparable quality to the original heuristic. Our approach, called Savant, is inspired from the Savant syndrome. Its concurrency model is based on map-reduce. The approach is evaluated with the well-known Min-Min heuristic. Simulation results on two problem sizes are promising, the produced algorithm is able to find solutions of comparable quality.**

***Keywords*-Pattern matching, Parallelism and concurrency, Conversion from sequential to parallel forms**

## I. Introduction

Parallel algorithms are becoming necessary in every aspect of computing. Previously, parallelism was only needed in specific cases, typically for performance. The default computer is now a parallel machine [1]. This trend is a consequence of the evolution of computer processors, where physical limits are forcing chip designers to reduce the clock frequency of processors, and packaging more of them. Current computers now come with multiple processors, which are themselves multi-core. In addition, alternative parallel co-processors are common, such as graphics processing units (GPU). Mass markets are also favoring distributed systems such as clusters, assembled from off-the-shelf components in contrast to specialized parallel hardware [2]. Finally, recent Internet trends, such as cloud computing, the ubiquity of JavaScript virtual machines and mobile devices add another level in parallelism, by massively distributing computation across cloud servers and browsers. However, the parallelism offered by hardware requires concurrent algorithms to be fully exploited. Recent algorithms seek concurrency as much as possible, but programming languages for concurrent program are only appearing [3], [4], [5], [6], and manually designing a concurrent program remains a difficult task. Moreover a great number of existing programs need to be adapted to the parallel architectures.

In light of this trend, we are investigating a method to automatically parallelize existing algorithms. This is of course a challenging problem. As a first step, we limit the problem's scope in several ways.

- We relax a common constraint that the parallel version must be similar to the original algorithm (equivalent instructions, in a different order). We are looking to produce a different algorithm that nevertheless provides the same function.
- We consider the problem a supervised learning problem, and leverage the efficiency of modern machine learning techniques. This is inspired by the Savant syndrome (Section III-B), which hints to a parallel machine performing seemingly sequential tasks. By analogy, we consider that the parallel version of the original algorithm must learn the behavior of the original one.
- We evaluate our proposed approach on a specific algorithm, a well-known heuristic for an NP-complete scheduling problem. This is motivated by three factors. First, we are familiar with this problem and its state-of-the-art parallel solvers. This is advantageous to assess the results obtained. Second, optimization problems highlight the key point of our approach: the design of different algorithms that solve the same problem. Indeed, solutions to optimization problems are evaluated with a fitness function, regardless of how this solution was found. This helps us abandon existing algorithms, as long as the solutions are comparable, and the production method is parallel. Finally, solving combinatorial problems is compute intensive, and parallel heuristics is an active research area that could benefit from automatic parallelization. The ultimate intention is to evaluate this approach on other algorithms and problems.

Our contribution is a method to automatically generate a parallel version of the chosen heuristic.

Section II describes the problem and reviews previous work. Section III presents and motivates our approach, called Savant. Section IV shows results of the Savant algorithm.

## II. Problem Statement

In this section, we present the automatic parallelization problem. Section II-A reviews past and current automatic parallelism. Section II-B presents the scheduling problem that we evaluate our approach on.

### A. Automatic Parallelization

Parallelism was initially considered a part of the automatic build of executables from source code. This optimization step is usually approached by applying source-to-source transformations [7], [8]. Transformations include loop-unrolling, data access patterns, and rely on careful inspection of data dependencies to extract concurrency from the source program. This approach to parallelization preserves the algorithm and most of the source code, by applying transformations that respect the semantics of the original program. The transformations are carefully defined, so as to guarantee identical behavior, and may even rely on formal reasoning [9]. Other authors apply AI techniques to identify the transformations and their order of application. Evolutionary algorithms and machine learning were applied in [10], [11], [12].

As mentioned, our focus is not to preserve most of the source program, nor even the algorithm, but to find new algorithms and code. Genetic Programming (GP) [13] is a method to achieve such a goal. Indeed, GP aims to automatically evolve a program that displays a set of properties. Parallelism can be one of them. A combined evolutionary and source-to-source transformation technique was presented in [14]. There is little detail presented however. In [15], [16], [17], the authors use GP to evolve a program in order to achieve parallelism. The programs found are evaluated both in terms of correctness (their purpose) and their degree of parallelism. Evolving the program allows for easier evaluation of the parallelism by executing the code. We find that the programs evolved are relatively simple (O(100) assembly instructions), and require considerable effort to find (the stopping condition is the absence of progress in the last $10^6 - 10^8$ evolutions). GP is a general technique which comes with its drawbacks, such as the computational effort required. Also, defining parallelism as a fitness function is an elegant formulation of the problem, but is not reliable regarding the parallelism obtained. We believe more specific, thus efficient, approaches can be used. Finally, genetic algorithms can be used to evolve rules for computation, instead of a solution to a problem [18]. Therefore, such an approach could in principle be used for automatic parallelization but we have not found previous work.

### B. The Min-Min Heuristic for the Independent Task Mapping Problem

We decide to focus on a certain class of algorithms and programs, those solving combinatorial optimization problems. Specifically, we will be parallelizing an algorithm for a known optimization problem of the scheduling domain: the independent task mapping problem. This problem assumes a set of independent computing tasks, which can be processed by a set of heterogeneous resources, or machines. Each task can only be processed by a machine. The estimated time to complete (ETC) each task, on each machine, is given. This assumption is usually made in the literature [19], [20], [21]. An ETC example is:

$$\begin{bmatrix} & t_1 & t_2 & t_3 & ... \\ m_1 & 12.3 & 17.8 & 45.7 & ... \\ m_2 & 15.9 & 18.3 & 23.0 & ... \end{bmatrix},$$

where task $t_1$ takes 12.3 units of time to execute on machine $m_1$.

The optimization problem is how to assign the tasks to the resources (to map) such that the finishing time of the last task, over all the machines, is minimum. This finishing time is called makespan in the scheduling literature. The makespan minimization problem is NP-complete [22]. A solution to the problem can be represented as an array of integers, where $solution[t] = m$ means that task t is assigned to machine m. The makespan of this solution is time when the last task finishes. The order in which the tasks are executed on a machine is not important. The heuristic chosen for the study is the well-known Min-Min resource allocation algorithm [23], [19]. It is a deterministic heuristic that runs in $O(MT^2)$, for M machines and T tasks. It can be manually parallelized with some effort, and for a specific architecture only (i.e. for the GPU [24]).

## III. Approach

Here we present our approach to automatic parallelization. Our starting point is a generic parallel algorithm that satisfies our concurrency objectives, Section III-A. We present a source of inspiration for our approach in Section III-B, and describe how we specialize the generic parallel algorithm to solve the optimization problem in Section III-C.

### A. Defining a Target Parallel Model

In Section II-A we mentioned that the previous GP approaches considered parallelism as an objective, which yields uncertain results. Here, we pose the problem by specifying a target parallel model. Our approach will only produce algorithms that conform to this model, thus guaranteeing the level of parallelism.

The chosen algorithmic model is a single iteration of a map-reduce application [25]. Open source and free software frameworks exist for this model, on different architectures (GPU, multi-core, clusters), which makes it a practical choice. Moreover, theoretical works have found it equivalent to BSP and PRAM [26], both well-studied parallel models. In this model, the input data is first processed independently by many mappers, their results are then further processed independently by reducers. Independent processing means that the mappers and reducers do not communicate or otherwise synchronize. In addition to this qualitative definition of the parallel model, we need a high number of mappers or reducers, regardless of the problem size. Also, the new algorithms must scale with respect to problem size.

### B. Analogy with the Savant Syndrome

Our automatic parallelization question can be restated as: how to automatically design a scalable and massively parallel map-reduce algorithm, that finds solutions to the independent task mapping problem of comparable quality (fitness) to the Min-Min heuristic.

We looked for previous occurrences where a massively parallel machine (composed of weak computing nodes, to ensure the reliance on parallel processing), was able to solve small, sequential problems in a short time. This question lead to the Savant syndrome [27], [28], [29], [30], [31]. People displaying symptoms of this syndrome can compute small sequential tasks, such as calendar computation (finding the day of the week for a given date), in a very short time (700 msec), using unknown methods. Their methods for calendar computation are unknown because experiments showed that the distribution of the response time does not match those of known computer programs. Also, Savants can perform other date computations with similar performance while this is more time-consuming for a computer algorithms (and reported impossible with classical algorithms [29]). Although not fully understood, the Savants seem to be learning pattern-recognition rules from data, which are later applied in parallel to new input. This matches their ability to perform calendar computation while ignoring complicated details of calendars, and to enumerate prime numbers while ignoring what a prime number is, or how to multiply and divide. The mental activities that some Savants (such as D. Tammet) describe incline us to believe that their learning method is supervised. Finally, Savants seem to manipulate probabilities: their answers are not 100% correct, and although they are not very proficient in mathematics in general, they understand probabilities better than average. The learned pattern matching is also consistent with other studies, such as chess perception in players of different skill [32], [33].

### C. *Application to Automatic Parallelization*

Exploiting the analogy of the Savant syndrome for the resolution of the independent task mapping problem is straightforward. As presented in Section II-B, a solution to the problem is an array of integer numbers. There is much fewer machines than tasks to assign, usually hundreds of tasks, and $\times 32$ less machines.

The mappers in the map-reduce model are multi-class classifiers. Each classifier must correctly assign the task to a machine. Correctness means choosing the same machine as Min-Min, because this is the algorithm we are parallelizing. The mappers' input is the ETC matrix. However each mapper does not need all or the same ETC data (Section IV-A provides more details). The classifiers result from supervised learning, in analogy to the Savant syndrome, and because it is well-suited to the parallelization problem (we are given an original algorithm or program to parallelize, which can generate as much training data as required). This is another advantage of the classification approach over genetic programming, which randomly proposes algorithms in hope of meeting the objectives. We use one mapper per task in the independent task mapping problem. This is high number of mappers for a given problem instance, and scales linearly with the problem size (the number of tasks). Also, the classifiers work independently of each other.

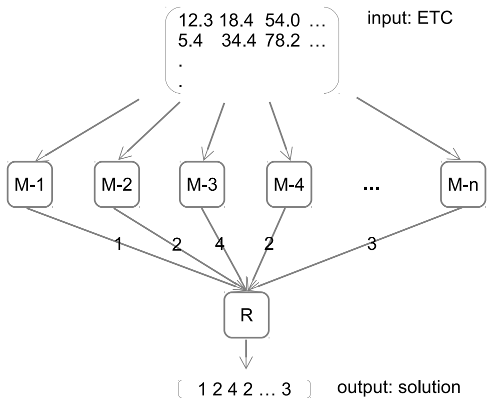

Fig. 1. Overview of the Savant parallel algorithm

The reduce step collects the mappers' output for all tasks and assembles the final solution. A simple reducer can do nothing: just relay the classifiers' solution. However, the independent task mapping problem optimizes a fitness (minimizes makespan), and this fitness is not exploited so far. The reduce step could improve the solution provided by the mappers by exploiting the fitness objective. The reducer we propose is a random local search. It performs a fixed number of random swaps in the solution (swaps machine assignments between two randomly selected tasks) and updates the solution when a swap improves fitness. This reducer runs in constant time, because it only depends on the number of swaps chosen, and is not problem specific (beyond the fitness calculation). The parallel algorithm produced is presented in Figure 1. The input is an independent task mapping problem instance, an ETC matrix of size $machines \times tasks$. The various M boxes are the mappers, one for each task. The R box is the reducer, in this diagram the simple reducer is shown (relays the classifiers' results).

The classifiers are trained under supervised learning, with ETC and their solution obtained by Min-Min. The next section details the parameters for the training and evaluation of the approach.

## IV. Results

This section presents the simulation setup and the results observed.

### A. *Setup*

The ETC were randomly generated according to [19]. The machines and tasks are sorted. The low-indexed tasks have smaller computation time than the higher indexed tasks. The low-indexed machines are faster than the hight indexed machines. This is necessary to train the classifiers with common index values for tasks and machines across ETC instances. Two problem sizes are used in the simulations, 128 tasks x 4 machines, and 512 tasks x 16 machines. The intention is to observe the behavior of the Savant algorithm when problem size increases.

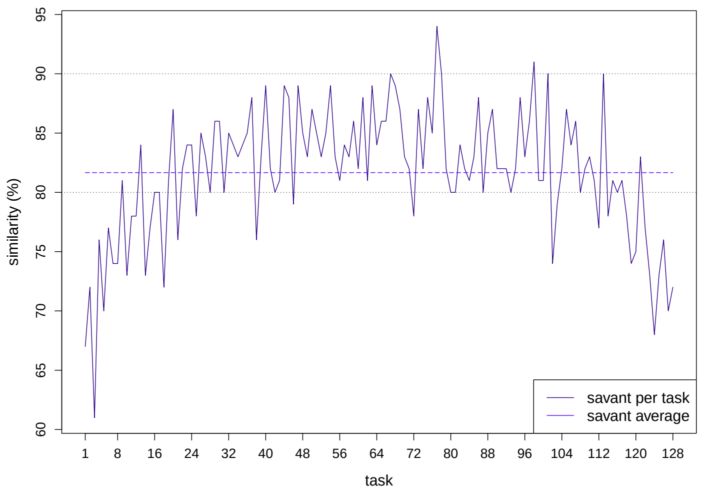

Fig. 2. Savant solution similarity for $128 \times 4$ problems (without the local search reducer)

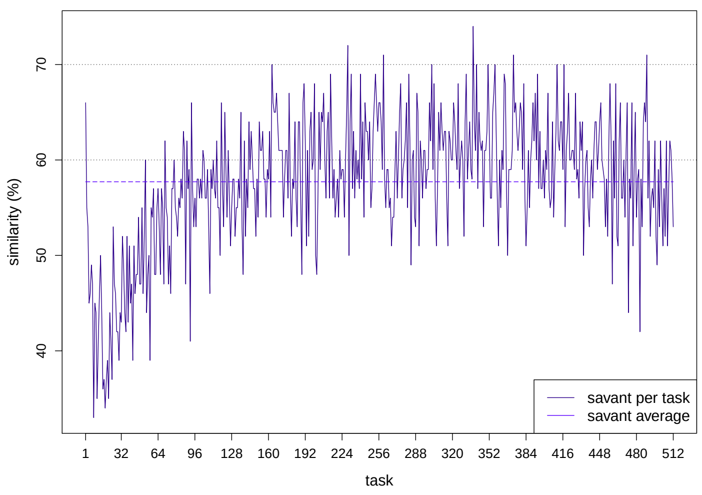

Fig. 3. Savant solution similarity for $512 \times 16$ problems (without the local search reducer)

The classifiers are multi-class SVM, implemented in libSVM [34]. We choose the recommended default parameters. The kernel chosen is RBF. Cross-validation is used to select the parameters. 600 ETC different instances are used in the training phase. 100 ETC different instances are used in the testing phase. The ETC input is scaled.

An important design decision of the Savant algorithm is the selection of features to use for the SVMs, in training and testing. Choosing the entire ETC input data for classification provides poor results, in addition to longer training time. We have chosen, after a short investigation, to use a simple rule: the features used for the classifier of a task are the values of the ETC column of that task. We investigated the use of neighboring columns, in addition to the task's ETC column, but with no significant improvement. In scheduling terms, these values are the estimated completion times of the task on all machines. Therefore to train or test the SVM for each task, the features are first extracted from the ETC.

When used, the local search reducer (Section III-C) is run for 10,000 iterations.

## B. Savant Solution Similarity

In this section, we compare the solutions found by the Savant algorithm to the ones found by Min-Min. The reducer simply assembles the results from the individual task classification, the local search reducer is not used. This comparison reflects the classifiers accuracy in predicting the Min-Min assignments. The accuracy measure is the count of correct classification over the 100 test instances, per task.

Figure 2 shows that the accuracy for the smaller tasks and bigger tasks is low. The accuracy for the smaller tasks is acceptable because they have little influence on the fitness function. The average accuracy is 82%, which is quite good.

Figure 3 shows that the accuracy of the classifiers is worse than for the smaller problem. This is perhaps due to the increased number of machines (classes) to assign tasks to, although we kept the same number of training instances. Also, this may point to a weakness of the common feature selection rule for all tasks.

## C. Savant Solution Quality

In this section, we evaluate the quality of the solutions found by the Savant algorithm. Three comparisons are shown. The leftmost boxplot (labeled ”savant vs minmin”) reports the differences in solution quality (fitness, expressed in %) between the Savant algorithm without the local search reducer and the original Min-Min algorithm, for the same test ETC instances. The middle boxplot (labeled ”savant+ls vs minmin”) shows the same comparison but with the local search reducer presented in Section III-C. The local search in the reducer may be responsible for these results. To measure this possible effect, we add the rightmost boxplot (labeled ”savant+ls vs ls”) to compare the Savant and local search reducer with the same local search applied to random solutions.

Figure 4 shows boxplots results for the smaller instances. We note that the Savant algorithm using only classifiers produces very good results, 10% worse in median than Min-Min. With the local search reducer, Savant results are very good because the algorithm actually finds better solutions than Min-Min. The rightmost boxplot tells us that the classifiers do play a role in the quality of the solutions found.

Figure 5 shows results for the larger instances. We see that the degraded accuracy observed in Figure 3 impacts the quality of the solutions found. However, the Savant with the local search reducer still produces good results. The rightmost boxplot shows that the Savant classifiers contribute significantly to the quality of the solutions found. However, the poor results from the random solutions with local search are due to the same number of iterations used, while the solution is bigger. In both problem sizes, we notice that the local search reducer

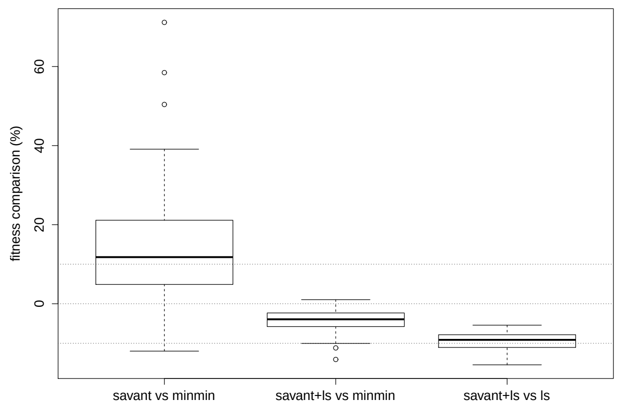

Fig. 4. Savant solution quality for $128 \times 4$ problems

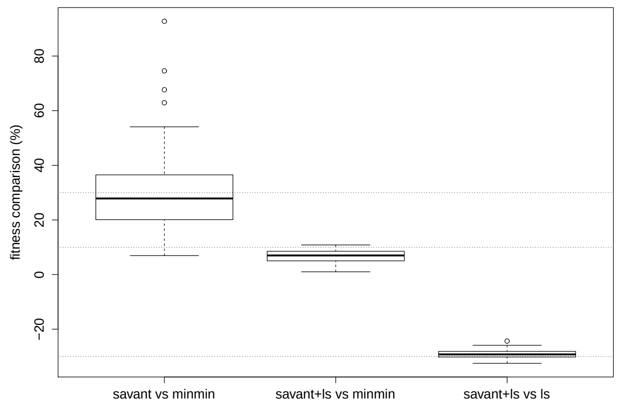

Fig. 5. Savant solution quality for $512 \times 16$ problems

significantly reduce the variance in the results.

## V. Conclusions

This paper investigated the possibility of automatically parallelizing a heuristic for a combinatorial optimization problem. The long term goal is to find a parallelization method applicable to as many algorithms as possible. The approach presented, Savant, contrasts with previous work: we defined a generic parallel pattern-matching engine (suited to map-reduce) that learns the algorithm to parallelize. The parallel algorithm produced is completely different from the original sequential algorithm, yet achieves the same results. This concept is easier to apply with an optimization problem, because solutions are compared on quality, and not similarity.

We consider the results presented promising. The Savant algorithm provides comparable solutions to the original algorithm. However, there is much room for improvement.

The optimization problem instances addressed are not large, and the original algorithm (Min-Min) is fast on such problem sizes. Although our initial goal is to automatically design competitive parallel algorithms, rather than rival on execution speed, because of the on-going trends in computing architectures. However, the degraded accuracy and solution quality when the problem size increases needs to be addressed. We plan to use more training samples when the number of classes increases, as is the case when problem size increases.

A more fundamental improvement is the feature selection for the tasks SVMs. We used a simple rule, common to all task SVMs (that nevertheless achieves good results): select the task's ETC column. We are currently investigating the use of different feature selection rules for each task, given the tasks' differences. One alternative is to use different ETC columns for each task's classifier. Another alternative is to use elements of the ETC matrix, instead of columns. We plan to automatically discover such rules, so as to meet our goal of automatic parallelization.

A minor improvement is the re-design of the reducer step, into a parallel version.

Other future work will investigate if the Savant algorithm is suitable for different, more elaborate and thus time-consuming, algorithms for the same optimization problem. Also, we need to understand how this approach performs on different problems altogether.

## Acknowledgment

This work is supported by the Fonds National de la Recherche Luxembourg: CORE Project Green-IT and AFR contract no 4017742.

We thank the anonymous reviewers for their comments and suggestions for improvement.

## References

[1] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek *et al.*, "A view of the parallel computing landscape," *Communications of the ACM*, vol. 52, no. 10, pp. 56–67, 2009.

[2] A. S. William Gropp, Ewing Lusk, *Using MPI*. MIT Press, 1999.

[3] J. Armstrong, "The development of Erlang," in *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, ser. ICFP '97. New York, NY, USA: ACM, 1997, pp. 196–203. [Online]. Available: http://doi.acm.org/10.1145/258948.258967

[4] D. Callahan, B. L. Chamberlain, and H. P. Zima, "The cascade high productivity language," in *High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings. Ninth International Workshop on*. IEEE, 2004, pp. 52–60.

[5] Google, "The go programming language," http://golang.org/, accessed July 19, 2013.

[6] Mozilla, "The rust programming language," http://www.rust-lang.org/, accessed July 19, 2013.

[7] C. D. Callahan, K. D. Cooper, R. T. Hood, K. Kennedy, and L. Torczon, "ParaScope: A parallel programming environment," *International Journal of High Performance Computing Applications*, vol. 2, no. 4, pp. 84–99, 1988.

[8] F. Irigoin, P. Jouvelot, and R. Triolet, "Semantical interprocedural parallelization: An overview of the PIPS project," in *Proceedings of the 5th international conference on Supercomputing*. ACM, 1991, pp. 244–251.

[9] X. Leroy, "Formal certification of a compiler back-end or: programming a compiler with a proof assistant," in *ACM SIGPLAN Notices*, vol. 41, no. 1. ACM, 2006, pp. 42–54.

[10] D. Zhang and J. J. Tsai, "Machine learning and software engineering," *Software Quality Journal*, vol. 11, no. 2, pp. 87–119, 2003.

[11] M. Harman, "The current state and future of search based software engineering," in *2007 Future of Software Engineering*. IEEE Computer Society, 2007, pp. 342–357.

[12] C. Ryan, A. H. van Roermund, and C. J. M. Verhoeven, *Automatic re-engineering of software using genetic programming*. Kluwer Academic, 2000.

[13] J. R. Koza, "Genetic programming as a means for programming computers by natural selection," *Statistics and Computing*, vol. 4, no. 2, pp. 87–112, 1994.

[14] P. Walsh and C. Ryan, "Paragen: a novel technique for the autoparallelisation of sequential programs using gp," in *Proceedings of the First Annual Conference on Genetic Programming*. MIT Press, 1996, pp. 406–409.

[15] S. M. Cheang, K. S. Leung, and K. H. Lee, "Genetic parallel programming: Design and implementation," *Evolutionary Computation*, vol. 14, no. 2, pp. 129–156, 2006.

[16] K. S. Leung, K. H. Lee, and S. M. Cheang, "Evolving parallel machine programs for a multi-alu processor," in *Evolutionary Computation, 2002. CEC'02. Proceedings of the 2002 Congress on*, vol. 2. IEEE, 2002, pp. 1703–1708.

[17] K. Thearling and T. S. Ray, "Evolving parallel computation," *Complex Systems*, vol. 10, no. 3, p. 229, 1996.

[18] D. E. Goldberg, "Genetic and evolutionary algorithms come of age," *Communications of the ACM*, vol. 37, no. 3, pp. 113–119, 1994.

[19] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund, "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *J. Parallel Distrib. Comput.*, vol. 61, pp. 810–837, June 2001. [Online]. Available: http://portal.acm.org/citation.cfm?id=511973.511979

[20] A. Ghafoor and J. Yang, "A distributed heterogeneous supercomputing management system," *IEEE Computer*, vol. 26, no. 6, pp. 78–86, 1993.

[21] M. Kafil and I. Ahmad, "Optimal task assignment in heterogeneous computing systems," in *Heterogeneous Computing Workshop*. IEEE Computer Society, 1997, pp. 135–146.

[22] E. Horowitz and S. Sahni, "Exact and approximate algorithms for scheduling nonidentical processors," *Journal of the ACM (JACM)*, vol. 23, no. 2, pp. 317–327, 1976.

[23] O. H. Ibarra and C. E. Kim, "Heuristic algorithms for scheduling independent tasks on nonidentical processors," *Journal of the ACM*, vol. 24, no. 2, pp. 280–289, 1977.

[24] F. Pinel, B. Dorronsoro, and P. Bouvry, "Solving very large instances of the scheduling of independent tasks problem on the gpu," *Journal of Parallel and Distributed Computing*, 2012.

[25] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[26] M. F. Pace, "Bsp vs mapreduce," *Procedia Computer Science*, vol. 9, pp. 246–255, 2012.

[27] H. Welling, "Prime number identification in idiots savants: Can they calculate them?" *Journal of autism and developmental disorders*, vol. 24, no. 2, pp. 199–207, 1994.

[28] L. Mottron, M. Dawson, I. Soulieres, B. Hubert, and J. Burack, "Enhanced perceptual functioning in autism: An update, and eight principles of autistic perception," *Journal of autism and developmental disorders*, vol. 36, no. 1, pp. 27–43, 2006.

[29] L. Mottron, K. Lemmens, L. Gagnon, and X. Seron, "Non-algorithmic access to calendar information in a calendar calculator with autism," *Journal of autism and developmental disorders*, vol. 36, no. 2, pp. 239–247, 2006.

[30] J. R. Hughes, "A review of savant syndrome and its possible relationship to epilepsy," 2010.

[31] H. Darius, "Savant syndrome-theories and empirical findings," Ph.D. dissertation, University of Skövde, 2007.

[32] W. G. Chase and H. A. Simon, "Perception in chess," *Cognitive Psychology*, vol. 4, no. 1, pp. 55 – 81, 1973. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0010028573900042

[33] N. Charness, E. M. Reingold, M. Pomplun, and D. M. Stampe, "The perceptual aspect of skilled performance in chess: Evidence from eye movements," *Memory & Cognition*, vol. 29, no. 8, pp. 1146–1152, 2001.

[34] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Trans. Intell. Syst. Technol.*, vol. 2, no. 3, pp. 27:1–27:27, May 2011. [Online]. Available: http://doi.acm.org/10.1145/1961189.1961199