DOI 10.12694/scpe.v18i2.1284
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 18, Number 2, pp. 117–131. http://www.scpe.org

AUTOMATIC TUNING ON MANY-CORE PLATFORM FOR ENERGY EFFICIENCY VIA SUPPORT VECTOR MACHINE ENHANCED DIFFERENTIAL EVOLUTION

ZHILIU YANG, ZACHARY I. RAUEN, AND CHEN LIU*

Abstract. The modern era of computing involves increasing the core count of the processor, which in turn increases the energy usage of the processor. How to identify the most energy-efficient way of running a multiple-program workload on a many-core processor while still maintaining a satisfactory performance level is always a challenge. Automatic tuning on the voltage and frequency level of a many-core processor is an effective method to aid solving this dilemma. The metrics we focus on optimizing are energy usage and energy-delay product (EDP). To this end, we propose SVM-JADE, a machine learning enhanced version of an adaptive differential evolution algorithm (JADE). We monitor the energy and EDP values of different voltage and frequency combinations of the cores, or power islands, as the algorithm evolves through generations. By adding a well-tuned support vector machine (SVM) to JADE, creating SVM-JADE, we are able to achieve energy-aware computing on many-core platform when running multiple-program workloads. Our experimental results show that our algorithm can further improve the energy by 8.3% and further improve EDP by 7.7% than JADE. Besides, in both EDP-based and energy-based fitness SVM-JADE converges faster than JADE.

Key words: Machine learning, many-core processors, energy-aware computing

AMS subject classifications. 68M20, 68T05

1. Introduction.

In the modern computing era, the core count of the processor has continuously being on the rise in pursuit of higher computational throughput. One downside, however, is the increased power consumption and energy cost associated with core count increase. There have been several different methods to minimize the energy usage of the processor, one of the most popular options being dynamic voltage and frequency scaling (DVFS). With DVFS, the voltage and frequency level of the cores can be changed in an effort to reduce the overall energy usage. On the other hand, we can keep the cores running in a very low power state in order to save energy. But this will result in a very poor user experience as the performance in terms of computation speed is unsatisfactory. How to identify the most energy-efficient way of running the workload while still maintaining a satisfactory performance level is always a challenge. When running multiple programs on a many-core platform, one energy-aware approach is to use different voltage and frequency setting with different number of cores related to the inherent property of the program. Since it is not always feasible to search through all possible combinations of frequency, voltage, core count with a brute force method, automatic tuning becomes an effective and promising method to aid solving this dilemma.

Several algorithms exist to accomplish the task of automatic tuning such as Genetic Algorithm (GA) and Differential Evolution Algorithm (DE), which search for solution as the algorithm evolves through generations. In evolutionary computation, all of the individuals in a single iteration are called a generation. Similar to the evolution of living creatures, individuals in one generation can exchange their unique features, which provides the possibility for generating better offspring. Standard differential evolution (DE) can provide a good solution for auto-tuning [1], however, its convergence rate is still relatively slow. Convergence in a performance curve is the point the performance stops to change obviously. Fast convergence rate means the algorithm reaches or gets close to the optimal performance much faster. In the realm of auto-tuning for many-core platforms, there have been other algorithms that attempt to converge faster than standard DE, one of which being the adaptive differential evolution algorithm (JADE).

In [3], Jiang et al. did a comprehensive study between JADE and several other representative algorithms such as Genetic Algorithm (GA), Particle Swarm Optimization (PSO) and the β algorithm. In [3], JADE is proven to converge faster than standard DE algorithm and still produce the best results among the comparable approaches, making it the new benchmark for this type of algorithm.

*Department of Electrical and Computer Engineering, Clarkson University, 8 Clarkson Ave, Potsdam, New York, 13699 (zhilyan@clarkson.edu, rauenzi@clarkson.edu, cliu@clarkson.edu)

Although it converges quicker than the normal DE algorithm, JADE still requires several generations of evolution that requires a large computation overhead. In order to decrease the overhead of JADE, we aim to decrease the number of generations required to achieve convergence. As shown in [2], Machine Learning Enhanced Differential Evolution (mDE) can result in a much quicker convergence. Since JADE is proven to be more effective than DE in energy-aware automatic tuning on many-core processors, and mDE converges quicker than JADE, we propose SVM-JADE, a new auto-tuning algorithm for fast DVFS auto-tuning, which can identify the optimal settings and number of cores per program to minimize energy and energy-delay product (EDP) while running multiple programs on many-core platforms.

The rest of the paper is organized as follows: A brief overview of background works can be found in Sect. 2, followed by the methodology for our specific version of differential evolution in Sect. 3. The implementation of scheme are introduced in Sect. 4 with the experimental results being discussed in Sect. 5. Our conclusions are presented in Sect. 6.

2. Background. In this section we introduce the hardware platform we use to conduct this research, as well as review some representative works on energy-aware auto-tuning.

2.1. Hardware Platform. The Single-chip Cloud Computer (SCC) [4] is an experimental processor created by Intel Labs for many-core based research. The architecture of SCC is shown in Fig. 2.1 [6]. The SCC has 48 Intel Pentium P54C cores and employs mesh communication across the entire core-set. These cores are divided into 6 power domains consisting of 8 cores each. Each of these power domains can be set to different voltage levels independent of one another. The individual cores are grouped into pairs within the power domain yielding 4 pairs per power domain. These core pairs form frequency domains that can have their frequency set individually, much like the voltage of the power domain. One example of these domains can be seen in Fig. 2.1. Cores 32 through 35, and cores 44 through 47 all belong to the same power domain as seen by the box encompassing them. The pairings of the cores seen within the power domain, such as cores 46 and 47, are all individual frequency domains.

Due to these abilities, the SCC becomes an ideal testing platform for energy-related many-core research. Also, due to the computing capability of the SCC, it is possible to deploy multiple programs to run on the SCC simultaneously.

With these characteristics, it is possible to not only run multiple programs on the SCC, but run the programs at different voltage and frequency levels. This allows for a dynamic and energy-aware system when running computations. However, there are many gears to choose from for each program, where a gear is defined as a specific voltage and frequency combination of the cores. Specifically on the SCC, the voltage of the core can range from 0.7V to 1.3V and the frequency can range from 100MHz to 800MHz [5]. An exemplary gear would be [1.1V, 800MHz] for a power domain of 8 cores. This leaves us an abundance of choices to choose from. As a result, an automatic tuning method needs to be developed.

2.2. Related Works. Auto-tuning with specific algorithms is a promising method to reduce the energy consumption of processors. Flautner et al. [7] applied the Differential Evolution algorithm to dynamic voltage scaling problem of processor first. After that, Hsu et al. [8] worked out an algorithm named β algorithm which focused on another power-aware run-time system. These two are early exploratory auto-tuning works. Some auto-tuning works based on SCC platform have been carried out as well. Berry et al. [9] adopted a manual approach to scale voltage and frequency levels on SCC. Later, Berry et al. [10] introduced the Differential Evolution algorithm to solve SCC power-aware computation problem, where a large program with several different phases was tested. Continuing the work on differential evolution, Roscoe et al. [1] expanded the workload on SCC to 3 programs. Recently, Jiang et al. [3] found a better energy configuration with Adaptive Differential Evolution Algorithm (JADE), with 4 programs being examined in their work. The results produced by above researchers are solely relied on the evolution computation algorithm techniques. With respect to the prediction ability of machine learning, we believe using machine learning techniques can enhance the searching ability of the previous evolutionary related algorithms. Yan et al. [2] put forward a method to enhance the DE algorithm by Adaptive LS-SVM method. Zhang et al. [11] performed a survey about machine learning enhanced evolutionary computation.

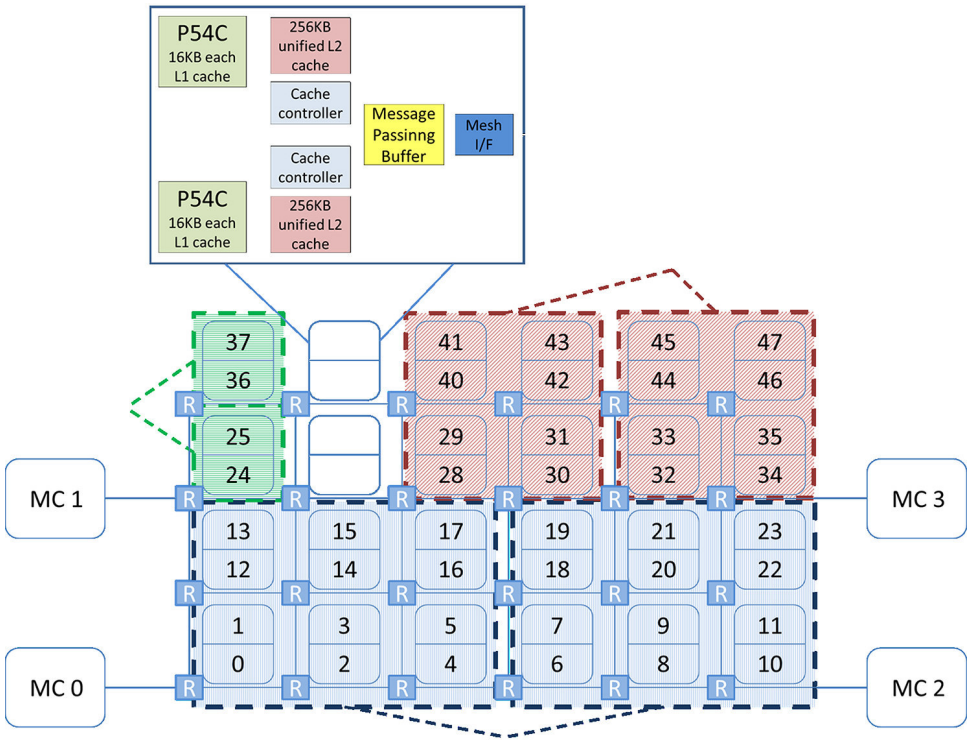

Fig. 2.1: Inner architecture of the SCC

3. Methodology. In order to achieve auto-tuning, the many-core platform needs to dynamically balance voltage, frequency, core numbers occupied by each program. These different variants construct a huge search space, which is difficult to deal with by manual or brute force methods. So the key aspect of finding the most energy-aware way of running the workload is transformed to find the method or algorithm which could converge in the least amount of searching time. In this work, a machine learning enhanced DE algorithm is adopted for this purpose. Classical DE algorithms and our new algorithm are discussed in this section.

3.1. Differential Evolution. Differential Evolution (DE) algorithm is a parallel direct search method. Fig. 3.1 [12] shows an example of the relationship between the searching space and searching vectors of DE.

This algorithm utilizes D-dimensional vectors, where NP vectors constitute the entire population in one generation. In other words, NP stands for the size of the population. The algorithm flow of DE is shown in Fig. 3.2.

The initial population vectors are:

$$x_{i,G}, i = 1, 2, ..., NP \tag{3.1}$$

Selecting one vector from current generation as target vector, randomly selecting other three vectors to perform the arithmetic operation as shown in Fig. 3.2 to get mutated vector which is represented as:

$$v_{i,G+1}, i = 1, 2, ..., NP \tag{3.2}$$

The arithmetic operation is defined by:

$$v_{i,G+1} = x_{r_1,G} + F \cdot (x_{r_2,G} - x_{r_3,G}), i = 1, 2, ..., NP \tag{3.3}$$

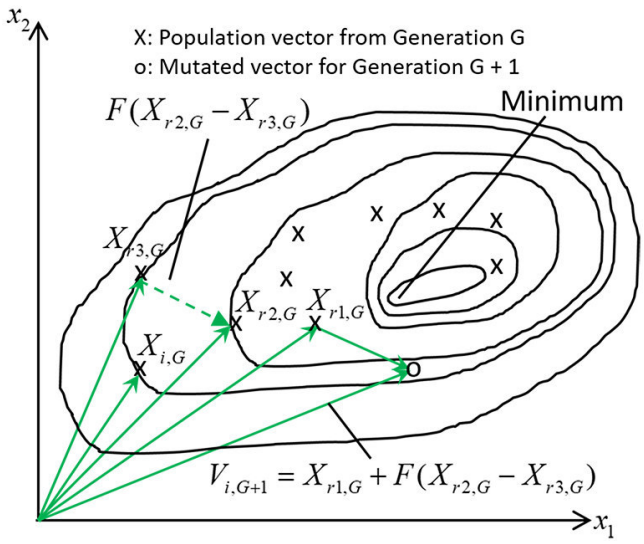

Fig. 3.1: Two-dimensional Function optimization showing contour line and its process for generating $v[i, G+1]$

Here, F is amplification value. Then randomly taking elements of target vector $x_{i,G}$ cross compare with $v_{i,G+1}$, determined by random-generated number and CR factor. This process simulated chromosomal crossover in creature's cell. After that, totally new vectors:

$$u_{i,G+1}, i = 1, 2, ..., NP \tag{3.4}$$

are generated, then these vectors get their fitness value by the evaluation process. In the context of genetic algorithms, fitness is any arbitrary metric that rates the quality of any given candidate. In this work, the fitness we use is energy and EDP. Finally, there is a selection between the parent and the offspring according to their corresponding fitness values. The above process should be applied to each individual in the generation at one time. In this way, the next generation:

$$x_{i,G+1}, i = 1, 2, ..., NP \tag{3.5}$$

is produced successfully.

3.2. JADE. Adaptive Differential Evolution Algorithm (JADE) [13] improved the performance of classical DE algorithm mentioned above. JADE is a greedy algorithm whose mutation strategy is mutating the top $p\%$ of current generation. So one difference between JADE and DE is that mutation operator is changed to:

$$v_{i,G+1} = x_{r_1,G} + F_i \cdot (x^p_{best,G} - x_{r_1,G}) + F_i \cdot (x_{r_2,G} - x_{r_3,G}), i = 1, 2, ..., NP \tag{3.6}$$

Amplification factor F_i is also changed. On the other hand, JADE changed its crossover factor into:

$$\mu_{CR} = (1 - c) \cdot \mu_{CR} + c \cdot mean_A(S_{CR}) \tag{3.7}$$

$$CR_i = randn_i(\mu_{CR}, 0.1) \tag{3.8}$$

where initial μ_{CR} is the mean of the normal distribution with 0.1 variance at first generation. c is a constant to balance the proportions of two mean values. S_{CR} is the set comprised by "successful" CR values of last generation. It is neglected at first generation. The mean here is the arithmetic mean. In this way, JADE introduces the experience from the previous successful generation. For example, classical DE algorithm only

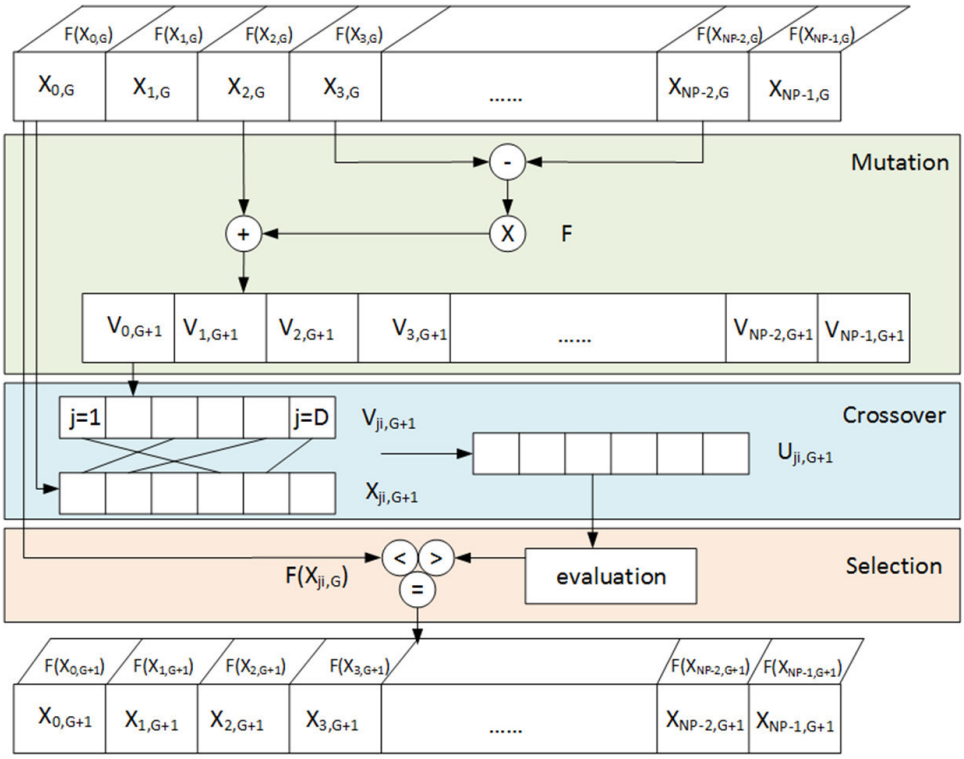

Fig. 3.2: Algorithm Flow of Differential Evolution

mutates one candidate, which is the best one in the generation including 40 candidates. JADE algorithm can mutate top $p\%$ candidates. If p is equal to 20, then 8 candidates will be mutated in the generation including 40 candidates. And if constant c is set to 0.1, the μ_{CR} in $(n+1)_{th}$ generation is made up by 90% μ_{CR} and 10% mean value of "successful" CR values in $(n)_{th}$ generation. And the general flow of JADE can be described in the following pseudo code:

Algorithm 1 Classical DE Algorithm

```
function DE(Initial Candidates, NP, F, CR)
    for (i to NP) do
        P1, P2, ..., Pm ← Candidate Randomly Picking
        V ← P3 + F_i · (P1 + P2) + F_i · (P3 + P4)        ▷ Mutation
        CR_i ← Mean values
        U ← P5, M, CR_i                                   ▷ Crossover
        f(U) ← U
        Compare f(U) with f(P6)                           ▷ Selection
    return New Candidates
```

3.3. Support Vector Machines. Originally a lot of artificial intelligence algorithms were considered as case-by-case method. They performed well on a certain problem, but when the input condition or applied objects changed, the algorithm performed poorly. The creation of the Support Vector Machine (SVM) [14] algorithm largely improved this kind of drawback. Initially, SVM method was mainly used for classification problem. Later, it was extended to the case of regression. In our paper, we use the regression function of the SVM, which could also be called Support Vector Regression (SVR). The input of the SVM algorithm is training

data sets, which could be presented as n-dimensional vector x:

$$x = \{x_i\}_{i=1}^{n} \tag{3.9}$$

and the classification function of the data set is:

$$f(x) = \sum_{i=1}^{n} w_i \langle (x_i), (x) \rangle + b \tag{3.10}$$

Training data set could be mapped into another high-dimensional space by mapping kennel. The SVM model in the high-dimensional space could be represented as follows:

$$f(x) = \sum_{i=1}^{n} w_i \langle \phi(x_i), \phi(x) \rangle + b \tag{3.11}$$

Here, with respect to simplification of description, we use a 2-dimensional training data set example to express the theory of mapping [17]. Assuming two 2-dimensional vectors $x_1 = (\eta_1, \eta_2)$ and $x_2 = (\xi_1, \xi_2)$, and $\phi(\cdot)$ is the mapping relationship from 2-dimension to 5-dimension. In this way:

$$\langle \phi(x_i), \phi(x) \rangle = \eta_1\xi_1 + \eta_1^2\xi_1^2 + \eta_2\xi_2 + \eta_2^2\xi_2^2 + \eta_1\eta_2\xi_1\xi_2 \tag{3.12}$$

In another point of view:

$$(\langle \phi(x_i), \phi(x) \rangle + 1)^2 = 2\eta_1\xi_1 + \eta_1^2\xi_1^2 + 2\eta_2\xi_2 + \eta_2^2\xi_2^2 + 2\eta_1\eta_2\xi_1\xi_2 + 1 \tag{3.13}$$

Equation 3.12 and Equation 3.13 share the same shape. It is not hard to get the expression of $\phi(\cdot)$, which is:

$$\phi(x_1) = (\sqrt{2}\eta_1, \eta_1^2, \sqrt{2}\eta_2, \eta_2^2, \sqrt{2}\eta_1\eta_2, 1) \tag{3.14}$$

Here, Equation 3.12 calculates in the high-dimension space, while Equation 3.13 calculates easily in the low-dimension space and avoids to express the mapping relationship explicitly. The method of Equation 3.13 is called kernel function method, which reduces the computation complexity by committing computation in the high-dimensional space of problem.

3.4. SVM-JADE. When multiple programs are running simultaneously on the SCC, energy consumption will differ by using different number of cores and gears of voltage and frequency. Among them, finding the configuration vector x_D which make SCC consume the lowest energy-delay product (EDP) or energy is meaningful. The vector x_D is made up of gears and number of cores. The G_i represent the gear level of i_{th} program, and the C_i represent the number of cores of i_{th} program. Considering 4 programs are running at the same time in our experiment, our x_D could be represented as:

$$x_D = (G_1, G_2, G_3, G_4, C_1, C_2, C_3, C_4)^T \tag{3.15}$$

with constraints:

1. The total number of cores used by all 4 programs cannot exceed 48 at any time, which is the total number of cores of the SCC platform.
2. One individual program must be run under one gear, even if it occupies more than one power domain.
3. Number of possible gears are regulated down to 5, one of which is an idle state to be used on a domain when no cores from it are in use. The different gears can be seen in Table 3.1. This limitation on gears means that each voltage level will be using the maximum possible frequency for that power level, which allows no power to be wasted.

Table 3.1: Different gear combinations in JADE

Gear	Frequency	Voltage
1	800 MHz	1.1 V
2	533 MHz	1.0 V
3	400 MHz	0.9 V
4	320 MHz	0.8 V
Idle	100 MHz	0.7 V

The third constraint reduces the search space for brute force searching to a certain degree, as it removes some unnecessary state. But even under this scenario, assuming a 4-program case where each program has one of 5 possible gears and is limited to a maximum of 8 cores, we still have a search space of size 2.56×10^6. With such a search space, brute force searching is still too labor intensive to be feasible.

Yan et al. [2] put forward a method to enhance the DE algorithm by Adaptive LS-SVM method. Synthesizing the advantages of above-mentioned algorithm, we adopt a method named SVM-JADE to accelerate the converge speed of finding optimum low-power combination in SCC auto-tuning system. The training set of this problem could be presented as: $\{x_i, f(x_i)\}_{i=1}^{n}$. The flowchart of SVM-JADE is given in Fig. 3.3.

The SVM-JADE embedded an SVM prediction procedure after the selection procedure of each generation. At the end of one generation, current population status is defined as:

$$s_{i,G+1}, i = 1, 2, ..., NP \tag{3.16}$$

while selecting the n best individuals among Equation 3.16 and their corresponding fitness values as input into SVM model to fit a regression hyper-plane for predicting Energy or EDP value of SCC platform. Next, using regression hyper-plane performing the vicinity search around the best performance vector in Equation 3.16. The vicinity method is shown in Fig. 3.4. Here, with respect to the difficulty in visualizing a D-dimensional space when $D > 3$, we use 3-dimensional vector as an example to explain the search procedure.

With vicinity search, we obtain the optimum vector. Optimum vector is predicted by the SVM regression model which compares across the fitness values of individual in current generation. Specially, the optimum vector should be feasible, i.e., could be executed on the SCC platform. Then, the configuration of optimum vector will be tested on the SCC to get the actual fitness value, being energy or EDP in our case. Finally, compare the fitness of optimum vector with lowest fitness of vectors in Equation 3.16, the candidate with better fitness value stays. Then we get the generation in Equation 3.5 of SVM-JADE. At the end of every generation, repeat these procedures. The evolution is usually stopped either after a certain number of generations or by reaching a specified threshold.

4. Implementation. The general diagram of the implementation of SVM-JADE is shown on Fig. 4.1. Our initial approach in implementing our SVM was to test the SVM in MATLAB using the libSVM library [15] as a quick prototyping environment. After each generation, a population of 40 individuals is generated. We input them into SVM model with different size of data-sets, varying from D to NP. We find that the lowest mean squared error (MSE) between actual value and regression value of individual is obtained when the size of training set is D, which is 8 in this case. That is to say, the squared correlation coefficient is almost equal to 1.0 in this size of training data set.

Our implementation here used both the theories of JADE and SVM as discussed in Section 3. We started with a base implementation of JADE as laid out in [3]. This implementation used the Inspyred [16] python library to assist in the evolutionary computations. In accordance with the theory of JADE, a modified version of the mutation and crossover steps were used in place of the standard DE versions. Other things we followed were running four programs simultaneously and the same fitness tests. The two different fitness types used here are energy and energy-delay product (EDP). Aside from adding an SVM portion to JADE, there were a few other modifications made as well. SVM-JADE has an auto-initialization at the beginning to ensure that all tests

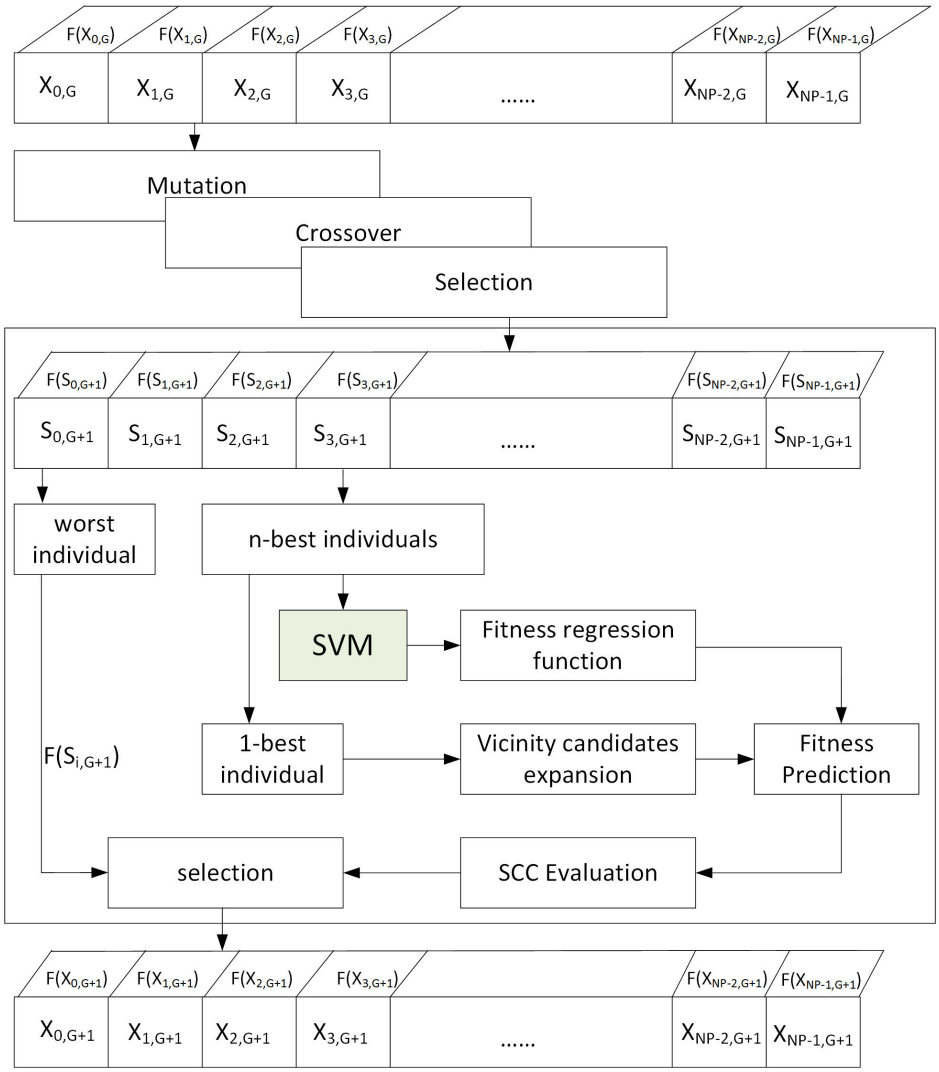

Fig. 3.3: Algorithm Flow of SVM-JADE

begin with the same initial conditions. Another change is that the initial population can be predetermined, which allows for more specific testing including continuing testing in the case of a failure during experiment.

The four programs used in our experiments are *mmul*, *conv*, *cpi*, and *seqpi*, respectively. *mmul* is a matrix multiplication program which can be partially paralleled without error. *conv* is a program similar to *mmul* by operating on matrices, however, *conv* finds the convolution of the two matrices which is much easier to parallelize. *cpi* is a highly parallel program that calculates the value of π, stopping after a given number of iterations. The iterations number used here is one billion, which is the same as used in JADE. Lastly, we have *seqpi* which is a very sequential program. This accomplishes the same task as *cpi* but in sequential fashion. We use these four programs to form our multi-program workload for the many-core processor, the same as described in [3].

In order to add the SVM module into JADE, we again used the libSVM library [15]. However, this time we used the python extensions to make combining the algorithms easier. This also made organizing our implementation easier by designing our SVM as an object that is able to perform all necessary computations on its own.

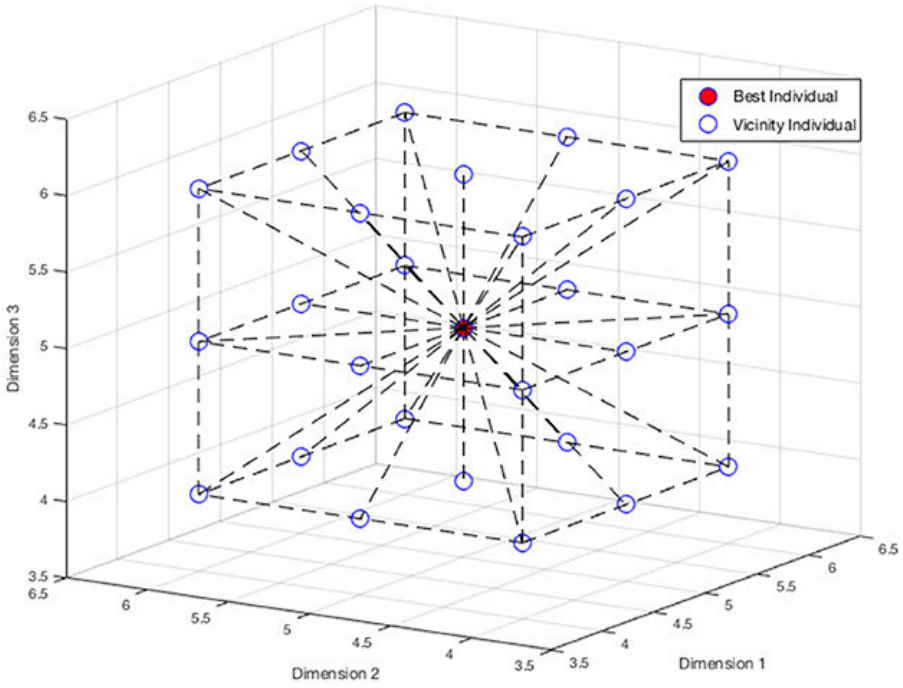

Fig. 3.4: Demo of vicinity Search

Our SVM was inserted into the selection phase, being trained on the candidates of the current population based on the current fitness. After being trained, the SVM then generates its own search space for both gears and core count. The search space generated follows the algorithm laid out below:

Algorithm 2 Search Space Generation for Gears

```
1: function GearSpace(candidate, dimension)
2:     value ← candidate[dimension]
3:     if value = 1 then
4:         return [1, 2]
5:     else if value ∈ [2, 3] then
6:         return [value − 1, value, value + 1]
7:     else if value = 4 then
8:         return [3, 4]
```

As can be determined from the algorithm, for the search space of gears we end up with a search size of 2 or 3. Similarly, when generating the search space for the core count, we end up with possible sizes of 3 and 5. These are a bit larger because there is a larger set of values to choose from.

In summary, our algorithms form small search space around the best candidate of the offspring population and uses the SVM decision function to predict the different fitness values for each candidate in the search space. The smallest value is then chosen as the SVM predicted candidate, because we are trying to minimize the fitness value, such as energy usage. This candidate is then evaluated against the worst candidate of the offspring population, replacing it if the SVM prediction is better. This modified offspring population becomes

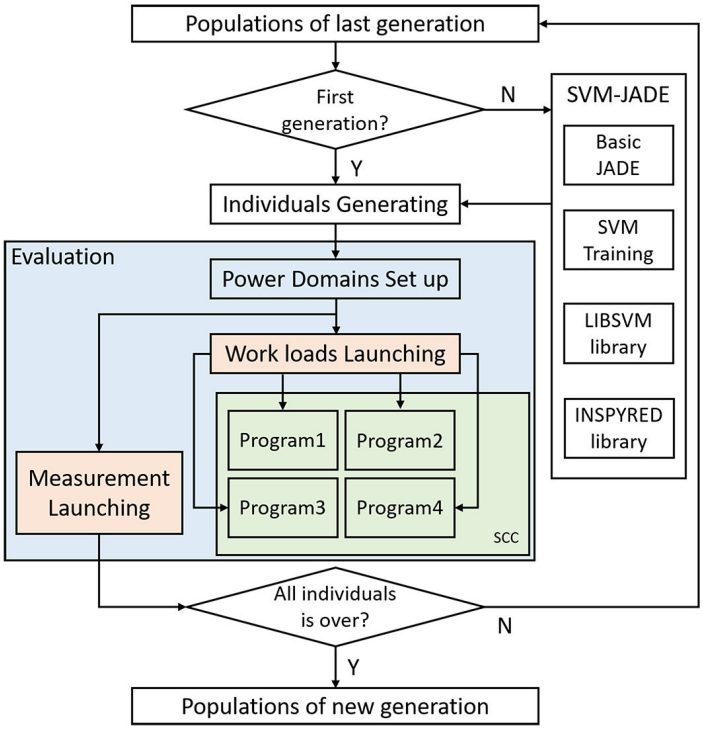

Fig. 4.1: Implementation of SVM-JADE

Algorithm 3 Search Space Generation for Core Count

```
1: function CoreSpace(candidate, dimension, max)
2:     value ← candidate[dimension]
3:     if value ∈ [1, 2] then
4:         return [1, 2, 3, 4, 5]
5:     else if 3 ≤ value ≤ max − 2 then
6:         return [value − 1, value, value + 1]
7:     else if value] ∈ [max − 1, max] then
8:         return [max − 4, max − 3, max − 2, max − 1, max]
```

the new population for the next generation and continues to follow DE's flow of operations shown in Fig. 3.2.

5. Experimental Results. SVM-JADE described in Section 4 was implemented onto the SCC with the Intel SCCkit v1.4.0. The controlling computer, the MCPC, runs Ubuntu 10.04 64-bit with 8GB of RAM. The initial setup conditions for all the tests were that no single program could use more than 32 cores at any given time. This limitation is in place to ensure each program has at least one power domain (group of 8 cores). The SVM is trained on the top 8 candidates of the population as opposed to the entire population, and the four programs are run simultaneously. As we mentioned previously, energy and energy-delay product (EDP) are used as fitness values. EDP is a measure of both performance as well as energy as it takes both the energy it takes to execute the program as well as the user response time, e.g., the time to execute the program into consideration. The different fitness values were tested individually 3 times and the results are aggregated by average value. Since SVM-JADE and JADE are based on DE, the important part of the data is the fitness value of the single best candidate from each generation. This also corresponds to the data analysis performed in [3]. In order to conduct a fair comparison between SVM-JADE and Jade, the random candidate generator

for the initial generation (Generation 0) was fixed to the same seed.

Our first experiment was to use the EDP fitness type. The results are presented in Fig. 5.1, which represents the EDP from the best single candidate of each generation averaged across all runs. It can demonstrate the overall performance of SVM-JADE vs. Jade including the convergence rate.

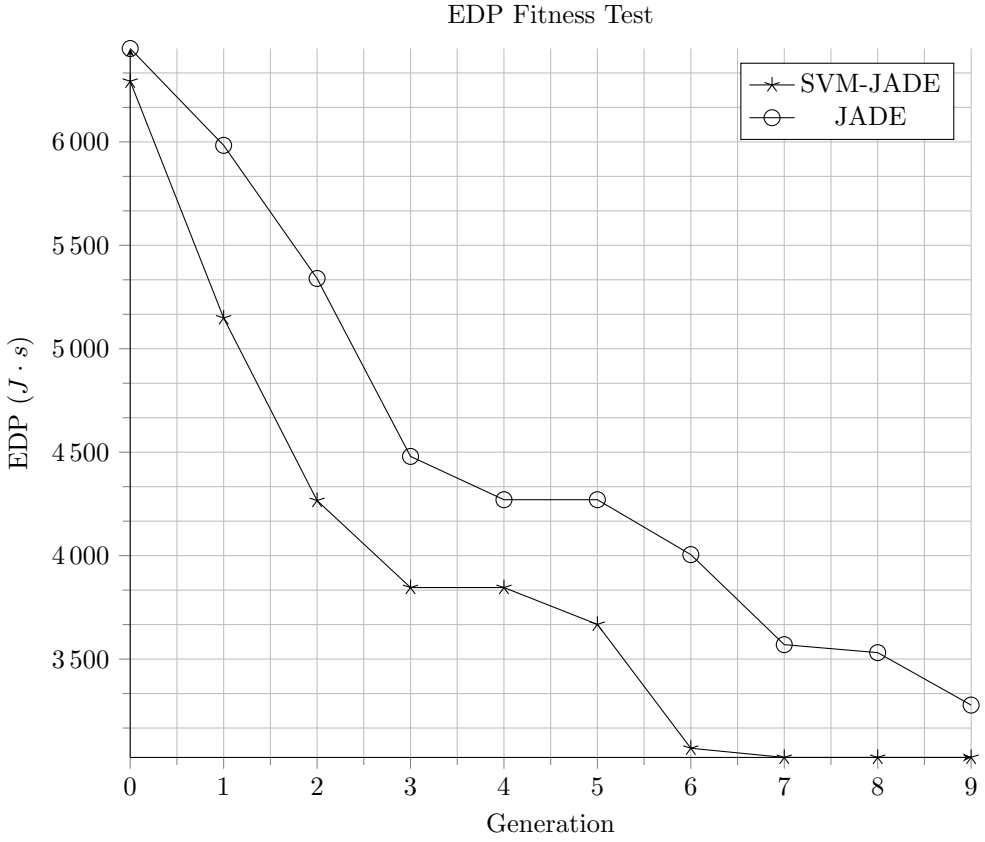

Fig. 5.1: Algorithms using EDP fitness

From Fig. 5.1, SVM-JADE achieved an EDP of 3024.499 Joule-seconds in Generation 9 while Jade ended up only decreasing the EDP to 3277.522 Joule-seconds in Generation 9. SVM-JADE is able to further reduce the EDP by 7.7% than JADE. Past Generation 6, SVM-JADE decreases by less than 1.5% showing convergence. Jade, however, is still continuing to decrease even through Generation 9.

By evaluating the difference across generations, we find that SVM-JADE results in an EDP changing rate at about -322.952 Joule-seconds per generation while Jade results an rate of -352.646 Joule-seconds on average. This point can be exemplified through a linear regression across the convergence. That is to say SVM-JADE should be regressed from Generations 0 to 6, while Jade should be regressed from Generations 0 to 9. The point of doing a linear regression across the convergence is to show the rate at which the algorithms converged since they ideally converge before the end. The regressions shows that SVM-JADE has a trend of $EDP = -427.49X + 5963.4$ and JADE has a trend of $EDP = -339.42X + 6384.6$. As can be seen from the slopes, SVM-JADE decreased to its convergence point at a quicker rate than Jade. The actual values for both SVM-JADE and JADE for EDP-based fitness can be seen in Table 5.1 with SVM-JADE in subtable (a) and JADE in (b).

The best overall result using EDP fitness from a single run was with vector $[4, 4, 4, 4, 11, 12, 23, 1]$ yielding 2997.477 Joule-seconds. This means that all four programs ran on Gear 4 and Programs 1 through 4 ran on 11, 12, 23, and 1 cores, respectively. Recall that the four programs used were *mmul*, *conv*, *cpi*, and *seqpi*.

cpi, the most parallel of all the programs, used the most cores of 23. *mmul* and *conv* are both fairly parallel programs and they used 11 and 12 cores, separately. The last program, *seqpi*, is a very sequential program and consequently only used a single core, showing the effectiveness of our proposed approach.

Table 5.1: Tabular data for EDP based fitness

Generation	EDP $(J \cdot s)$
0	6 292.992
1	5 148.874
2	4 266.357
3	3 845.74
4	3 845.74
5	3 667.679
6	3 068.848
7	3 024.499
8	3 024.499
9	3 024.499

(a) SVM-JADE

Generation	EDP $(J \cdot s)$
0	6 451.335
1	5 983.302
2	5 339.798
3	4 479.627
4	4 270.295
5	4 270.295
6	4 004.921
7	3 569.608
8	3 531.114
9	3 277.522

(b) JADE

The next form of experiment was to use energy as the fitness metric in SVM-JADE and JADE. The results for the energy fitness test are presented in Fig. 5.2.

We can see again that SVM-JADE stops significant changes in Generation 6. Jade, however, still continues to decrease even through to Generation 9. SVM-JADE achieved an energy value of 392.945 Joules in Generation 9 while Jade ended up only decreasing the energy to 428.632 Joules in Generation 9. SVM-JADE is able to reduce the energy consumption further by 8.3% than JADE. Past Generation 6, SVM-JADE continues to improve at a slowed rate as before, but JADE does not converge even by Generation 9. In this case, we have reached our near-ideal value very quickly.

Evaluating the difference across generations, we can find that SVM-JADE is changing at a rate about -16.277 joules per generation while Jade results a rate of -17.658 joules per generation on average. This can be shown via a linear regression across the convergence. That is to say SVM-JADE should be regressed from Generations 0 to 6 while Jade should be regressed from Generations 0 to 9. The regressions shows that SVM-JADE has a trend $Energy = -22.349X + 550.6$ and Jade has a trend $Energy = -17.615X + 587.55$. As can be seen from the slopes, SVM-JADE decreased to its convergence point at a quicker rate than Jade. The actual values for both SVM-JADE and JADE for energy-based fitness can be seen in Table 5.3 with SVM-JADE in subtable a and JADE in b.

The best overall result using energy fitness from a single run was with vector $[1, 4, 1, 1, 10, 10, 21, 1]$, yielding an energy consumption of 399.753 Joules. This means that *mmul*, *cpi*, and *seqpi* all ran on Gear 1 and *conv* ran on Gear 4. This is reasonable as *conv* is less sequentially heavy than programs such as *mmul* or *cpi*. *mmul* and *conv* both ran on 10 cores as they are both mainly parallel programs. *cpi*, the most embarrassingly parallel program, again has the highest core count of 21. Lastly, *seqpi*, the purely sequential program, was again dedicated just a single core, as was the case with EDP-based fitness.

It is also useful to observe how the candidate evolves across generations based on our algorithms, mapping to the program descriptions as seen in Sec. 4. Table 5.5 shows example candidates from a single run using EDP-based Fitness.

Please note here each gear and core combination in the table lines up with the program *mmul*, *conv*, *cpi* and *seqpi* respectively. Looking at $Gear_1$ and $Cores_1$, we can see that in the final generation it ends with 1 and 9 respectively. Using Gear 1 allows for heavy computations, and 9 cores allow for a semi-parallel program, which matches the previous description. *conv* obtains a gear of 4 and 16 cores because the computations are not too difficult, but the program runs much faster on multiple cores. *cpi*, a very parallel program ends with a

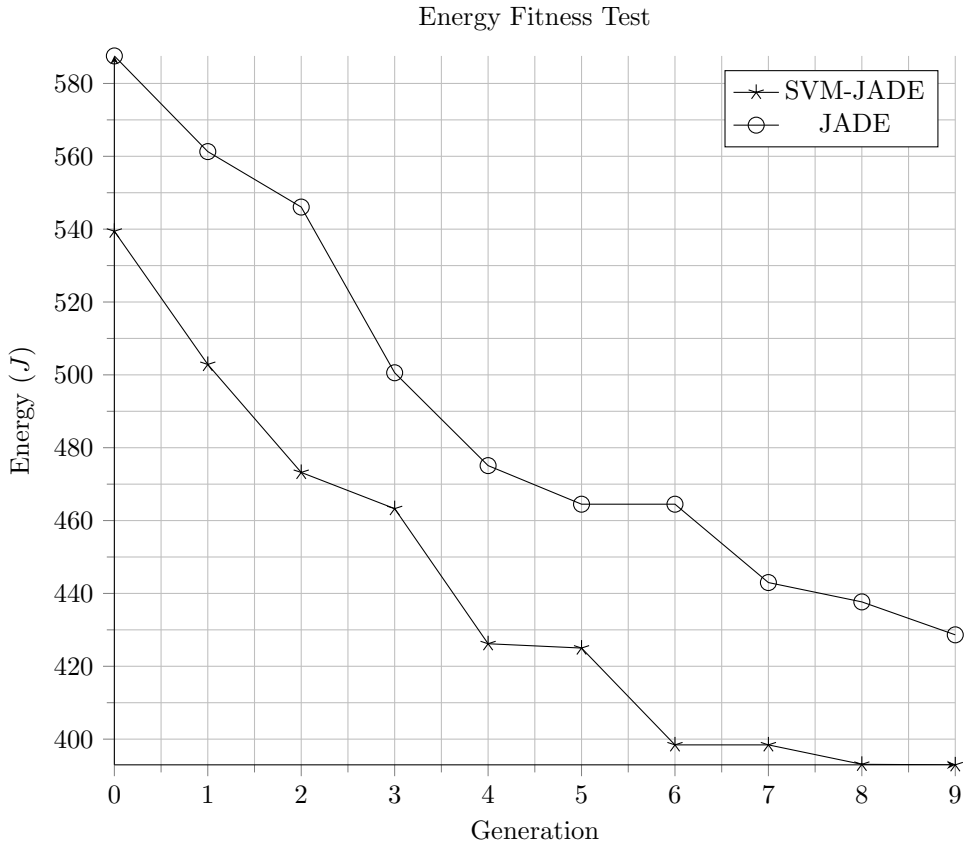

Fig. 5.2: Algorithms using Energy fitness

gear of 1 and 17 cores. Gear 1 is caused by the complexity of the integral computation, and 17 cores because the program is embarrassingly parallel. Finally, *seqpi* receives gear 1 and only 1 core. This is because it has complex computations but is a very sequential program and cannot be easily run with multiple cores.

Lastly, it is important to see the savings provided by SVM-JADE over other algorithms such as Jade. In terms of computation overhead, because SVM-JADE converges in Generation 6 as opposed to Generation 9 for Jade, in practice we are able to save the calculation and testing of over 120 candidates. This results a reduction in total computation overhead by one third.

6. Conclusions. Working in many-core computing adds great computational horsepower but also brings in added energy consumption. How to identify the most energy-efficient way to run a specific workload while still maintain a satisfactory performance level is always a challenge. The work we presented here shows that a brute force search for an ideal DVFS configuration can be avoided by using an automatic tuning algorithm. In this work, we were able to successfully apply Support Vector Machine (SVM) to an Adaptive Differential Evolution algorithm (JADE), creating a new automatic tuning algorithm SVM-JADE. SVM-JADE is more effective than JADE, as the final EDP value achieved by SVM-JADE is 7.7% less than that of JADE and the final energy value achieved by SVM-JADE is 8.3% less than that of JADE, respectively, after running 10 generations. Besides, SVM-JADE, was able to minimized desired fitness levels quicker than JADE with faster convergence speed. This is a significant improvement since each candidate from these generations gets mutated and compared. The overhead brought by adding an SVM evaluation becomes negligible when considering the potential savings in calculations and energy saving it provides. SVM-JADE is a step forward in DVFS automatic tuning on many-core platforms by minimizing fitness values lower and quicker than a representative comparable

Table 5.3: Tabular data for Energy based fitness

Generation	Energy (J)
0	539.435
1	502.859
2	473.217
3	463.278
4	426.189
5	425.005
6	398.424
7	398.424
8	393.088
9	392.945

(a) SVM-JADE

Generation	Energy (J)
0	587.553
1	561.293
2	546.062
3	500.576
4	475.089
5	464.496
6	464.496
7	442.984
8	437.675
9	428.632

(b) JADE

Table 5.5: Candidate examples from a single run

Generation	$Gear_1$	$Gear_2$	$Gear_3$	$Gear_4$	$Cores_1$	$Cores_2$	$Cores_3$	$Cores_4$
0	1	3	1	1	10	11	11	11
1	1	3	1	2	8	8	12	9
2	1	3	1	2	8	8	12	9
3	1	1	1	1	14	11	18	5
4	1	1	1	1	14	11	18	5
5	1	1	1	1	14	11	18	5
6	1	1	1	1	12	11	18	6
7	1	4	1	1	9	16	17	1
8	1	4	1	1	9	16	17	1
9	1	4	1	1	9	16	17	1

algorithm JADE. Our future work includes adjusting our SVM, its training set, as well as the decision function to continue improve the accuracy of support vector regression when selecting candidates.

7. Acknowledgement. The authors would like to thank Intel Labs for providing us with the SCC platform to conduct this research. The authors would also like to thank the anonymous reviewers for their feedbacks that greatly improved the quality of this paper. This material is based upon work supported by the National Science Foundation under Grant No. ECCS-1301953. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of either Intel or the National Science Foundation.

REFERENCES

[1] B. Roscoe, M. Herlev, C. Liu, *Auto-tuning multi-programmed workload on the scc*, 2013 International Green Computing Conference Proceedings, 06 2013.
[2] X. Yan, M. Wu, B. Sun, *An adaptive ls-svm based differential evolution algorithm*, 2009 International Conference on Signal Processing Systems, 2009.
[3] Y. Jiang, X. Qi, C. Liu, *Energy-aware automatic tuning on many-core platform via differential evolution*, The 5th International Workshop on Power-aware Algorithms, Systems, and Architectures, 2016.
[4] I. Labs, *Scc platform overview*, Intel Many-Core Application Research Community, 2010.

[5] I. Labs, *The SCC Programmer's Guide*, Intel Many-Core Application Research Community, 2012.
[6] G. Torres, J. Chang, F. Hua, C. Liu, S. Schuckers, *A Power-Aware Study of Iris Matching Algorithms on Intel's SCC*, Proceedings of the 2013 42Nd International Conference on Parallel Processing, 2013.
[7] K. Flautner, S. Reinhardt, T. Mudge, *Automatic performance setting for dynamic voltage scaling*, pp. 260–271, 2001.
[8] C.-h. Hsu and W.-c. Feng, *A power-aware run-time system for high-performance computing*, pp. 1–, 2005.
[9] K. Berry, F. Navarro, C. Liu, *A manual approach and analysis of voltage and frequency scaling using scc*, pp. 1–4, 2012.
[10] K. Berry, F. Navarro, C. Liu, *Application-level voltage and frequency tuning of multi-phase program on the scc*, pp. 1:1–1:7, 2013.
[11] J. Zhang, Z. h. Zhan, Y. Lin, N. Chen, Y. j. Gong, J. h. Zhong, H. S. H. Chung, Y. Li, Y. h. Shi, *Evolutionary computation meets machine learning: A survey*, IEEE Computational Intelligence Magazine, vol. 6, pp. 68–75, 2011.
[12] R. Storn, K. Price, *Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces*, Journal of global optimization, vol. 11, no. 4, pp. 341–359, 1997.
[13] J. Zhang, A. C. Sanderson, *Jade: adaptive differential evolution with optional external archive*, IEEE transactions on evolutionary computation, vol. 13, no. 5, pp. 945–958, 2009.
[14] C. Cortes, V. Vapnik, *Support-vector networks*, Machine learning, vol. 20, no. 3, pp. 273–297, 1995.
[15] C.-C. Chang, C.-J. Lin, *LIBSVM: A library for support vector machines*, ACM Transactions on Intelligent Systems and Technology, vol. 2, pp. 27:1–27:27, 2011. Software available at http://www.csie.ntu.edu.tw/ cjlin/libsvm.
[16] A. Garrett, *Inspyred: A framework for creating bio-inspired computational intelligence algorithms in python*. Software available at https://aarongarrett.github.io/inspyred/.
[17] CSDN, *Support Vector Machine Learning Notes–Three Levels*, 2012. Available at http://blog.csdn.net/v_july_v/ article/details/7624837.

Edited by: Pedro Valero Lara
Received: Dec 23, 2016
Accepted: May 24, 2017