A Parallel Memetic Algorithm on GPU to Solve the Task Scheduling Problem in Heterogeneous Environments

Sayyed Ali Mirsoleimani
School of Electrical and Computer Engineering
Shiraz University
Shiraz, Iran
alimirsoleimani@gmail.com

Ali Karami
School of Electrical and Computer Engineering
Shiraz University
Shiraz, Iran
karami@cse.shirazu.ac.ir

Farshad Khunjush
Department of Electrical and Electronics Engineering
Hormozgan University
Bandar Abbas, Iran
School of Electrical and Computer Engineering
Shiraz University
Shiraz, Iran
khunjush@cse.shirazu.ac.ir

ABSTRACT

Hybrid metaheuristics have shown their capabilities to solve NP-hard problems. However, they exhibit significantly higher execution times in comparison to deterministic approaches. Parallel techniques are usually leveraged to overcome the execution time bottleneck for various metaheuristics. Recently, GPUs have emerged as general purpose parallel processors and have been harnessed to reduce the execution time of these algorithms. In this work, we propose a novel parallel memetic algorithm which is fully offloaded onto GPUs. In addition, we propose an adaptive sorting strategy in order to achieve maximum possible speedups for discrete optimization problems on GPUs. In order to show the efficacy of our algorithm, a task scheduling problem for heterogeneous environments is chosen as a case study. The output of this problem can have a tangible impact on overall performance of parallel heterogeneous platforms. The achieved results of our approach are promising and show up to 696x speedup in comparison to the sequential approach for various versions of this problem. Moreover, the effects of key parameters of memetic algorithms in terms of execution time and solution quality are investigated.

Categories and Subject Descriptors

G.1.0 [**Numerical Analysis**]: General—*Parallel algorithms*; F.2.2 [**Analysis OF Algorithms And Problem Complexity**]: Nonnumerical Algorithms and Problems—*Sequencing and scheduling*

General Terms

Algorithms

Keywords

Memetic algorithms, Task scheduling, GPU computing, Parallel algorithms

1. INTRODUCTION

Metaheuristic algorithms have been increasingly used to solve NP-hard problems. This is mainly because of the flexibility and effectiveness of these algorithms in comparison to heuristic methods. There are two main classes of metaheuristics: the trajectory and population-based classes. Trajectory methods only use a single solution to find the problem's answer while the population-based ones deal with a set of solutions in each iteration of the algorithm. Hybridization of these approaches has provided more powerful search techniques called *Memetic Algorithms* (MAs). The efficacy of hybrid algorithms has been demonstrated in many practical academic optimization problems [8]. These approaches usually deliver better solution qualities compared to heuristic methods. Nevertheless, their execution times remain limiting factors to using them in practical applications. One proper solution for tackling the execution time problem is to use parallel computing. Therefore, we have chosen the GPU architecture to propose a parallel memetic algorithm.

In order to evaluate the effectiveness of our parallel algorithm, a multiprocessor task scheduling problem is chosen as a case study. Given a set of tasks and a set of processors in a parallel computing system, the main objective of this problem is to map these tasks to the processors in such a way that the entire tasks complete at the earliest time. In general, there exist two types of task scheduling: *static* and *dynamic*[3]. In a static scheduling, a scheduler assigns tasks to available processors with prior knowledge of the tasks processing times, their precedence relationships, and their related communication costs. On the other hand, in a dynamic scheduling the states of the tasks are only known during the execution. In this paper, only the static scheduling problem is addressed.

The task scheduling problem remains to be NP-hard[17]. Several heuristic-based methods (e.g., Heterogeneous Earliest-Finish-Time (HEFT) [15] etc) have been proposed to solve this problem instantly. However, as these solutions lack high qualities, metaheuristic algorithms such as Ant Colony Optimization [16], Artificial Immune System [20], Genetic Algorithm (GA) [19], MA [18] and other evolutionary algorithms have been increasingly leveraged to reach high quality solutions, but the execution times of these methods remain challenging concerns in comparison to heuristic methods. The execution time of any algorithm to find an effective scheduling is an important efficiency metric, which means the one with a minimum running time could be the most practical implementation [15]. The proposed approach in this paper is mainly inspired by the fact that parallel MAs might find high quality solutions in a reasonable and practical time.

The main contribution of this paper is to propose a novel and efficient parallel MA on GPU architectures to solve a combinatorial optimization problem, that is, task scheduling. To the best of our knowledge, there is no any result in the literature regarding parallelizing the heterogeneous multiprocessors task scheduling problem on GPUs. The key features of our work include: first, running the whole search process of the MA on the GPU side; therefore, it does not require any computation on the CPU side. Thus, unnecessary data transfers between CPU and GPU are eliminated. Second, it speeds up the search process considering the existence of fast GPU's memories. Third, it proposes an efficient problem representation that is specially designed for the task scheduling problem. Finally, it introduces an adaptive sorting strategy based on population sizes in order to achieve higher speedups for small instances of the problem. The achieved results of our method are promising and reveal that higher quality solutions for the task scheduling problem could be reached in a reasonable amount of time as compared to deterministic methods.

The remainder of this paper is organized as follows. Section 2 surveys the literature. The task scheduling problem and its sequential hybrid evolutionary solution alongside GPU programming are provided in Section 3. Section 4 describes our proposed parallel method for accelerating MAs. The results are presented and discussed in Section 5. Finally, we conclude the paper in Section 6.

2. RELATED WORK

Nesmachnow et. al. [9] proposed GPU implementations of two heuristics for solving the scheduling of the independent tasks problem. Pinel et. al. [13] also proposed a parallel cellular genetic algorithm and a heuristic method for solving the same problem on GPUs. The selected problem and approaches of these two papers are different from our work.

Luong et. al. [6] proposed and implemented several schemes for island GA on GPUs. Although, they claimed considerable speedup in comparison to a single CPU implementation for continuous optimization problems, the proposed approaches could not be applicable for combinatorial optimization problems such as task scheduling due to shortage of shared memory space on GPUs. Luong et. al. [7] also proposed a cooperative CPU-GPU model to implement a hybrid genetic algorithm for solving Quadratic Assignment Problem (QAP). The major bottleneck in this model is the communication between the CPU and GPU, which degrades the performance of the algorithm. Kruger et. al. [5] implement a MA which is a combination of an evolutionary algorithm and a simple deterministic local search. They did not consider the efficiency of algorithm in finding the best solution. In this approach the evolutionary engine is kept on the CPU and the evaluation function and local search operations are transferred to the GPU for execution. However, to overcome the overhead of data transfers between the CPU and GPU large population sizes are required. On the contrary to these approaches, our proposed parallel MA is fully offloaded onto a GPU and as a result no inter-processor data transfer is necessary. Furthermore, we show that it is possible to reach orders of magnitude speedup even for small populations by using an efficient algorithm.

3. BACKGROUND

In this section we provide the required background on the problem description and GPU programming.

3.1 Problem Description

In this part, we provide a formal description of the static multiprocessor scheduling problem. In this problem, we are given m heterogeneous processors (j=1,...,m), which are fully connected through a network. Furthermore, a parallel program, decomposed into n smaller tasks with dependencies representing the precedence constraints (i=1,...,n), is also provided. These dependencies are usually shown by *Directed Acyclic Graphs* (DAGs), which we call them as *Task Precedence Graphs* (TPGs). A TPG consists of a set of nodes (V) as tasks and a set of edges (E) as the precedence constraints among the tasks. In this graph, the starting node of an edge is called *predecessor* and the ending node of the edge is called *successor*. The precedence constraints between two tasks mean that the result of the predecessor task must be transferred to the successor task before starting its execution. When two dependent tasks are assigned to two different processors, communication cost should be paid to transfer the required data from the predecessor to the successor task. The ultimate goal is to run tasks on available processors in parallel with focus on minimizing the *makespan*.

As we are dealing with heterogeneous systems, there are various types of processing elements with different computational capabilities. In our representation, the computation costs of tasks on processors are stored in a matrix called W. This matrix has dimensions of $n \times m$ and the value of W_{ij} represents the computation cost of task T_i on processor P_j. The weights associated with edges of the TPG are also stored in a communication cost matrix (C). The matrix C has dimensions of $n \times n$ and the value of C_{ij} is the communication cost between task T_i and task T_j. Consider an instance with n=10 tasks, and m=4 processors, computation costs are presented in Table 1, and a sample TPG, representing associated communication costs to its edges, is shown in Figure 1. A possible schedule is also illustrated in Figure 2.

In order to solve the task scheduling problem several attributes should be defined. First, the earliest time when a processor will be available for executing a new task is called *Earliest Start Time* (EST), which can be measured by (1).

$$EST(T_i, P_j) = max\{\max_{T_k \in run(P_j)} \{AFT(T_k)\}, \max_{T_l \in Pred(T_i)} \{AFT(T_l + C_{li})\}\} \quad (1)$$

In the first step, the maximum of *Actual Finish Time* (AFT) for all tasks that have already been assigned to processor P_j ($run(P_j)$) is found. This value shows when processor P_j is available to execute a new task T_i. Next, in order to execute T_i on processor P_j, the maximum time needed for

Table 1: Computation cost matrix

	P_1	P_2	P_3	P_4
T_1	33	47	42	31
T_2	33	44	32	38
T_3	33	39	31	30
T_4	40	43	34	51
T_5	44	31	41	35
T_6	35	38	57	38
T_7	41	38	36	40
T_8	42	37	41	42
T_9	31	31	37	57
T_{10}	38	37	44	43

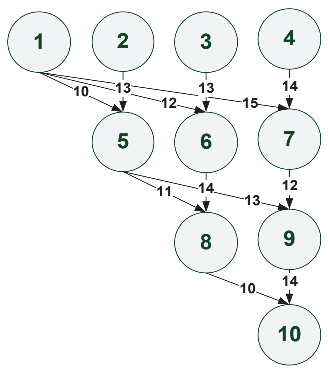

Figure 1: TPG graph

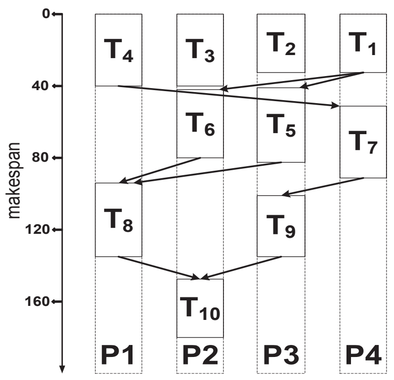

Figure 2: Sample schedule

transferring all required data from the predecessors of T_i to processor P_j is calculated. As mentioned before, this time is equal to 0 if both tasks are assigned to the same processor. *Earliest Finish Time* (EFT) of task T_i is equal to the time that the execution of task T_i on processor P_j is finished and is defined as follow:

$$EFT(T_i, P_j) = EST(T_i, P_j) + W_{ij} \tag{2}$$

When task T_i is explicitly allocated to processor P_j the EFT of all tasks are assigned to AFT. The *makespan* of the schedule is the largest AFT among all tasks and defined by (3).

$$makespan = \max_{T_i \in V} \{AFT(T_i)\} \tag{3}$$

In the next section, a sequential hybrid metaheuristic for solving the task scheduling problem will be briefly surveyed.

3.2 Sequential Algorithm

In order to evaluate our proposed parallel algorithm for MAs, a hybrid metaheuristic method proposed by Wen et. al. [18] for solving task scheduling problem is selected. This hybrid algorithm combines two metaheuristics: GA and VNS. GAs are population-based and stochastic search techniques in order to model the natural evolution. These algorithms are powerful in exploration of the search space, but their exploitation of found solutions are weak. In order to tackle this problem, a VNS metaheuristic is integrated with GA, which has a desirable exploitation feature. VNS is a single-point metaheuristic designed to search for an optimal point by swapping different neighborhood structures around a single solution.

This algorithm starts by initializing a population of randomly generated individuals. Each individual is an encoded version of a candidate schedule. Afterward, it generates new solutions by applying a crossover operator on selected individuals from the current generation as parents. Then, the mutation operator is applied on these newly created solutions with certain probabilities. To evaluate the completion time of each solution, the existing precedence constraints among scheduled tasks create challenges that should be addressed. Therefore, an upward-rank heuristic has been utilized to prioritize the assignments of tasks in TPG to processors [20]. The rank calculation of each task is defined as follow:

$$rank(T_i) = \begin{cases} W_{ij} & \text{if } succ(T_i) = \varnothing \\ W_{ij} + \max\limits_{T_k \in succ(T_i)} & \\ \{rc(T_i, T_k) + rank(T_k)\} & \text{if } succ(T_i) \neq \varnothing \end{cases} \tag{4}$$

where $succ(T_i)$ is a list containing the successors of task T_i, and $rc(T_i, T_k)$ represents the communication cost between task T_i and task T_k. If both tasks are scheduled on the same processor (P_j), this value is equal to 0; otherwise, it is equal to C_{ik}. In the evaluation function, in order to calculate the AFT of all tasks, the assignment order of each task to a processor in a schedule is chosen based on the rank of the task. Thereafter, the *makespan* of each solution is evaluated based on Sect 3.1 as its fitness value. After offspring production, only a subset of the population is selected for use by the VNS procedure because the VNS operation is time consuming . The size of this sub population is equal to the sampling rate parameter of the whole population. If an offspring's *makespan* is improved during the VNS process, it will be replaced. At the final stage, the algorithm combines the offspring with the current population, sorts them in an increasing order, and selects the first set of solutions, which are equal to the population size, to create the next population. This process continues until a desired termination criterion is satisfied.

3.3 GPU Programming

GPUs are processors with hundreds of processing cores which provide high throughput and high memory-bandwidth. The CUDA programming model [12] is developed by NVIDIA in order to run sequential programs on GPUs. The basic component of the CUDA programming model is a *thread.* A group of these basic elements creates a *Thread Block* (TB). Threads within the same TB can cooperate with each other to execute a piece of code on different data. There is a well-defined memory hierarchy in CUDA. Threads within a thread block can share data among themselves using a shared memory. TBs within a grid can also share data through a memory space called global memory. There are also data caches in the Fermi architecture to reduce the latency of accesses to the global memory [10]. In the following, we propose our parallel memetic algorithm based on the CUDA programming model, focusing both on design and implementation aspects.

4. PROPOSED PARALLEL ALGORITHM

The execution time of MAs is a challenging barrier to leveraging them in practical applications. As they have iterative identical operations on different individuals (that can be considered as different data), the realization of a parallel approach seems promising in achieving real-time solutions. In addition, the execution flow of these algorithms fits *Single Instruction Multiple Data* (SIMD) processors like GPUs, which are capable of executing the same instruction on different data.

Figure 3 illustrates the general scheme of the proposed parallel algorithm for MAs and the interactions between a CPU and a GPU. The main reasons for presenting this algorithm include: first, to accelerate the search process of MAs while having no adverse impact on the semantics of the algorithm itself; second, to take best advantage of these processors without becoming so bogged down in communication time between CPU and GPU by executing the whole process of evolution and local search on the GPU; third, to remove the restriction on the problem size; finally, to utilize efficiently the GPU's high-bandwidth memory subsystem and its computation power. As the heart of the selected MA is a GA, we describe the proposed parallel GA algorithm step by step in the following section. Afterward, the parallel VNS approach will be presented.

4.1 Parallel Genetic Algorithms

In a rough classification, parallel GAs can be classified into *master-slave*, *island*, and *cellular* models [4]. These models have different approaches in running GAs in order to accustom to different parallel executing platforms. We describe these models in the following.

The first model in this category is the master-slave model. The main objective of this approach concerns parallelizing the most time consuming operation, which is the evaluation of individuals. A master processor stores the population, executes the evolution engine, and divides the evaluation of individuals among several slave processors. However, the data transfer between CPU and GPU is the main bottleneck, which makes this model an unattractive choice. The second model is the island model in which a population is divided into multiple sub-populations (islands). The islands are evolved mostly isolated from each other and occasionally exchange individuals, which is called *migration.* This model is usually implemented on distributed systems. The main idea behind this model is to improve the search diversity along with the acceleration of the GA process. Nevertheless, the main bottleneck in front of porting this model to GPUs is the lack of sufficient shared memory for storing islands, especially in combinatorial optimization problems that need a large amount of memory [6]. Moreover, as local search is often used in MA to overcome the shortcoming of GA exploitation, improving the solution quality by applying this model seems to be unnecessary. The cellular model is the last parallel approach which considers a large number of small populations. Each of these small populations is assigned to a processor, and the crossover operator is restricted to use the neighborhood individuals. This model is a natural fit for massively parallel computers. However, this model is not naturally suitable to be used in the proposed parallel MA. To overcome the above-mentioned limitations as much as possible, a single population GA is selected to implement the proposed parallel algorithm. We have also decided to fully offload the GA operations onto the GPU side.

4.1.1 Problem Encoding

In this section, we elaborate the problem representation in our parallel algorithm. A solution is encoded in an array called *schedule list.* As shown in Figure 4 the cell index represents task number and its value determines the corresponding processor number. Along with this representation, all solutions in a population are also stored consecutively in the GPU's global memory, as shown in Figure 4. This strategy provides an efficient usage of the GPU's memory bandwidth through allowing coalesced accesses to the global memory. In coalesced accesses, consecutive memory blocks are fetched using one memory transaction. Furthermore, as the TPG is a sparse matrix, we propose a novel representation for storing TPG in the GPU's memory as efficiently as possible. In this encoding, the successors of all tasks are stored consecutively into the *successor array.* Then, the total number of successors of each task, equal to the out-degree of the task, is stored in *count array.* This allows retrieving the appropriate successors of a task. The same representation is used to store predecessors. Figure 4 shows this representation for TPG in Figure 1. This method enables storing large TPGs along with W and C matrices inside Fermi's small caches which consequently reduces the retrieval time of this read-only data.

4.1.2 Initialization

After allocating the required memory on the GPU and transferring the prerequisite data, a kernel initializes the population of the first generation. The *Generate Population* kernel is run with $PopSize \times Tasks$ number of threads. Each thread of a TB initializes one cell in a schedule list by assigning a randomly generated processor number to a task with a uniform distribution. A device RNG function from *curand* [11] library produces the required random numbers. This library provides the host and device sides with high quality random number generators.

4.1.3 Evaluation

Evaluation of candidate solutions is one of the most time consuming operations and requires to be parallelized. First, the *Task Priority* kernel calculating the ranks based on Section 3.2 is executed with $PopSize$ number of threads. Here, each thread in a TB is responsible for calculation of all tasks in a schedule list, and precedence constraints among tasks avoids any further parallelism. As the rank calculation is a recursive operation, a kernel is implemented to find the ranks of the tasks of TPG in a bottom-up manner. Next, the *Evaluation* kernel is run with $PopSize$ number of threads. A thread in a TB calculates the *makespan* of each solution of the population. This approach results in better GPU utilization by enlarging the population size.

4.1.4 Selection, Crossover and Mutation

In our implementation, genetic operators are also executed on the GPU to improve the performance by removing data transfers between the two processors (i.e., CPU and GPU). In addition, the *selection* and *crossover* operations could be integrated into a single GPU kernel. This integration decreases the overhead of costly global memory accesses because the selected solutions as parents are stored in the

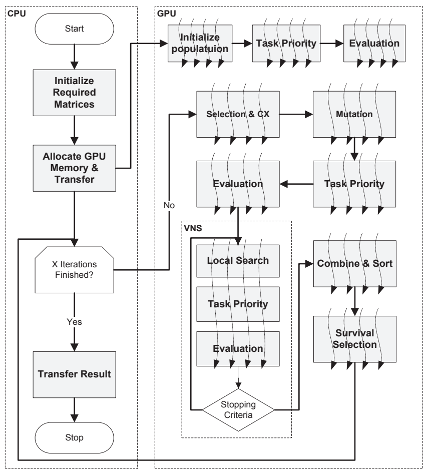

Figure 3: Parallel memetic algorithm

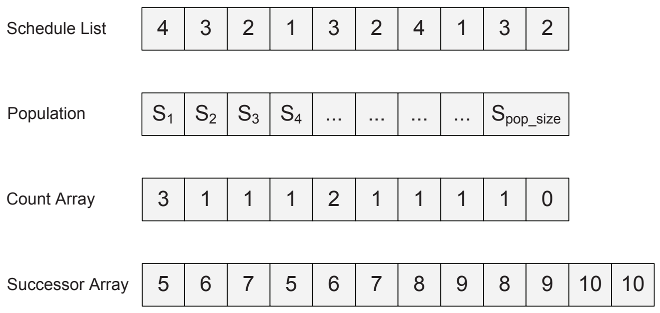

Figure 4: Problem representation for GPU

GPU's *shared memory*. The *Selection and CX* kernel is run with *PopSize* number of threads. Each thread in TB selects one parent from the population and copies it into the *shared memory*. Having selected all required parents, the thread selects two of them and applies crossover between them to generate a new solution. Finally, all the new generated solutions will be transferred into the global memory. After that, a mutation operator with a predefined probability (i.e., *mutation rate*) is applied on these solutions. The mutation operator assigns a random processor to a task. The expected number of mutations per solution is equal to the *mutation rate* multiplied by the *Tasks*. The *Mutation* kernel is run with *PopSize* number of threads. Each thread in a TB is responsible for applying the mutation operator on one solution. At this stage, the new solutions are ready for the next operation in the MA process which is VNS. This operation is explained in Section 4.2.

4.1.5 Survival Selection

The last stage of GA is the replacement of the current population with the new solutions. In this phase, the current population and new solutions are merged and sorted based on their *makespans*. For solving many discrete optimization problems small-size populations are sufficient; on the other hand, increasing the population size results in a better solution quality for complex ones. In our experiments, we have learned that the time elapsed in the sorting section, which is proportional to the population size, has a profound impact on the whole execution time of the parallel algorithm. Therefore, the selection of an efficient GPU-based sorting algorithm is vital to improve the execution time. Consequently, we use an adaptive sorting strategy to choose sorting mechanism based on the input population size. In this strategy, a single-TB bitonic sort algorithm is selected when the population size is less than a specific threshold, which is half of the TB size. Moreover, a multi-TB merge sort algorithm is leveraged to address the large populations. This adaptive approach makes it possible to achieve noticeable speedups even for small-size populations.

4.2 Parallel Variable Neighborhood Search

One of the most time consuming sections of the hybrid algorithm is VNS which makes the parallel execution of this part important to reaching a high performance algorithm. The kernel's algorithm is shown in Algorithm 1. The *VNS* kernel is run with *PopSize * sampling rate* number of threads. The *sampling rate* parameter is a criterion for selecting candidate solutions for the VNS operation. As shown in Algorithm 1, the VNS kernel has a complex control flow; therefore, this kernel is run by many TBs that each of them has few number of threads. As a result of this approach, we reduced adverse effects of thread divergence which decreases the performance of the parallel algorithms on GPUs. In the *VNS* kernel a set of neighborhood structures is chosen in the initialization step. Here, we have two neighborhood structures. Then, a certain number of solutions from the newly generated ones in the global memory are randomly selected and transferred to the *shared memory*, and the index of the neighborhood structure (k) is set. Each thread in a TB executes four device kernels: *Shaking*, *Local Search*, *Task Priority*, and *Evaluation*. These kernels are executed iteratively on the GPU for a single solution which is resided in the *shared memory* until a termination condition is met. This explains why we call this approach *massively parallel VNS* (MPVNS). In the *Shaking* device kernel, a new solution $x^{'}$ is randomly generated based on the selected neighborhood of the x solution. In literature, different strategies exist for neighborhood structures of the task scheduling problem, from fully randomized [2] to problem-specific [18]. We use two neighborhood structures which are proposed in [18]. One of the neighborhood structures is used to improve the load balancing of processors. First, the computational load of each processor should be calculated:

$$comp(P_j) = \sum_{T_i \in run(P_j)} W_{ij} \tag{5}$$

where $run(P_j)$ denotes the set of tasks that have been scheduled on processor P_j. Afterward, the processor with maximum computational load is found ($P_{maxcomp}$). Then, a task is selected randomly from the solution $x^{'}$ which is scheduled on this processor. Finally, the task is assigned to a new randomly generated processor number other than $P_{maxcomp}$. The goal of the second neighborhood structure is to reduce communication costs. First, the communication costs for all

Algorithm 1 Massively Parallel Variable Neighborhood Search (MPVNS)

```
Select a set of neighborhood structures $N_k$, for $k = 1, \ldots, k_{max}$
Move solutions from global memory to shared memory, do ($x \leftarrow x_{global}$)
for all threads in a thread block do
    repeat
        function SHAKING($x^{'}$,x,k) Generate a solution $x^{'}$ at random from the $k^{th}$ neighborhood of $x$
        end function
        function LOCAL SEARCH($x^{''}$,$x^{'}$,k) Apply local search method with $x^{'}$ as initial point; denote $x^{"}$ the so obtained local optimum
        end function
        function TASK PRIORITY($x^{''}$) Apply upward-ranking heuristic on $x^{''}$ solution.
        end function
        function EVALUATION($x^{''}$) Calculate the makespan of the $x^{''}$ solution.
        end function
        if $x^{"}$ makespan is less than $x$ makespan then
            $x \leftarrow x^{"}$
            $k \leftarrow 1$
        else
            $k \leftarrow k + 1$
        end if
    until $k \leq k_{max}$
end for
$x_{global} \leftarrow x$
```

processors are calculated as follow:

$$comm(P_j) = \max_{T_i \in run(P_j)} AFT(T_i) - comp(P_j) \tag{6}$$

where the $AFT(T_i)$ denotes when the task T_i actually finishes its execution on processor P_j. After that, a processor with a maximum communication cost is selected ($P_{maxcomm}$). Then, the predecessors of all assigned tasks on this processor ($T_i \in run(P_j)$ which are scheduled on other processors are found. Finally, a task randomly is chosen from this predecessors set and assigned to the $P_{maxcomm}$ processor.

The *Local Search* device kernel starts to proceed by using solution $x^{'}$ as its starting point and generating a new solution $x^{''}$. This solution is used by *Task Priority* and *Evaluation* device kernels in order to calculate its *makespan*. Finally, if the *makespan* of solution $x^{"}$ is less than that of x, it replaces x in *shared memory* and the algorithm continues with current neighborhood structure. Otherwise, k is incremented by one for using the next neighborhood structure. Due to existing dependencies between consequent phases of VNS there are no more rooms for extra parallelisms. However, this approach will be efficient on GPUs because of using the *shared memory* for storing solutions, intermediate results, and caches for prerequisite data such as W and C matrices. The final solution and its *makespan* will be stored back to the global memory for next operations.

5. EXPERIMENTAL WORK

In this section, we present a comparative evaluation between sequential and parallel versions of GVNS execution times. To show the effectiveness of our parallel algorithm, Gauss-Jordan elimination (GJ) with 15 and 21 tasks, Fast Fourier Transformation (FFT) with 15 tasks, and Laplace equation solver (L) with 16 and 36 tasks are chosen as the benchmarks [18]. Figure 1 shows a TPG of GJ as a well-known parallel application for 10 tasks. We randomly generate 10 computation cost matrices for each benchmark graph. The values of matrix entries are driven by Poisson distribution with an average computation time of 40 [20]. The value of all required parameters is shown in Table 2. The communication to computation ratio (CCR) is set to 0.25 for all tests, and the data transfer time between dependent tasks on different processors are the same, which is equal to $40 * 0.25 = 10$ in our experiments. The sequential algorithm has been executed on a system which is equipped by an Intel Xeon processor with 8 cores and 8 GBs of RAM. The GPU card used in our experiments is a GTX480 with 480 CUDA cores.

Table 2: Parameters

Parameter Name	Value
Generations Count	100
Population Size (PopSize)	128-8192
Crossover Rate	1
Mutation Rate	0.1
Sampling Rate	0.1
TPG Kind	GJ,FFT,L
Tasks	(21,45)(15)(16,36)
Processors Count	8

5.1 Performance Evaluation Metrics

The metrics to measure performance of parallel metaheuristics involve both computational effort and solution quality. The first and the most important metric is the *relative computational speedup* ($speedup_c$). The relative speedup is the ratio of serial execution time (T_s) to parallel execution time (T_p) of a program [1]. We have tried to satisfy two conditions in order to make fair comparisons: first, the serial code should be the most efficient one; second, the parallel time should include any overhead such as data transfers since the objective of parallelism is to reduce the real time of execution. As the execution times of non-deterministic algorithms vary for different runs, the average of execution times of 50 runs for each of 10 different configurations are considered as the expected execution time of the algorithm. The relative computational speedup is defined as follow:

$$speedup_c = \frac{E\left[T_s\right]}{E\left[T_p\right]} \tag{7}$$

After measuring the efficiency of our approach; the last important metric to evaluate is the quality of the obtained solutions. As explained, the goal of the algorithm is to find a schedule of tasks that its parallel execution time ($makespan_p$) is less than the serial execution time of all tasks' ($makespan_s$). By serial execution we mean to execute all tasks on a single processor instead of multiple processors. The ratio of these two values is the quality metric of the algorithm and is denoted by $speedup_q$:

$$speedup_q = \frac{makespan_s}{makespan_p} \tag{8}$$

As the optimum *makespan*$_p$ is unknown, the mean of the *makespans* of the best solutions over all predefined number of algorithm runs is measured. The *makespan*$_s$ is also computed as:

$$makespan_s = \min_{P_j \in P} \left\{ \sum_{T_i \in V} W_{ij} \right\} \quad (9)$$

In the context of discrete optimization, the overall goal is not full GPU utilization but to develop powerful parallel algorithms to find better solutions in a shorter time [14]. As the scheduling problem is a combinatorial optimization problem, the *speedup*$_q$ is increasing until the algorithm reaches a convergence point and after that no further improvement is observed and higher GPU utilization is not useful. Therefore, it would not be fair to report the *speedup*$_c$ of the parallel algorithm below and above of this certain point. Hence, we define *effective speedup*$_c$ as the measured computational speedup when the algorithm achieves the maximum *speedup*$_q$ for the first time with no further improvement after that.

5.2 Results and Discussion

In this part, we provide our experimental results for the sequential and parallel implementations. For the sequential algorithm, as we expect the increase in number of iterations results in better solution qualities but causes longer execution times as illustrated in Figure 5(a). Therefore, finding a solution with less number of iterations with a reasonable quality is a desirable goal. However, enlarging population size provides the same or better quality solutions while keeping a small number of iterations with much less execution time, as shown in Figure 5(b). Therefore, we choose the latter paradigm to achieve better qualities for 100 iterations in our parallel experiments.

As shown in Table 3, increasing the population size improves the solution quality of the algorithm. The maximum *speedup*$_c$ for GJ-21, GJ-45, FFT-15, L-16, and L-36 are 343, 696, 254, 272, and 568 respectively for population size of 512. However, this speedup may not be the *effective speedup*$_c$ of the parallel algorithm. Hence, we continue the execution of the algorithm by enlarging the population size in order to achieve the maximum *speedup*$_q$. As explained earlier, we have employed a multi-TB merge sort algorithm when the population size is more than 512. By changing the sorting algorithm the *speedup*$_c$ of the algorithm suddenly falls to much lower values, as shown in Table 3. However, when the population size becomes sufficiently large the overhead of sorting algorithm is amortized and large computational speedups are observed. The *effective speedup*$_c$ for GJ-21, GJ-45, FFT-15, L-16, and L-36 are 181, 40, 220, 13, and 42 for population sizes of 8192, 8192, 8192, 1024, and 4096 respectively. It is obvious that the *effective speedup*$_c$ depends on the problem size and complexity.

Table 3 shows speedups of 568 for L-36 and 696 for GJ-45 in population size of 512. The reason for this superlinear speedup is the fact that sequential algorithm requires more memory access in comparison to parallel one. As mentioned, we use a single-TB bitonic sort to remove memory accesses to global memory for population sizes of less than 1024. By increasing the size of the population more memory accesses are required; therefore, the execution times of the sequential algorithms increase while the execution times of the parallel implementations remain unchanged due to usage of single-TB sort.

Table 3: The average execution time and solution quality for parallel and sequential algorithms on different benchmarks for various population sizes.

		Avg. Time (Sec.)			
TPG	*Pop.*	*Par.*	*Seq.*	*Speedup*$_q$	*Speedup*$_c$
GJ-21	128	0.003	0.183	3.30	61
	256	0.003	0.415	3.35	138
	512	0.003	1.028	3.38	343
	1024	0.238	2.832	3.39	12
	2048	0.289	8.758	3.39	30
	4096	0.384	29.899	3.39	78
	8192	0.603	109.268	3.40	181
GJ-45	128	0.003	0.447	4.32	149
	256	0.003	0.948	4.40	316
	512	0.003	2.088	4.47	696
	1024	0.798	4.993	4.53	6
	2048	0.945	13.243	4.58	14
	4096	1.183	39.558	4.61	33
	8192	3.247	131.446	4.64	40
FFT-15	128	0.003	0.118	2.77	39
	256	0.003	0.287	2.78	96
	512	0.003	0.763	2.79	254
	1024	0.166	2.268	2.80	14
	2048	0.209	7.539	2.80	36
	4096	0.286	27.059	2.80	95
	8192	0.464	102.022	2.81	220
L-16	128	0.003	0.131	2.20	44
	256	0.003	0.306	2.20	102
	512	0.003	0.815	2.20	272
	1024	0.178	2.379	2.21	13
	2048	0.226	7.763	2.21	34
	4096	0.307	27.49	2.21	90
	8192	0.495	102.879	2.21	208
L-36	128	0.003	0.335	3.03	110
	256	0.003	0.728	3.05	233
	512	0.003	1.646	3.08	568
	1024	0.564	4.089	3.09	7
	2048	0.673	11.444	3.10	17
	4096	0.844	35.85	3.12	42
	8192	1.883	123.422	3.12	66

Sampling rate is one of the most important parameters in the VNS algorithm. For example, higher sample rates are appropriate for complex search spaces to reach better solution quality. On the other hand, higher sample rates increase the execution time dramatically [18]. However, as shown in Figure 5(c), increasing sampling rate for improving the solution quality has a minimum impact on the execution time of the proposed algorithm because our parallel algorithm is highly scalable with regards to population size.

6. CONCLUSION

In this work, a parallel memetic algorithm has been proposed and implemented on GPUs. The proposed algorithm has been used to solve a task scheduling problem for heterogeneous multiprocessor systems as a case study. Furthermore, an especial encoding method has been designed for this case study to utilize the GPU's memory efficiently. In addition, we have shown that different sorting strategies

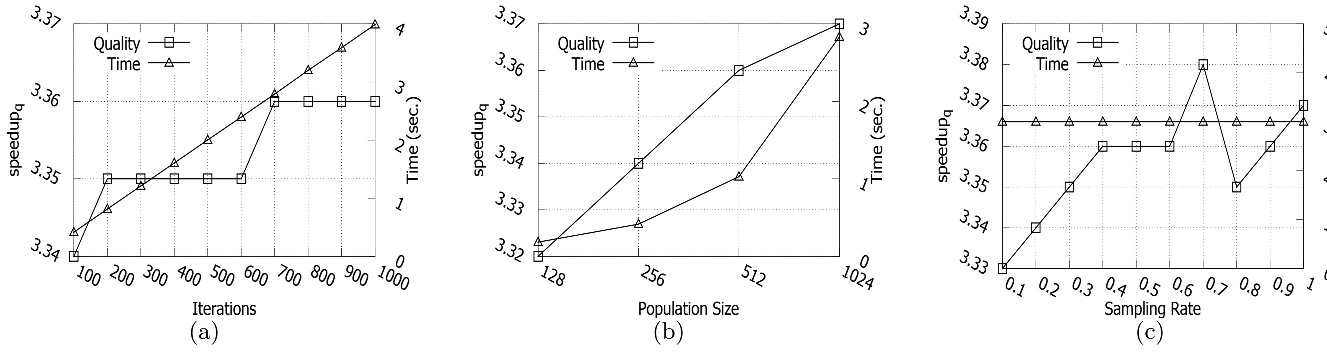

Figure 5: The effect of parameters on time and quality for GJ-21 : (a)different iterations for population of 256 individuals; (b)various population sizes for 100 iterations; (c)different sampling rates in parallel algorithm for population of 256 individuals

could be beneficial in terms of running times for various population sizes. As a result, an adaptive sorting policy has been proposed in our parallel algorithm. By using this method, we have achieved up to 696 computational speedup even for small populations. In the future, we plan to extend this algorithm into a general framework that can be leveraged by users in parallelizing different kinds of hybrid metaheuristics on GPUs with minimum effort.

6.1 Acknowledgments

The authors would like to thank Ali Hamze for his helpful comments on this work. This work was supported in part by Institute for Research in Fundamental Sciences (IPM) under grant number CS1391-4-03.

7. REFERENCES

[1] R. S. Barr and B. L. Hickman. Reporting Computational Experiments with Parallel Algorithms: Issues, Measures, and Experts' Opinions. *INFORMS Journal on Computing*, 5(1):2–18, Jan. 1993.

[2] T. Davidovic, P. Hansen, and N. Mladenovic. Permutation-based Genetic, Tabu and Variable Neighborhood Search Heuristics for Multiprocessor Scheduling with Communication Delays. *Asia-Pacific Journal of Operational Research*, 22(03):297–326, 2005.

[3] H. El-Rewini, T. G. Lewis, and H. H. Ali. *Task Scheduling in Parallel and Distributed Systems.* Prentice Hall, 1994.

[4] R. L. Haupt and S. E. Haupt. *Practical Genetic Algorithms.* John Wiley & Sons, 2 edition, 2004.

[5] F. Krüger, O. Maitre, S. Jiménez, L. Baumes, and P. Collet. *Applications of Evolutionary Computation*, volume 6024 of *Lecture Notes in Computer Science.* Springer Berlin Heidelberg, Berlin, Heidelberg, Apr. 2010.

[6] T. V. Luong, N. Melab, and E.-G. Talbi. GPU-based island model for evolutionary algorithms. In *GECCO'10*, page 1089, New York, New York, USA, 2010. ACM Press.

[7] T. V. Luong, N. Melab, and E.-G. Talbi. Parallel hybrid evolutionary algorithms on GPU. In *IEEE CEC*, pages 18–23. IEEE, 2010.

[8] F. Neri, C. Cotta, and P. E. Moscato. *Handbook of Memetic Algorithms, Studies in Computational Intelligence, Vol. 379.* Springer, 2011.

[9] S. Nesmachnow and M. Canabé. GPU implementations of scheduling heuristics for heterogeneous computing environments. In *Proceedings of the XVII Congreso Argentino de Ciencias de la Computación,*, pages 1563–1570, 2011.

[10] NVIDA. Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Fermi, 2009.

[11] NVIDIA. CUDA Toolkit 4 . 1 CURAND Guide, 2012.

[12] NVIDIA. NVIDIA CUDA C Programming Guide, 2012.

[13] F. Pinel, B. Dorronsoro, and P. Bouvry. Solving very large instances of the scheduling of independent tasks problem on the GPU. *J. Parallel Distrib. Comput.*, 73(1):101–110, Jan. 2013.

[14] C. Schulz. Efficient local search on the GPU - Investigations on the vehicle routing problem. *J. Parallel Distrib. Comput.*, 73(1):14–31, 2013.

[15] H. Topcuoglu, S. Hariri, and M.-Y. W. M.-Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, 2002.

[16] A. Tumeo, C. Pilato, F. Ferrandi, D. Sciuto, and P. L. Lanzi. Ant colony optimization for mapping and scheduling in heterogeneous multiprocessor systems. In *2008 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 142–149. IEEE, July 2008.

[17] J. D. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384–393, 1975.

[18] Y. Wen, H. Xu, and J. Yang. A heuristic-based hybrid genetic-variable neighborhood search algorithm for task scheduling in heterogeneous multiprocessor system. *Information Sciences*, 181(3):567–581, 2011.

[19] A. S. Wu, H. Yu, S. Jin, K. C. Lin, and G. Schiavone. An incremental genetic algorithm approach to multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 15(9):824–834, 2004.

[20] H. Yu. Optimizing task schedules using an artificial immune system approach. In *GECCO'08*, page 151. ACM Press, 2008.