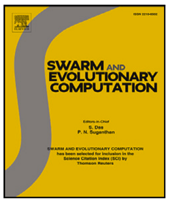
Swarm and Evolutionary Computation 98 (2025) 102110

Contents lists available at ScienceDirect

Swarm and Evolutionary Computation

journal homepage: www.elsevier.com/locate/swevo



Parallel genetic algorithms on hybrid servers: Design, implementation, and optimization for performance and energy

Amr Abdelhafez [a,b,c,*], Ravi Reddy Manumachu [b], Alexey Lastovetsky [b]

[a] Department of Computing, South East Technological University, Carlow Campus, Ireland
[b] School of Computer Science, University College Dublin, Belfield, Dublin 4, Ireland
[c] FCI, Assiut University, Egypt

ARTICLE INFO

Keywords:
Genetic algorithms
Parallel computing
Hybrid server
Energy consumption
Optimization
Heterogeneous computing

ABSTRACT

Parallel Genetic Algorithms (PGAs) have been widely applied to accelerate solutions for real-world problems such as energy optimization in building constructions, data preprocessing and model selection steps in data mining, real-time control of multilevel inverters in electronics, land-use planning, nanoscience, optimal power flow in power systems, and road traffic management.

The state-of-the-art research proposes PGAs optimized solely for performance and for solving optimization problems on a multicore CPU, GPU, or clusters of multicore CPUs. However, no research has analyzed PGAs for heterogeneous hybrid platforms comprising multicore CPUs and multiple accelerators that utilize all computing devices in parallel. Furthermore, no definitive comparative research comprehensively investigates the energy consumption of PGAs in hybrid systems versus multicore CPUs or GPUs.

We address the above gaps in the prior art in this work. First, we present a novel parallelization approach (HPIGA) tailored for heterogeneous hybrid platforms, featuring a portable implementation that utilizes all available computational devices, including multicore CPUs and GPUs. We conduct a comprehensive investigation into the performance and energy profiles of this approach. We compare it with three other traditional parallel approaches across a range of dimensions, varying from 100 dimensions and up to 5000 dimensions. The results showed HPIGA's competitive energy consumption behavior and promising performance compared to other traditional approaches under the study.

Moreover, we formulate a bi-objective optimization problem of a PGA employing a parallel island model and executing on a hybrid server comprising p compute devices. The problem has two objectives: performance and energy. The decision variable used in our bi-objective optimization problem is workload distribution, which is proportional to the number of islands. We study the efficacy of our proposed PGA on a hybrid server platform with an Intel Icelake multicore CPU and two Nvidia A40 GPUs, analyzing execution time and dynamic energy profiles under two power governors. The resulting Pareto front graphs provide valuable insights, serving as crucial benchmarks for the future development and use of efficient, energy-aware optimization techniques across diverse computational devices.

1. Introduction

Genetic algorithms (GAs) are metaheuristics inspired by the process of natural selection employed to solve optimization problems where the objective functions are highly non-linear, discontinuous, non-differentiable and therefore lack analytical expressions amenable to using traditional calculus methods [1–3]. Furthermore, GAs are predominantly used to solve multi-objective optimization problems where the objective functions possess conflicting goals and display ill-defined or undesirable calculus properties, rendering finding an efficient exact algorithm to solve the problems intractable [4,5].

GAs with long execution times due to large problem datasets, high problem dimensionality, complex objective functions with time-consuming function evaluation, and customized genetic operators necessitated the development of Parallel Genetic Algorithms (PGAs) to reduce the execution times. PGAs have been widely used to accelerate solutions to real-world problems such as energy optimization in building constructions [6], data preprocessing and model selection steps in data mining [7], real-time control of multilevel inverters in electronics [8], land-use planning [9], nanoscience [10], optimal power flow in power systems [11], and road traffic management [12].

* Corresponding author at: Department of Computing, South East Technological University, Carlow Campus, Ireland.
E-mail addresses: amr.abdelhafez@setu.ie (A. Abdelhafez), ravi.manumachu@ucd.ie (R.R. Manumachu), alexey.lastovetsky@ucd.ie (A. Lastovetsky).

https://doi.org/10.1016/j.swevo.2025.102110
Received 11 December 2024; Received in revised form 25 June 2025; Accepted 26 July 2025
Available online 16 August 2025

One of the critical components of a PGA implementation is the parallel hardware architecture or platform, which is essential for optimizing the PGA for the specific platform. The parallel hardware architectures employed in the PGA implementations include multicore CPUs [9,13], clusters of multicore CPUs [7,11], Graphics Processing Units (GPUs) [8,14], FPGAs [15,16] and Clouds [17]. The software packages popularly used for PGA implementations on multicore CPUs are Parallel Computing Toolbox provided in MATLAB and Java threads (Java concurrency package). PGA implementations on clusters commonly employ a message-passing interface (MPI) [18]. PGA implementations on GPUs are mostly written in CUDA, which is not portable to other vendors and can be challenging to implement or extend.

The word *hybrid* in the state-of-the-art PGAs is used in two contexts. First, it refers to the type of algorithm and second, the hardware platform employed. Several hybrid PGAs have been proposed in the literature in the first context, which combine two different PGA models or integrate the genetic algorithm (GA) with another heuristic method, such as simulated annealing (SA), to solve optimization problems. In this work, we follow the second context and reserve the word *hybrid* to refer to the PGA implementation executing on a heterogeneous server platform comprising multicore CPUs and one or more accelerators and employing all the computing devices in parallel. While the research works surveyed above focus on improving the performance of PGAs, there is a lack of such research devoted to analyzing and minimizing the energy consumption of PGAs. To summarize, the state-of-the-art research works above propose PGAs optimized only for performance and running on either a single computing device (multicore CPU or GPU or FPGA) or a homogeneous cluster of multicore CPUs. There is no research on PGAs running on heterogeneous hybrid platforms comprising multicore CPUs and multiple GPU accelerators that employ all computing devices in parallel. Furthermore, no previous research comprehensively investigates the energy consumption of PGAs in hybrid heterogeneous systems, comparing them to their traditional parallel counterparts.

The primary objective of this work is to address the aforementioned gaps in the existing literature by developing HPIGA, a parallel genetic algorithm designed for execution on hybrid heterogeneous platforms. We develop a portable HPIGA implementation for one multicore CPU, and $p-1$ accelerators. The implementation is based on OpenH [19], a programming model and API for developing portable parallel programs on heterogeneous hybrid servers composed of a multicore CPU and one or more different accelerators. We compare the energy consumption and performance of this approach to three other parallel configurations: a multicore-CPU model, a single GPU model, and a functional model. The multicore-CPU and single GPU models represent the classical parallel approaches commonly used in the literature, while the functional model introduces an innovative parallel approach that explores the computational and intercommunication collaboration between devices. Detailed explanations of these parallel models are provided in Section 6. The key objectives of this study are to:

- Developing a portable implementation of HPIGA across multicore CPUs and accelerators. Comparing the performance and energy consumption of HPIGA against traditional parallel models (multicore-CPUs, GPUs).
- Investigating trade-offs between performance and energy efficiency, with Pareto-optimal solutions for workload distribution across devices.

In summary, we experimentally analyze the performance and energy consumption of the proposed PGA on a heterogeneous hybrid server consisting of an Intel Icelake multicore CPU and two Nvidia A40 GPU accelerators. Specifically, we present and discuss the Pareto-optimal solutions from the bi-objective optimization of the PGA, optimizing for performance and energy, with the distribution of islands between the devices as the decision variable.

The prime contributions of this work are:

- A novel algorithm design tailored for efficient execution on hybrid systems, employing all computing devices through workload distribution among them.
- Presenting a portable implementation of our proposed PGA, optimizing parallel GAs for performance and energy on a heterogeneous hybrid server.
- Analyzing the performance and energy profiles of our proposed PGA across different devices on a heterogeneous hybrid platform, comprising a multicore CPU and two GPU accelerators. Additionally, we present Pareto fronts for optimizing PGAs on hybrid servers to improve performance and energy efficiency.

The rest of the paper is organized as follows. Section 2 presents the related work. In Section 3, we briefly outline the origins and principles of GAs and PGAs. Further, we cover the design and implementation of hybrid heterogeneous PGAs in Section 5. Then we detail our experimental framework in Section 6. Section 7 provides the experimental results for our study. we summarize the findings and outline future work in Section 8.

2. Related work

Over the past two decades, numerous studies have addressed analyzing the parallel performance of metaheuristic algorithms. In this section, we review the efforts made to analyze the performance and energy consumption of Metaheuristics and GA on multicore CPUs and accelerators.

2.1. Parallel metaheuristics on multicore CPUs

The efforts to analyze the performance of parallel metaheuristics on processors have been quite extensive over the past two decades. However, these studies typically focused on the algorithm's performance and lacked a comprehensive analysis of energy consumption. Here, we will explore certain initiatives conducted across both past and contemporary times.

One of the early efforts to analyze the performance of parallel metaheuristics was presented in [20]. In the article, the author explores the application of the parallel tabu search algorithm to address large traveling salesman problems. Additionally, the article describes its implementation on a transputer network, highlighting the efficiency of the parallel algorithm through numerical results and speedup assessments. Another early parallel approach to investigate the performance evaluation of a parallel tabu search algorithm was presented in [21]. The authors assessed the achieved makespan reduction of various parallel applications. Their conclusions suggest that, in numerous cases, the parallel tabu search algorithm yields significantly better solutions compared to the greedy algorithm.

Over the past few years, efforts and studies have been presented to study the performance of PGAs. A study of the performance of parallel GA models over multicore CPUs was presented in [22]. They studied the performance of three parallel models (master–slave model, synchronous, and asynchronous distributed GAs). Their findings illustrated the performance characteristics of these models, although the paper did not delve into their energy consumption profiles. In [23], the authors presented the first approach to study the performance and energy consumption of sequential and parallel distributed GAs with an examination of the consumption of the algorithm components. The results reveal the energy profile of sequential and distributed GAs over multi-cores. A recent approach to evaluate the performance of distributed PGAs on a cluster of multicore CPU processors was presented in [24]. The authors of that article aim to study the numerical and computational behavior of algorithms by proposing a mathematical model representing the observed performance curves. Their research follows the typical path of investigating the performance of PGAs, with no mention of the energy profiles. A comparative study of the performance and energy consumption of several metaheuristics (including GA) over multi-cores was presented in [25]. In their research, they conduct two comprehensive investigations into the solution quality, energy consumption, and execution time of three distinct metaheuristics and their corresponding distributed versions. The primary objective of their studies is to assess the effectiveness of parallel execution of these metaheuristics within emerging computing environments. For more research on the performance of parallel metaheuristics over multi-cores and clusters, we refer the reader to articles [26–32]. However, none of the studies mentioned in this section investigate the performance and energy consumption of GPUs or hybrid systems.

2.2. Parallel metaheuristics on GPUs

GPUs are emerging as powerful computation devices, expanding beyond their traditional graphical capabilities to solve complex parallel problems. Over the past decade, a handful of studies have delved into the potential of GPUs to enhance the performance of metaheuristics. Here, we will review key studies exploring this aspect.

One of the early attempts to study the performance of metaheuristics was presented in [33], where the authors presented two double-level parallel metaheuristic algorithms to solve the flexible job shop scheduling problem over GPU. The algorithms consist of two key modules: the machine selection module, which is executed sequentially, and the operation scheduling module, which operates in parallel over GPUs. The authors have not compared the results to the multicore counterparts or energy consumption evaluation. Another early effort to study the distributed tabu search metaheuristic using a multi-GPU cluster is presented in [34]. The article's authors proposed a hybrid parallelization approach for tabu search designed to solve the flexible job shop problem, where hybridization involves simultaneously examining multiple solutions from a neighborhood using several GPUs (multi-GPU). Also, the study does not include energy consumption or an evaluation of their proposed approach employing multicore CPUs.

Over the past decade, there has been intense competition in deploying search algorithms on GPUs. Authors of [35] introduced a local search strategy named Variable Neighborhood Descent (DVND), which was developed for CPU and multi-GPU environments. They introduced a neighborhood search strategy for the massive parallelism of GPUs to enhance local search (LS) procedures. Their study also does not include a comparison with standard algorithms or multicore CPU counterparts. Another approach to parallelize particle swarm optimization (PSO) algorithm was presented in [36]. The authors of that work proposed a GPU-PSO algorithm based on CUDA, utilizing a combination of coarse-grained and fine-grained parallelism for global efficiency. They designed a CUDA-based data structure and merged memory access mode to enhance data-parallel processing and access efficiency. Their experimental results highlight the algorithm's effectiveness in reducing solution times for high-dimensional, large-scale optimization problems. For further investigation into the performance of parallel metaheuristics across GPUs, readers are directed to [37–39].

In the context of genetic algorithms, a leading metaheuristic, substantial initiatives have been undertaken in recent years to implement it on GPUs. An effort to present a GPU-based PGA for solving the flow shop scheduling problem is presented in [40]. They propose an energy-aware dynamic flexible flow shop scheduling model that considers peak power values for GPUs. They introduce a priority-based hybrid PGA with a predictive reactive complete rescheduling strategy. Their method was designed for the NVIDIA CUDA software model. The numerical experiments presented there demonstrate a competitive performance achieved by their approach for GPUs. A recent method employing parallel Cellular GAs in identifying classification rules on GPUs is presented in [41]. In that study, a partitioning strategy was introduced for optimizing 3D parallel cellular GAs on a 2D processing array using Field Programmable Gate Arrays (FPGAs). That research addresses both combinatorial and continuous domain benchmark problems, focusing on optimizing objective function modules through trigonometric and arithmetic-tailored units. Their results demonstrate up to three and two orders of magnitude speed-up compared to CPU and parallel GPU implementations. For an additional investigation into the implementation of genetic algorithms on GPUs, we direct the reader to Refs. [42–44]. For a survey on the acceleration of genetic algorithms through GPU computing, we direct the reader to Ref. [45].

A recent effort to predict the energy–time behavior of applications using multi-population genetic algorithms is found in [46]. Their approach focused on dynamically distributing the evaluation of individuals among CPU–GPU devices in heterogeneous clusters, providing a more accurate prediction compared to traditional linear regression methods. The study demonstrated the effectiveness of their model in improving energy and time efficiency in high-performance computing systems. It also highlighted the importance of focusing on key parameters to further refine the model. However, their work did not include running or comparing genetic algorithms executed separately on individual devices (CPU and GPUs) with heterogeneous ones, which is a key aspect addressed in our study. Another recent study [47] focused on the parallel GPU acceleration of optimization algorithms for solving large-scale nonlinear equation systems. The authors proposed GPU-based implementations of several recent algorithms (such as Jaya, and a new MaGI algorithm) employing a unified and efficient parallelization strategy. The implementations, written in Julia, were tested on high-end and consumer-grade GPUs, demonstrating notable speedups, especially for high-dimensional problems. The analysis revealed significant performance gains, with the scalability and adaptability of their approach highlighted across various GPU architectures. While their work provides valuable insight into the GPU-based acceleration of emerging metaphorless algorithms, it primarily concentrates on the performance improvements achieved by GPU-based parallelizing these methods for numerical equation solving. Unlike our study, their research does not investigate or compare the behavior of genetic algorithms or other population-based methods distributed across heterogeneous computing devices. A recent study by Silva et al. (2025) [48] presented a GPU-accelerated implementation of a modified PSO algorithm to solve large-scale systems of nonlinear equations. Their work compared the performance of the GPU-parallelized PSO against both sequential and multithreaded CPU versions, demonstrating significant speedups. The study highlighted the scalability advantages of GPU-based parallelism, especially for high-dimensional problems, while also discussing the trade-offs between computational precision and performance. Additionally, the paper examined the effects of increasing CPU thread counts, identifying optimal threading for maximum efficiency. These findings underscore the potential of heterogeneous computing platforms to effectively accelerate population-based metaheuristics for complex optimization problems. For more recent studies that analyze GPU-based and heterogeneous execution behavior of search and evolutionary algorithms, we kindly refer the reader to [49–52].

The aforementioned works in this section highlight numerous attempts to implement metaheuristics on multi-cores and accelerators. However, these efforts have primarily focused on running metaheuristics on either the multicore CPU or the GPU independently. As we span across these studies, we observe that all these articles examined the performance of the algorithms without addressing/comparing their energy consumption when being run over CPUs and accelerators. To the best of our knowledge, no study has yet compared the energy consumption and performance of multicore CPUs, GPUs, and all computing devices within heterogeneous systems. Our research aims in this work is to investigate the simultaneous execution of metaheuristics on both CPUs and GPUs, comparing the results of this integrated approach with the outcomes from running on each device separately. Our study provides a comprehensive analysis of the performance and energy efficiency of combined CPU/accelerator execution versus single-device execution.

3. Background of genetic algorithms

In this section, we briefly delve into the origins of sequential and parallel GAs. We examine the key components of the Canonical GA and also outline various models commonly used for parallelizing GAs.

3.1. The genetic algorithm

The GA is a famous population-based metaheuristic that relies on searching the problem space with a population of randomly generated individuals [53]. Each individual (expected optimal solution) is represented by a chromosome and its fitness value. The chromosome consists of an array of genes, the size of which depends on the problem's dimension. The fitness function evaluates each individual's quality in the population to select the solutions for the next generation.

To generate new solutions for the subsequent generation, the GA considers applying genetic operators, typically involving crossover, mutation, and replacement. Selection in GAs involves choosing individuals from a population based on their fitness for reproduction. This iterative process continues until a stopping criterion is met, such as reaching the maximum number of fitness evaluations or obtaining a solution of satisfactory quality [23]. Algorithm 1 outlines the pseudocode for this panmictic algorithm.

Algorithm 1 The standard genetic algorithm.

```
1: Initialization. Randomly generate an initial population $P$.
2: Evaluation. Evaluate the individuals in $P$.
3: while not stop-condition do
4:     $P' \leftarrow$ Crossover ($P$)
5:     $P'' \leftarrow$ Mutation ($P'$)
6:     Fitness Function Evaluation ($P''$)
7:     $P \leftarrow$ Selection ($P''$)
8: end while
9: Output. Best Found Solution so far.
```

Genetic operators are employed to create new solutions from existing ones. The crossover (or recombination) operator merges two or more distinct solutions to generate new ones. The crossover operator is essential for inheriting traits from both parents to generate offspring. The mutation is a variation operator that produces a new solution by altering the genes of a distinct one. The mutation operator's key role is introducing genetic diversity within the population, thereby preventing the algorithm from converging to a local optimum [54]. Selection in GAs involves choosing individuals from a population based on their fitness for reproduction. Selection balances exploration and exploitation, influencing algorithm performance and convergence.

3.2. Parallel genetic algorithm

GA, like all metaheuristics, is time-consuming, which is the same issue with exact search algorithms [55]. Thus, parallel runs of these algorithms arise as a promising approach for overcoming this flaw.

Parallel Islands GA (PIGA) emerged as the best and most common GA parallelization approach [56,57]. This parallel model is widely used as a promising approach to parallelize GAs [23,25,58]. In this model, the GA population is divided into sub-populations (islands) as shown in Fig. 1.

These islands are distributed over the parallel processors to run in parallel. They execute the identical code of the standard GA and can evolve in physical parallelism over the different processors. The model involves a migration procedure that requires sharing search knowledge by exchanging the individuals between these small-distant populations [59]. Periodic migration occurs among islands, with ring migration being an example, as outlined in Algorithm 2.

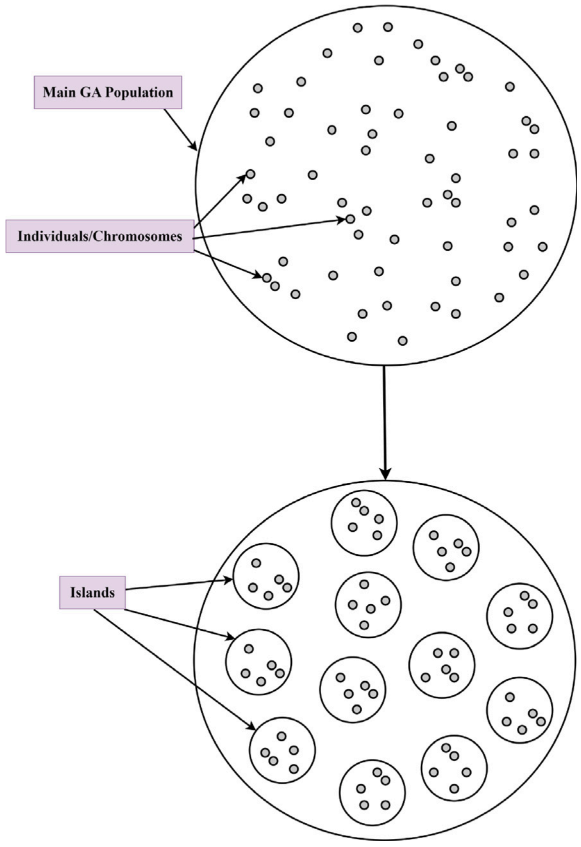


Fig. 1. Population partitioned into islands.

Algorithm 2 Ring migration among islands.

```
1: Input: Population of individuals, $P$; Number of islands, $nIs$.
2: procedure RingMigration($P$, $nIs$)
3:     for i ← 0 to nIs − 1 do
4:         $P_{(i+1)\%nIs}$'s worst individual $\leftarrow$ $P_i$'s best solution.
5:     end for
6: end procedure
```

4. Bi-objective optimization problem formulation and exact algorithms

In this section, we present our methodology to obtain Pareto-optimal solutions in our study. We formulate the bi-objective optimization problem HPIGAOPT, which minimizes the execution time and energy consumption of PIGA applications. HPIGAOPT generates a set of globally Pareto-optimal solutions for both execution time and energy. These applications execute on a heterogeneous hybrid platform comprising p heterogeneous processor, using the optimal application configuration and a fixed platform configuration. The problem employs workload distribution, a vector of p workload sizes where a workload size is the number of islands, representing the application configuration as the decision variable.

The problem considers the execution of an application workload of size n representing the total number of islands on a heterogeneous hybrid platform with base/idle power consumption, B_s, and comprising p heterogeneous processors. Let the sets, $T = \{t_1(x), \ldots, t_p(x)\}$, and $E = \{e_1(x), \ldots, e_p(x)\}$, contain the execution time and dynamic energy functions of workload size of the p processors. The function $e_i(x)$ gives the amount of dynamic energy consumed by P_i to execute the workload size x, and $t_i(x)$ is the execution time of the workload size x on this processor. The optimization problem formulation is as follows:

HPIGAOPT$(n, p, T, E, B_s, \mathcal{N})$:

$$f_T(\mathcal{N}) = \max_{i=1}^{p} t_i(n_i)$$

$$f_E(\mathcal{N}) = B_s \times \max_{i=1}^{p} t_i(n_i) + \sum_{i=1}^{p} e_i(n_i)$$

$$\underset{\mathcal{N}}{\text{minimize}} \quad (f_T(\mathcal{N}), f_E(\mathcal{N})) \tag{1}$$

$$\text{subject to:} \quad \sum_{i=1}^{p} n_i = n, 0 \le n_i \le n, i \in \{1, \dots, p\}$$

The two objective functions are $f_T(\mathcal{N})$ and $f_E(\mathcal{N})$. The objective function $f_T(\mathcal{N})$ gives the execution time of the application workload of size n employing the workload distribution, $\mathcal{N} = \{n_1, \dots, n_p\}$. The objective function $f_E(\mathcal{N})$ gives the total energy consumption during the execution of the application workload. $f_T \times f_E : \mathbb{R}_{\ge 0} \times \mathbb{R}_{\ge 0}$ denotes the objective space of this problem. HPIGAOPT returns globally Pareto-optimal solutions (workload distributions) minimizing the two objective functions. If the input parameter B_s is 0, HPIGAOPT returns a set of globally Pareto-optimal solutions for execution time and dynamic energy.

4.1. Exact algorithms for the optimization problem variants

We present an overview of the exact algorithms solving HPIGAOPT for two different categories of time and energy functions. The exact algorithms, LBOPA-TE and PARTITION, solve a special case of HPIGAOPT where the sets T and E contain linear increasing execution time and dynamic energy functions of workload size, respectively [60]. LBOPA-TE outputs a piecewise linear Pareto front comprising a number of segments less than or equal to $p - 1$. Furthermore, given a point on the Pareto front, PARTITION finds the optimal workload distribution.

The exact algorithm, HEPOPTA, solves a special case of HPIGAOPT where the sets T and E contain discrete execution time and dynamic energy functions (with arbitrary shape and represented by a set of points) of workload size [61]. HEPOPTA returns a set of tuples:

$\{(f_E(\mathcal{N}_{opt}), f_T(\mathcal{N}_{opt}), \mathcal{N}_{opt})\}$, where $\mathcal{N}_{opt}$ is a vector of size p giving the optimal workload distribution, $f_T(\mathcal{N}_{opt})$ is the optimal execution time, and $f_E(\mathcal{N}_{opt})$ is the optimal total energy.

5. Design and implementation of heterogeneous hybrid PGA

This section describes the design and implementation of our Heterogeneous Parallel Island Genetic Algorithm (HPIGA), focusing on its execution on a hybrid server platform that includes multicore CPUs and accelerators.

5.1. HPIGA design

HPIGA takes as input a set of islands (nIs), which are randomly generated. The target platform is a hybrid heterogeneous system consisting of a multicore processor and multiple accelerators (np). The islands are partitioned among the computing devices available in the hybrid system. Unlike common approaches found in the literature, our design utilizes a self-adaptive algorithm that distributes the workload across the available computing devices in the hybrid system, proportionally to their performance.

The parallel model in this study is the distributed island model described in [22,54]. The model involves partitioning the islands to align with the distributed processors available, ensuring optimal distribution and utilization across the system. Algorithm 3 describes the basic structure of the PIGA algorithm.

We consider the uniform crossover operator, which is a commonly used operator in the literature [53]. In the uniform crossover method, each bit is randomly selected from either parent with a predefined

Algorithm 3 Parallel Genetic Algorithm (PGA)

```
1: Input: Population of individuals, $P$; Number of islands, $nIs$;
   Number of processors, $np$.
2: Output: Best individual in the population.
3: Initialization: Divide the population $P$ into $nIs$ islands,
   $\{P_0, \cdots, P_{nIs-1}\}$.
4: procedure Parallel Islands GA($P$, $nIs$, $np$)
5:     #pragma omp parallel for numthreads($np$)
6:     for $i \leftarrow 0$ to $nIs - 1$ do
7:         Fitness Evaluation($P_i$)
8:     end for
9:     while termination criterion NOT met do
10:        #pragma omp parallel numthreads($np$)
11:        #pragma omp for
12:        for $i \leftarrow 0$ to $nIs - 1$ do
13:            Crossover($P_i$)
14:            Mutation($P_i$)
15:            Fitness Evaluation($P_i$)
16:            Selection($P_i$)
17:        end for
18:        if migration condition met then
19:            RingMigration($P$, $nIs$)
20:        end if
21:    end while
22: end procedure
```

probability, ensuring an efficient inheritance of genetic information from both parents. We consider the Bit-Flip mutation, which involves selecting one or more random bits and flipping them. This mutation is typically employed in binary-encoded Genetic Algorithms (GAs) [62]. Binary tournament selection is employed for selection. Tournament selection involves conducting multiple tournaments among randomly chosen individuals from the population. This procedure involves randomly selecting two solutions from the population and selecting one based on its fitness for the next generation.

GAs termination criteria are essential for determining when the optimization process should end. Typically, these criteria are based on reaching a certain number of generations, reaching a certain number of function evaluations, or when the algorithm converges to a stable solution. These different criteria help prevent premature convergence and ensure that the GA terminates efficiently based on each experiment's specific objectives and constraints. Once the stopping criteria are met, each processor identifies the best individual within its subset of islands. The multicore processor's main thread then aggregates these best individuals. The algorithm returns the overall best solution found among all processors. This parallel approach leverages the diverse computational capabilities of heterogeneous processors, maximizing efficiency and accelerating the optimization process.

HPIGA employs the island model, which involves the random generation of islands, each representing a distinct sub-population. The islands each have an identical number of individuals. This is a data-parallelism approach that considers individualized GA for each island's operation. We assign these islands to the cores and threads across the various computational devices available in the system. Fig. 2 shows the structural design of HPIGA where the entire set of islands is divided among the available computing devices. Subsequently, these subsets of islands are executed in parallel across the computing devices. Algorithm 4 outlines a procedure for partitioning islands based on the speeds/performances of heterogeneous processors.

The partitioning algorithm assigns islands to processors based on their speeds. It calculates the total speed (T) and allocates islands proportionally. The remaining islands are individually allocated to processors starting from 0. The final vector (N) is then returned, showing the number of islands assigned to each processor. Periodic

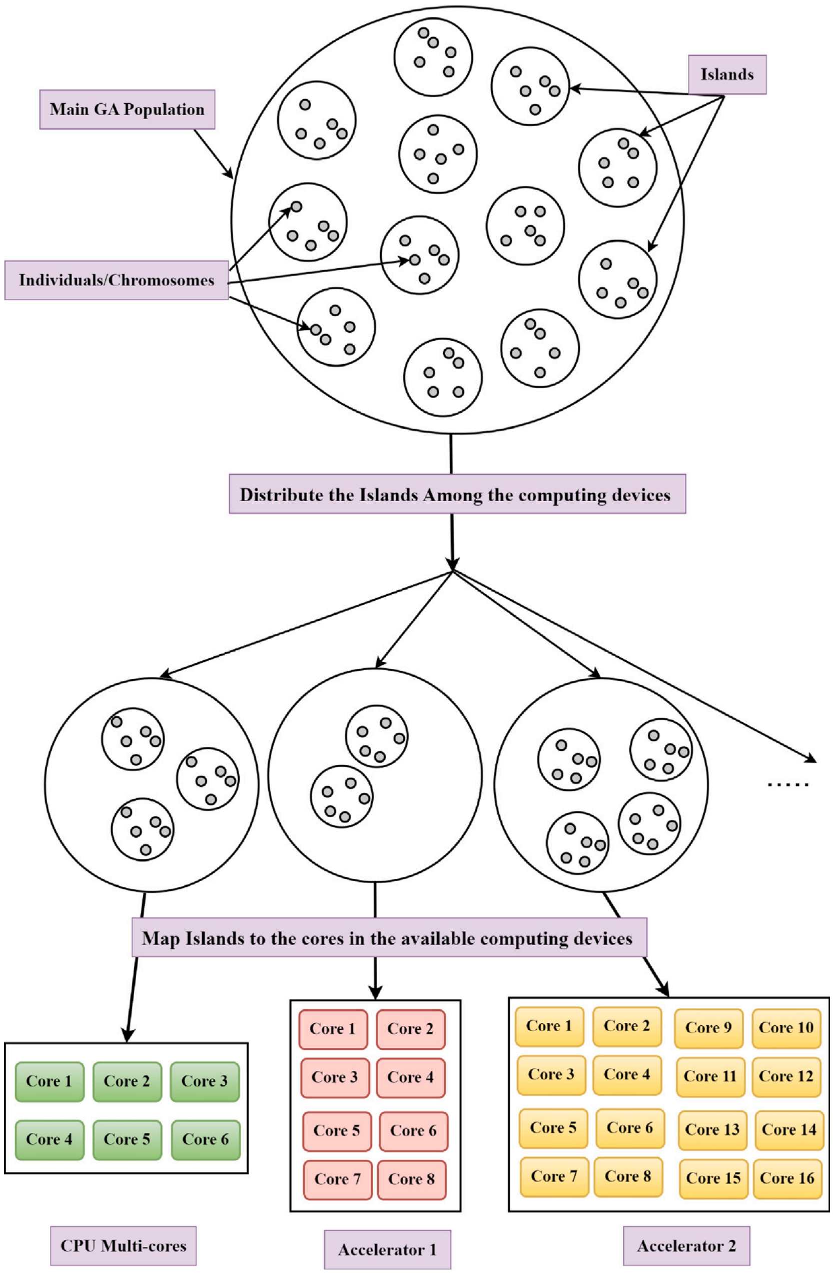


Fig. 2. Island distribution in HPIGA across CPUs and accelerators.

communication occurs between these islands through migration. We employ the ring migration policy described in [22]. In this operation, each island sends its best individuals to another island in a ring fashion in a preset iteration number described in Algorithm 2. The migration interval and rate are set to ensure independent exploration and structured communication. A fixed number of the best individuals migrate each interval, replacing the lowest fitness individuals on the destination island. This migration topology has a low communication overhead compared with other topologies [63]. The high-level HPIGA description is provided in algorithm 5.

HPIGA begins with a population of islands P that target running concurrently on a system with heterogeneous processors. The partitioning algorithm 4 is called to divide the islands among the heterogeneous processors D based on their computational speeds S to ensure an equitable workload distribution. Each processor independently runs a PGA kernel that executes genetic algorithms. Within each processor i, the workload is further divided among cores nt_i according to the specified number of islands assigned. This strategy optimizes the computational resources, allowing multiple cores/threads to work on different subsets of islands concurrently.

5.2. HPIGA implementation

We develop an HPIGA implementation for a heterogeneous hybrid server comprising p computing devices, one multicore CPU, and $p-1$ accelerators. The implementation is based on OpenH [19], a programming model and API for developing portable parallel programs on heterogeneous hybrid servers composed of a multicore CPU and one or more accelerators (generally speaking, of different types).

An OpenH parallel program executing on a heterogeneous hybrid server is composed of several software components (kernels) executing in parallel. There is a one-to-one mapping between the components and computing devices of the hybrid platform on which the program is executed. The execution of an accelerator component involves a

Algorithm 4 Partitioning islands between processors proportional to their speeds/performances.

```
1: Input:
     • Number of islands to divide, $n$.
     • Number of heterogeneous processors, $D$.
     • Vector of processor speeds, $S = \{s_i\}_{i=0}^{D-1}$, sorted in descending order.
2: Output: Vector $N$ containing the number of islands assigned to each processor $N = \{n_i\}_{i=0}^{D-1}$.
3: procedure HSP($N, D, S$)
4:     Calculate total speed $T = \sum_{i=1}^{D} s_i$.
5:     for $i = 0$ to $D - 1$ do
6:         $n_i = \left\lfloor \frac{s_i}{T} \times n \right\rfloor$
7:     end for
8:     for $i = 0$ to $(D - 1 - \sum_{j=0}^{D-1} N_j)$ do
9:         $n_i = n_i + 1$
10:    end for
11:    return N
12: end procedure
```

Algorithm 5 Heterogeneous Parallel Islands GA (HPIGA) algorithm.

```
1: Input:
2: $P = \{is_0, is_1, ...., is_{n-1}\}$: Population of $n$ islands.
3: $D$: Number of heterogeneous processors.
4: $nt = \{nt_0, nt_1, nt_2, ..., nt_{D-1}\}$: Number of cores per processor.
5: $S = \{s_0, s_1, s_2, ..., s_{D-1}\}$: Speeds of processors.
6: Output: Best individual in the population.
7: procedure HPIGA($P, n, D, S, nt$)
8:     $b = \{b_0, b_1, b_2, ..., b_{D-1}\} \leftarrow \{0, ..., 0\}$
9:     $N = \{n_0, n_1, ..., n_{D-1}\} \leftarrow$ HSP($n, D, S$)
10:    #pragma omp parallel numthreads(D)
11:    for $i \leftarrow 0$ to $D - 1$ do
12:        $myP \leftarrow \{is_{\sum_{j=0}^{i-1} n_j}, \dots, is_{\sum_{j=0}^{i} n_j - 1}\}$
13:        $b[i] \leftarrow$ PARALLEL ISLANDS GA($myP, n_i, nt_i$)
14:    end for
15:    return Best individual in $b$.
16: end procedure
```

dedicated CPU core, running the hosting thread, and the accelerator itself, performing the accelerator code. The execution of the accelerator component includes data transfer between the CPU and accelerator memory, computations by the accelerator code, and data transfer between the accelerator memory and CPU. The execution of a CPU component only involves the CPU cores executing the multithreaded CPU code.

The parallel program starts as a single *main* thread, creating a group of Pthreads called *hosting* Pthreads. The hosting Pthreads lead the execution of the software components of the hybrid parallel program in parallel. A CPU hosting Pthread leads the execution of a multi-threaded CPU software component employing either OpenMP or a multi-threaded library routine. There can be one or more CPU software components and, therefore, one or more CPU hosting Pthreads. For a CPU component employing an OpenMP parallel region, the hosting Pthread of the component becomes the *master* thread of the region. An accelerator hosting Pthread leads the execution of an accelerator component, which is an OpenACC (or OpenMP) component running on one of the accelerators of the server. Finally, the OpenH library provides API functions that allow programmers to get the configuration of the executing environment. Furthermore, the library provides API functions for binding the hosting Pthreads (and hence the execution of the software components) to the CPU cores of the hybrid server to get the best performance. Fig. 3 illustrates the HPIGA implementation using p computing devices (one multicore CPU with nlc logical cores and $p-1$ accelerators).

The HPIGA code snippet shows that the main thread creates a group of p hosting CPU and accelerator Pthreads to manage the execution of the CPU and accelerator software components in parallel. The routine *HSP()* partitions the population into islands using the speeds of the software components estimated at runtime. Each computing device is assigned a subset of islands proportional to its speed, ensuring balanced load distribution. The relative speeds of the software components using the OpenH API function, *openh_perf_benchmark()*, which executes small representative benchmark codes of the software components solving the same workload size in parallel. The execution times of all the benchmark codes are measured simultaneously, thereby considering the influence of resource contention. The API function implementation essentially executes a mini-version of the HPIGA employing the same affinity and binding settings for the Pthreads executing the benchmark codes as the hosting Pthreads and the same library settings for the library routines invoked in the software component implementations.[1]

Fig. 4 illustrates the OpenH fork-join model and assignment of islands to the components within HPIGA. The CPU software component executes the island GA on the multicore CPU using OpenMP. The accelerator component executes the island GA on the accelerator using OpenACC.

Fig. 5 complements the snippets of the HPIGA implementation. We begin by detailing the execution steps of the *main* thread. Line 7 initializes the OpenH library runtime using the API function *openh_init()*. The API function, *openh_get_num_accelerators*, returns the number of accelerators (Line 9). The variable p stores the number of hosting Pthreads in the program, which is equal to the number of software components. Lines 13–16 determine and assign the physical CPU core IDs closest to the accelerators for binding the accelerator hosting Pthreads. The API function, *openh_get_unique_lcore(i)*, returns the unique OpenH logical CPU core ID closest to the input accelerator, *i*. The hosting Pthread for the accelerator i is assigned the place using the API function, *openh_assign_acc_lcpuids* (Line 15). The OpenH library functions ensure that different accelerator hosting Pthreads are pinned to OpenH logical CPU core IDs that map to different OpenH physical CPU core IDs for optimal performance.

The CPU software component with ID 0 is assigned the remaining OpenH logical CPU core IDs using the API function, *openh_assign_cpu_free_lcpuids()*, that executes the component (Line 18). Lines 22–32 show the creation of the $p-1$ accelerator hosting Pthreads responsible for executing the accelerator components. The partition data for an accelerator software component is filled in Lines 24–28. Lines 31–33 contain the filling of the partition data for the CPU component and the creation of the hosting Pthread leading the execution of the CPU component. After completing the computations, the main thread synchronizes/joins with the p hosting Pthreads in Lines 34–36. Finally, the OpenH runtime is destroyed using the API function, *openh_finalize()* in Line 37.

Fig. 5 shows the main code fragments of the software components. Lines 1–9 contain the CPU software component code. The hosting Pthread leading the execution of the CPU component with ID, *cpuComponentId*, is first bound using the API function, *openh_bind_cpu_self()*. Then, it executes the PIGA_CPU() routine to perform the island GA on the CPU. Lines 10–20 demonstrate the execution of a component employing an accelerator. The hosting Pthread leading the execution of *accId* is bound using the API function, *openh_bind_acc_self()*. The GPU device ID is set using the OpenACC library function, *acc_set_device_num*

[1] The presented design automatically finds at runtime the performance optimal distribution of the workload between the heterogeneous devices. We disable this feature when using the workload distribution as the decision variable for bi-objective optimization for performance and energy in Section 7.

```c
#include <openh.h>
typedef struct _ComponentPartition_t {
  int cId; int nIs; float* P;
} ComponentPartition_t;

int main(int argc, char *argv[]) {
   openh_init();
   /* Get the number of accelerators */
   int nacc = openh_get_num_accelerators();
   /* Set the number of software components */
   int p = nacc+1;
   /* Assign the CPU core IDs for the accelerator hosting Pthreads */
   for (i = 0; i < nacc; i++) {
     int lcoreid = openh_get_unique_lcore(i);
     openh_assign_acc_lcpuids(i, &lcoreid, 1);
   }
   /* Assign the remaining CPU core IDs that execute the CPU component
       */
   openh_assign_cpu_free_lcpuids(0);
   int* islands; double* speeds;
   /* Estimate speeds using openh_perf_benchmark function */
   HSP(nIs, p, speeds, islands);
   int start = 0; ComponentPartition_t cData[p];
   pthread_t compT[p];
   for (i = 0; i < p-1; i++) {
      cData[i].cId = i;
      cData[i].nIs = islands[i];
      cData[i].P = &P[start];
      pthread_create(&compT[i], NULL, accComponent,
          (void*)(&cData[i]));
      start += islands[i];
   }

   cData[p-1].cId = 0;
   cData[p-1].nIs = islands[p-1];
   cData[p-1].P = &P[start];
   pthread_create(&compT[p-1], NULL, cpuComponent,
       (void*)(&cData[p-1]));
   for (i = 0; i < p; i++) {
     pthread_join(compT[i], NULL);
   }
   openh_finalize();
}
```

Fig. 3. The OpenH HPIGA code illustrates the OpenH fork-join model of execution. The OpenH library calls are highlighted in bold. The main thread creates the group of hosting CPU and accelerator Pthreads to lead the execution of the CPU and the accelerator software components in parallel. The CPU hosting Pthread will lead the execution of the CPU component, *cpuComponent*. The accelerator hosting Pthread will lead the execution of the GPU component, *accComponent*.

(Line 16). Then, it executes the PIGA_ACC() routine to perform the island GA on the accelerator.

The complete source code is available at [64] containing the implementations for the CPU and accelerator components. The CPU software component employs OpenMP. The accelerator software component employs OpenACC. While the code description above is generic and, for p computing devices, the code at [64] supports four versions. In the first version, HPIGA has one accelerator component that runs on one accelerator. The second version has only the CPU software component executing on the multicore CPU. The third version executes initial population generation and migration on the CPU and the main genetic operations (crossover, mutation, and selection) on the accelerators. The fourth version is HPIGA, that distributes the workload between the multicore CPU and accelerators.

6. Experimental framework

In our pursuit of analyzing energy efficiency and performance, we conduct experiments with various configurations featuring different numbers of islands. The underlying objective of this experimental design is to effectively address the research goals and overcome optimization challenges related to energy consumption and system performance across diverse island quantities. Our deliberate exploration of different running strategies is geared towards achieving a comprehensive understanding of how our algorithm behaves under varying conditions. By systematically evaluating the outcomes, we strive to identify the optimal configuration for our algorithm, contributing valuable insights for future algorithmic designs. Ultimately, our aim is to configure our application to operate in an energy-aware manner, facilitating the development of more efficient algorithms and fostering advancements in computational methodologies.

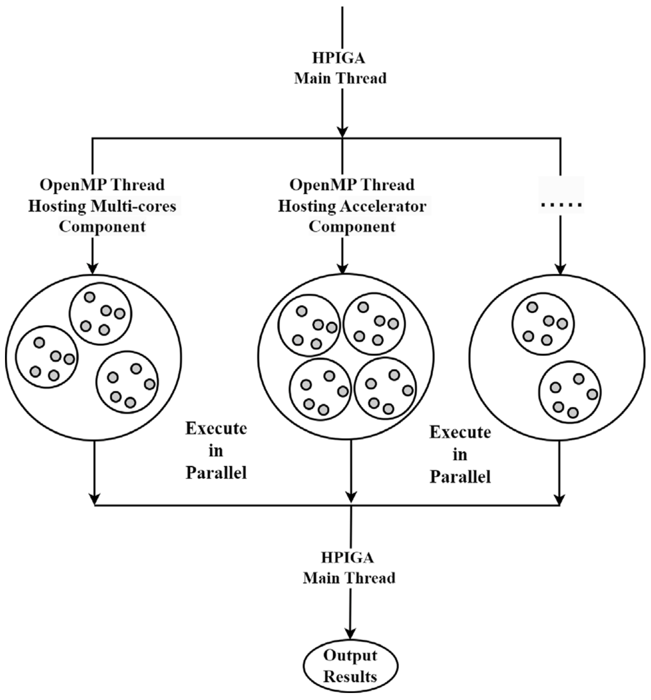


Fig. 4. The OpenH fork-join execution of the HPIGA. The main thread creates the group of p hosting CPU and accelerator Pthreads to lead the execution of the CPU and the accelerator software components in parallel. The islands are distributed among the computing devices proportional to their speeds/performances.

We perform two extensive experiments to examine the performance and energy consumption of PGAs running on hybrid systems. Furthermore, we conduct a second experiment to generate Pareto front sets for optimal execution time and energy consumption across these system configurations. In detail, we run the distributed island model of the GA using a different number of islands and dimensions over our hybrid server that includes multicore CPUs and accelerators.

The first experiment considers running four different parallel strategies. The first strategy is running the algorithm over one accelerator/GPU (named PIGA_ACC). In this setting, all the islands and the genetic operations are parallelized over one of the accelerators available on our system. The second strategy considers running the exact algorithm over a multicore CPU (named PIGA_CPU). We remark that we have only one kernel for executing all our four strategies in order to ensure fair comparisons and accurate Pareto front results.

The third and fourth parallel strategies consider running the island model in hybrid parallel execution over the CPU and accelerators. The third strategy employs an innovative heterogeneous CPU/Accelerator functional parallelism approach (named PIGA_HFP). In this experiment, Random Number Generation (RNG) for each generation is generated on the CPU and sent to the accelerator where the main genetic operations (crossover, mutation, and selection) are executed; all other operations (initial population generation and migration) are executed by the CPU. This design is proposed to divide the workload between the two devices and relies on the CPU for random number generation since RNG on CPUs using *rand_r* function is thread-safe and has a common use in literature [65]. The fourth strategy (named HPIGA) involves executing our proposed algorithm in Section 5, by dividing and distributing the GA islands across the three computational devices (CPU and two accelerators) in our system. The primary objective of this configuration is to investigate the common assertion in the literature that using more computational devices may lead to higher energy consumption.

In the second experiment, we perform extensive runs for our PIGA over the CPU and accelerator with a growing number of islands, starting from two islands to 128 islands. These experiments aim to generate the Pareto front set of optimal solutions for execution time and energy consumption for the hybrid PGA application employing the powersave and performance DVFS governors on the multicore CPU.

Table 1
Parameter settings.

Definitions	Values
Sub-population size	50 individual
Recombination	Uniform, $pc = 0.6$
Mutation	Bit-flip, $pm = 0.0001$
Selection	Binary tournament
Replacement	Replacing the worst
Elitism	Yes
Migration interval	Every 10 iterations

These two experiments employ different stopping conditions due to the experimental setup used. In the first experiment, the termination condition is defined by the total number of function evaluations, which remains the same for all island configurations within each dimension. However, due to the setup of the algorithm used to generate the Pareto fronts, the second experiment employs a fixed number of iterations. The specific values for this setup are provided in the next section. Our experiments were conducted using both powersave and performance governors. Choosing two different power governors can provide the researchers with beneficial insights into the energy consumption behavior and performance characteristics of parallel GAs. Furthermore, presenting Pareto-fronts Figures and discussion gives an optimal analysis of power consumption patterns. Hence, this detailed examination facilitates performance tuning based on the nature of the workload.

6.1. Benchmark problem and parameter settings

We summarize the benchmark problems used to evaluate our experimental design and parameter settings. Our primary focus in this research is to study the performance and energy profiles of PGAs. To achieve this, we employ two well-established test problems: One-Max and Rastrigin. One-Max provides a simple search space, allowing for a clear assessment of algorithmic behavior while ensuring reliable performance and energy consumption profiles. In contrast, the Rastrigin function presents a more computationally expensive landscape with numerous local optima. By evaluating both functions, we can assess the GA's efficiency across varying levels of problem complexity and computational cost. The mathematical formulations of the two benchmark problems are given below.

1- One-Max Problem (f_1) Definition:

$$f_1(\mathbf{x}) = \sum_{i=1}^{n} x_i, \quad x_i \in \{0,1\} \tag{2}$$

Search space: $x_i \in \{0,1\}$, $i = 1,\dots,n$. **Global minimum:** $\mathbf{x}^* = (0,0,\dots,0)$; $f_1(\mathbf{x}^*) = 0$.

2- Rastrigin Function (f_2) Definition:

$$f_2(\mathbf{x}) = 10n + \sum_{i=1}^{n} \left(x_i^2 - 10\cos\left(2\pi x_i\right)\right). \tag{3}$$

Search space: $-5.12 \le x_i \le 5.12$, $i = 1,\dots,n$. **Global minimum:** $\mathbf{x}^* = (0,0,\dots,0)$; $f_2(\mathbf{x}^*) = 0$.

Our experiments include lower dimensions (100) as well as higher dimensions (1000 and 5000), which result in increased computational effort and enable a more comprehensive energy profile analysis. Table 1 shows the problem parameter settings used in our experiments. We base our selection of these parameters on the most frequently employed values documented in the literature, and the parameters used in a preceding study [22]. In this study, we determine these parameters through various preliminary numerical experiments to illustrate distinctions among the algorithms under examination.

Our migrant selection policy for the island model involves choosing the best individuals from the sending island and replacing the worst

```
void *cpuComponent(void *arg) {
  ComponentPartition_t cData = *(ComponentPartition_t*)arg;
  int cpuComponentId = cData.cId;
  /* Bind the CPU hosting Pthread */
  openh_bind_cpu_self(cpuComponentId);
  int nIs = cData.nIs;
  float* P = cData.P;
  PIGA_CPU();
}
void *accComponent(void *arg) {
  ComponentPartition_t cData = *(ComponentPartition_t*)arg;
  int accId = cData.cId;
  /* Bind the non-GPU accelerator hosting Pthread */
  openh_bind_acc_self(accId);
  if (openh_get_acc_type(accId)== OPENH_CUDA_GPU)
    acc_set_device_num(accId, acc_device_nvidia);
  int nIs = cData.nIs;
  float* P = cData.P;
  PIGA_ACC(nIs, P);
}
```

Fig. 5. The OpenH HPIGA implementation is decomposed into p software components (one CPU and $p-1$ accelerator components) and executed on a heterogeneous platform comprising p computing devices, one multicore CPU, and $p-1$ accelerators. The CPU software component implementation is in the function, *cpuComponent*. The software component implementation specific to accelerators is in the function, *accComponent*. PIGA_CPU() executes the island GA on the multicore CPU using OpenMP. PIGA_ACC() kernel executes the island GA on the accelerator using OpenACC.

Table 2
Iterations and evaluations per dimension.

Dimension	Experiment 1 # of Evaluations	Experiment 2 # of Iterations
100	4E6	1250
500	5E6	1500
1000	7E6	2000
3000	1.5E7	6000
5000	2E7	8000

individuals on the destination island. We set the migration rate to be 5 individuals per communication phase, with migration occurring every 10 iterations. Table 2 provides the number of function evaluations and iterations for each dimension employed in first and second experiments, respectively. These values were determined through a series of preliminary experiments to ensure that all instances could reach the optimal solution. The number of function evaluations and iterations per dimension was kept constant across different island configurations to ensure a fair comparison and reproducibility of results.

6.2. System specifications and energy measurement

We employ the research hybrid server platform whose specifications are given in Table 3 for our experiments. The two Nvidia A40 GPUs are closest to all the cores (0–63) in the Intel Icelake multicore CPU of the hybrid server.

We employ system-level physical measurements using external power meters for component-level measurement of energy consumption. The measurements obtained this way are considered ground truth [66].

The hybrid server has one WattsUp Pro power meter between the wall A/C outlets and the node's input power sockets. The power meter captures the total power consumption of the node. It has a data cable connected to one USB port of the node. A Perl script collects the data from the power meter using the serial USB interface. The execution of these scripts is non-intrusive and consumes insignificant power. The power meters are periodically calibrated using an ANSI C12.20 revenue-grade power meter, Yokogawa WT210. The maximum sampling speed of the power meters is one sample every second. The

Table 3
Specifications of the Intel hybrid server containing a single-socket Icelake multicore CPU and two Nvidia A40 GPUs.

Intel Platinum 8362 Icelake	
No. of cores per socket	32
No. of threads per core	2
Socket(s)	2
L1d cache, L1i cache	1.5 MiB, 1 MiB
L2 cache, L3 cache	40 MiB, 48 MiB
Total main memory	62 GB DDR4-3200
TDP	265 W
NVIDIA A40 GPU	
No. of GPUs	2
No. of Ampere cores	10,752
Total board memory	48 GB GDDR6 (with ECC)
Memory bandwidth	696 GB/s
TDP	300 W

accuracy specified in the data sheets is $\pm 3\%$. The minimum measurable power is 0.5 W. The accuracy at 0.5 W is ± 0.3 W. The static power consumption of the server is 146 W.

To ensure the reliability of our results, we follow a statistical methodology where a sample average for a response variable (execution time and energy) is obtained from multiple experimental runs. The sample average is calculated by executing the application repeatedly until it lies in the 95% confidence interval and a precision of 0.05 (5%) is achieved. For this purpose, Student's t-test is used, assuming that the individual observations are independent and their population follows the normal distribution. We verify the validity of these assumptions using Pearson's chi-squared test.

7. Experimental results and analysis

In this section, we present the outcomes of our two distinct experiments designed to evaluate the performance of PIGA under various parallelization strategies. These experiments focus on presenting robust energy and performance profiles of PGAs running on multi-cores and accelerators, along with generating and analyzing Pareto front sets for optimal execution time and energy consumption across various system configurations.

Table 4
Mean total energy (Joules) for f_1 and f_2.

# of Islands	Algorithm	Dimensions under study									
		100		500		1000		3000		5000	
		f_1	f_2	f_1	f_2	f_1	f_2	f_1	f_2	f_1	f_2
128	PIGA_ACC	**462.63**	577.80	2 328.96	**2 562.91**	4 435.43	5 224.72	32 039.00	42 494.49	72 940.66	87 980.97
	PIGA_CPU	622.56	824.72	**2 206.20**	2 981.81	**3 620.10**	**4 727.74**	**22 499.36**	**31 822.14**	**50 844.00**	**69 379.94**
	PIGA_HFP	1991.80	3178.21	11 534.40	19 259.74	27 334.80	47 099.52	182 921.66	279 499.40	414 800.00	705 554.53
	HPIGA	424.93	**551.30**	2 245.13	2 753.49	7 573.27	9 305.09	55 920.66	67 520.86	133 355.66	168 353.49
64	PIGA_ACC	**456.83**	**614.03**	2 329.47	3 177.83	4 334.03	**5 153.36**	30 362.10	37 212.29	74 504.33	103 714.30
	PIGA_CPU	695.50	1060.85	2 708.23	3 760.44	**4 106.57**	5 710.52	**22 891.77**	**31 822.78**	**49 992.00**	**71 181.85**
	PIGA_HFP	1994.36	3256.82	11 530.27	18 105.14	27 039.90	41 071.65	180 245.00	309 355.82	414 213.33	720 104.93
	HPIGA	557.30	806.14	**2 019.80**	**3 013.32**	5 944.63	8 734.92	50 194.33	70 950.03	124 134.66	165 759.06
32	PIGA_ACC	**427.00**	**521.73**	2 324.37	3 177.12	**4 406.97**	**5 846.12**	28 592.76	35 381.40	70 088.66	81 747.47
	PIGA_CPU	694.00	1091.82	3 786.13	5 952.17	5 261.97	8 128.16	**25 346.33**	**33 589.44**	**52 597.33**	**77 497.54**
	PIGA_HFP	2033.13	3395.94	11 573.13	18 036.73	27 365.20	45 729.35	175 059.00	297 152.17	408 833.33	679 931.69
	HPIGA	821.36	1205.90	**2 069.37**	**3 013.32**	8 750.0	8 750.00	48 818.66	66 902.38	81 515.00	119 222.83
16	PIGA_ACC	**431.40**	**555.16**	2 331.57	3 096.69	**4 438.13**	**5 406.59**	**28 361.16**	**34 325.95**	64 764.00	**84 695.38**
	PIGA_CPU	708.43	1110.04	4 817.13	6 829.81	7 674.07	11 612.43	30 389.23	43 836.74	**61 600.00**	94 700.42
	PIGA_HFP	1933.96	3110.09	11 658.87	18 786.34	27 385.13	47 005.75	175 882.00	277 299.06	399 033.33	621 759.30
	HPIGA	1094.46	1395.71	**2 156.73**	**2710.66**	4 940.33	7 245.69	47 497.66	69 347.09	105 896.66	149 745.64

Boldfaced and underlined values represent the least and highest values for each algorithm per function within each dimension, respectively.

Table 5
Mean dynamic energy (Joules) for f_1 and f_2.

# of Islands	Algorithm	Dimensions under study									
		100		500		1000		3000		5000	
		f_1	f_2	f_1	f_2	f_1	f_2	f_1	f_2	f_1	f_2
128	PIGA_ACC	**214.12**	**267.42**	894.07	983.88	1 770.76	2 085.86	13 341.81	17 695.70	29 605.49	35 710.12
	PIGA_CPU	286.88	380.03	**415.93**	**562.15**	**559.24**	**730.34**	**6 308.15**	**8 921.97**	**16 535.28**	**22 563.46**
	PIGA_HFP	1246.72	1989.32	7114.87	11 880.16	18 910.19	32 583.40	126 514.35	193 310.54	286 850.06	487 917.92
	HPIGA	296.40	384.55	1431.06	1 755.08	5 252.44	6 453.54	38 650.75	46 668.47	92 371.44	116 613.38
64	PIGA_ACC	**209.03**	**280.96**	895.73	1 221.95	1 735.51	2 063.60	12 454.69	15 264.67	31 161.98	43 379.26
	PIGA_CPU	320.73	489.21	**739.94**	**1 027.42**	**838.38**	**1 165.83**	**3 538.14**	**4 918.51**	**12 583.61**	**17 917.35**
	PIGA_HFP	1245.99	2034.71	7139.87	11 211.22	18 891.97	28 695.53	126 420.90	216 977.12	285 141.37	489 450.49
	HPIGA	383.32	554.47	1286.62	1 919.49	4 159.40	6 111.74	35 142.62	49 674.33	84 841.72	113 290.54
32	PIGA_ACC	**186.07**	**227.35**	**887.53**	**1 213.15**	1 811.79	**2 403.45**	11 699.94	14 477.80	28 824.57	33 619.35
	PIGA_CPU	321.12	505.19	1446.00	2 273.25	**1 626.38**	2 512.26	**4 652.75**	**6 165.92**	**8 395.30**	**12 369.74**
	PIGA_HFP	1269.07	2119.73	7189.91	11 205.33	19 127.09	31 962.83	120 576.31	204 671.06	281 468.22	468 110.46
	HPIGA	556.28	816.72	1330.20	1 982.19	4 190.31	6 105.17	33 691.93	46 172.30	56 202.04	82 200.42
16	PIGA_ACC	**197.58**	**254.26**	**899.72**	**1 194.97**	**1 832.93**	**2 232.90**	11 584.69	14 021.12	26 538.00	34 705.18
	PIGA_CPU	334.45	524.05	2569.64	3 643.27	3 470.68	5 251.84	**7 932.32**	**11 442.44**	**12 318.02**	**18 937.04**
	PIGA_HFP	1228.09	1974.95	7250.09	11 682.32	19 071.87	32 736.29	120 998.56	190 768.74	275 529.20	429 319.64
	HPIGA	729.01	929.67	1384.79	1 740.45	3 455.70	5 068.27	32 689.37	47 726.83	73 114.82	103 389.72

Boldfaced and underlined values represent the least and highest values for each algorithm per function within each dimension, respectively.

7.1. Experiment 1: Energy and performance analysis

In this experiment, we run the PIGA using four different parallel approaches described earlier in Section 6. For this experiment, we employed the default “ondemand” governor as the energy governor. This choice was not only practical, as it is the default for Ubuntu Servers, but also provides a standardized and dependable basis for performance and energy analysis. The *ondemand* governor’s unique capability to automatically modify CPU frequency based on demand makes it a useful choose for exploring energy efficiency and computational performance.

Tables 4 and 5 show the total and dynamic energy consumption (in Joules) for various dimensions.

An analysis of total energy consumption (Table 4) reveals that PIGA_ACC consistently achieves the highest energy efficiency, exhibiting the lowest energy usage (boldfaced values) in most test cases across both functions and all dimensions. This highlights its advantage in leveraging acceleration hardware for energy-conscious optimization. PIGA_CPU demonstrates moderate energy consumption, outperforming PIGA_HFP in some lower-dimensional cases but generally consuming more energy than both PIGA_ACC and HPIGA, especially as dimensionality increases. HPIGA, by utilizing hybrid computing resources, offers a balanced energy profile, frequently outperforming PIGA_CPU and significantly reducing energy consumption compared to PIGA_HFP in higher-dimensional problems. In contrast, PIGA_HFP consistently exhibits the highest total energy consumption across all dimensions (underlined values) due to its design, which incurs costly inter-device communication overhead. These results confirm that the PIGA_HFP design is impractical, a logical conclusion now empirically validated.

The dynamic energy consumption results (Table 5) show that PIGA_CPU and PIGA_ACC record the lowest dynamic energy in most cases, reaffirming their suitability for energy-sensitive environments. HPIGA maintains a moderate position, balancing energy consumption and computational capability, and frequently achieving dynamic energy performance comparable to PIGA _CPU and PIGA_ACC. PIGA_HFP again ranks as the least energy-efficient, consistently exhibiting the highest dynamic energy consumption across all dimensions and functions. Overall, PIGA_ACC stands out as the most efficient algorithm in both total and dynamic energy terms, making it the best choice for applications where energy efficiency is critical. HPIGA offers a practical compromise between performance and energy use, while PIGA_HFP remains the most energy-intensive across all scenarios. These findings emphasize the importance of aligning algorithm design with appropriate hardware to optimize energy performance in large-scale optimization tasks. A notable finding from this comparison is that

Table 6
Mean execution time (seconds) for f_1 and f_2.

# of Islands	Algorithm	Dimensions under study									
		100		500		1000		3000		5000	
		f_1	f_2	f_1	f_2	f_1	f_2	f_1	f_2	f_1	f_2
128	PIGA_ACC	0.98	1.22	5.67	6.24	15.96	18.79	111.96	148.49	259.49	312.99
	PIGA_CPU	1.33	1.75	7.08	9.56	18.33	23.93	**96.95**	137.12	**205.44**	280.33
	PIGA_HFP	2.95	4.69	17.47	29.16	50.45	86.92	337.77	516.10	766.17	1303.21
	HPIGA	**0.51**	**0.65**	**3.22**	**3.94**	**13.90**	**17.07**	103.41	**124.86**	245.41	**275.82**
64	PIGA_ACC	0.98	1.31	5.67	7.73	15.56	18.50	107.23	131.42	259.54	361.28
	PIGA_CPU	1.48	2.25	7.78	10.80	19.57	27.21	115.89	161.10	**224.00**	318.94
	PIGA_HFP	2.96	4.83	17.35	27.24	48.79	74.10	322.30	553.16	772.89	1326.67
	HPIGA	**0.69**	**0.99**	**2.90**	**4.32**	**10.69**	**15.70**	**90.13**	**127.39**	235.29	**314.18**
32	PIGA_ACC	**0.95**	**1.16**	5.68	7.76	15.54	20.61	101.15	125.17	247.09	288.19
	PIGA_CPU	1.47	2.31	9.25	14.54	21.77	33.62	123.91	164.21	264.68	389.98
	PIGA_HFP	3.02	5.04	17.33	27.00	49.33	82.43	326.24	553.77	762.67	1268.39
	HPIGA	1.05	2.04	**2.92**	**4.35**	**10.87**	**15.83**	**90.58**	**124.13**	**151.57**	**221.69**
16	PIGA_ACC	**0.92**	**1.18**	5.66	7.51	15.60	19.00	100.46	**121.58**	228.90	299.34
	PIGA_CPU	1.48	2.31	8.88	12.59	25.17	38.08	134.47	193.97	295.10	453.67
	PIGA_HFP	2.79	4.48	17.43	28.07	49.78	85.44	328.64	518.14	739.55	1152.33
	HPIGA	1.44	1.84	**3.05**	**3.83**	**8.89**	**13.03**	**88.67**	129.46	**196.30**	**277.58**

Boldfaced and underlined values represent the least and highest values for each algorithm per function within each dimension, respectively.

search algorithms running on accelerators remain competitive with their multi-cores counterparts. For a comprehensive illustration of the performance behavior, we present the execution time of the parallel strategies in Table 6.

The execution time values reveal distinct behavior compared to the energy consumption values. As dimensionality grows, all algorithms experience a predictable increase in execution time. However, HPIGA demonstrates superior scalability, maintaining lower execution times even at high dimensions, showcasing its efficiency in handling large problem sizes through hybrid resource utilization. For the majority of instances under the study (32/40), HPIGA exhibits the lowest execution time (boldfaced values) among all the other strategies over the different dimensions and number of islands under the study. This behavior is quite prospective since this strategy considers distribute the workload by running the parallel algorithm over all the computing devices on the system. PIGA_ACC follows closely, particularly in lower dimensions where it competes well with HPIGA, reflecting the benefit of acceleration hardware for quick task execution. PIGA_CPU exhibits moderate performance, generally faster than PIGA_HFP but slower than both HPIGA and PIGA_ACC, especially as dimensionality increases. PIGA_HFP struggles significantly with scalability, as its execution time increases sharply with higher dimensions. This suggests potential bottlenecks caused by excessive communication overhead between the CPU and GPU devices.

The PIGA_HFP parallel strategy exhibits the highest energy consumption and execution times (underlined values) for all the dimensions under the study. This consumption behavior is expected due to the high communication overhead resulting from real-time data exchange between two different computing devices sharing computations in each iteration. The design of this strategy aimed to evaluate the feasibility of hybrid/shared operations between the CPU and GPU accelerator in each iteration. As anticipated, our results confirm the logical expectation that this approach incurs significant energy costs, making it impractical. In this context, we validate these perspectives and offer this conclusion as a reference point for future researchers. The tables also reveal an interesting accelerator energy consumption behavior compared to the multicore one. This outcome proves that the accelerators have competitive, promising results in solving parallel search algorithms. Overall, the HPIGA parallel strategy demonstrates competitive execution times and achieves a notable reduction in energy consumption.

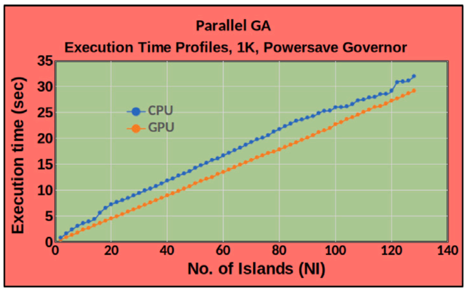
7.2. Experiment 2: Analysis of pareto fronts of dynamic energy vs. performance

This section outlines the methodology used to construct the execution time and dynamic energy profiles for processors running the hybrid parallel GA application. We investigate the bi-objective optimization problem concerning energy consumption and execution time, aiming to explore the Pareto fronts to determine the optimal workload distribution.

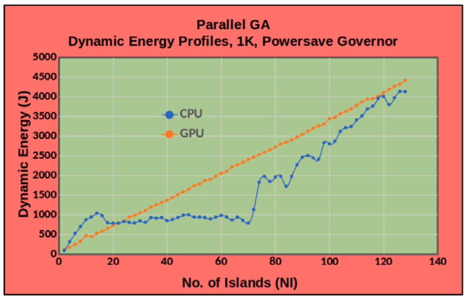
The application is executed under two governors: *powersave* and *performance*, allowing us to illustrate the trade-offs between dynamic energy consumption and performance for each governor. The choice of the powersave and performance governors was deliberate to ensure that voltage and frequency remained constant during execution. This stability is crucial for maintaining control over the execution environment, which is essential for accurately studying the bi-objective optimization problem. By preventing dynamic changes in voltage and frequency, we can isolate the impact of workload distribution on energy consumption and execution time, ensuring that the results reflect the true trade-offs between these two objectives.

To comprehensively evaluate the behavior of our algorithm, we conducted extensive experiments. These experiments involved running HPIGA on both the multicore CPU and one accelerator (A40 GPU) separately. We ran experiments with a growing number of islands, starting from 2 and increasing up to 128 in steps of 2 islands, to observe how the algorithm's performance and energy consumption change with the increasing number of islands. This approach allows us to understand the scalability and efficiency of the algorithm under different configurations and hardware settings, providing valuable insights into its performance dynamics. Fig. 6 illustrates the results of dynamic energy consumption and execution time as a function of the number of islands, for both 1000 and 5000 dimensions.

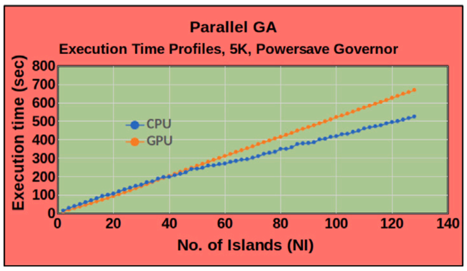
The results presented in the figures provide a comprehensive analysis of the dynamic energy profiles and execution times for a PGA application, comparing CPU and GPU implementations under different governors. In the powersave governor mode, the energy trend shows a notable difference compared to the maximum frequency scenario. Initially, the GPU energy consumption was lower than the CPU, but as the number of islands increases, the CPU eventually overtakes the GPU. This behavior indicates that while GPUs start with an energy efficiency advantage, the CPU's performance in powersave mode becomes more competitive as the computational load increases. This could be due to the power management features in the powersave governor optimizing the CPU's energy usage more effectively over time and across higher



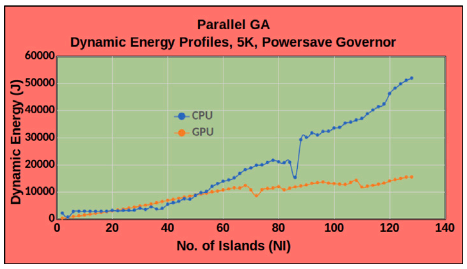
(a) Execution time against number of islands. Number of dimensions = 1K. The DVFS governor employed is powersave.



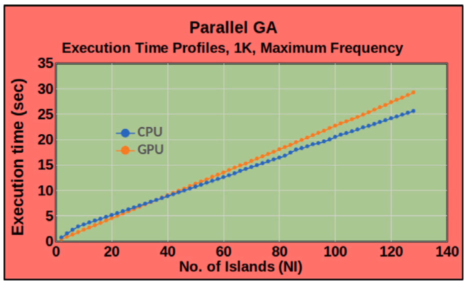
(b) Dynamic energy consumption against number of islands. Number of dimensions = 1K. The DVFS governor employed is powersave.



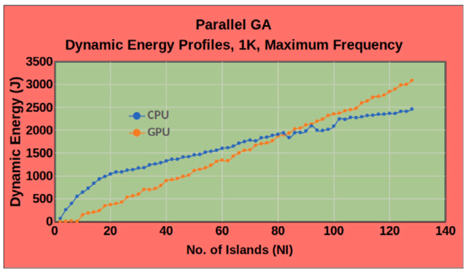
(c) Execution time against number of islands. Number of dimensions = 5K. The DVFS governor employed is powersave.



(d) Dynamic energy consumption against number of islands. Number of dimensions = 5K. The DVFS governor employed is powersave.



(e) Execution time against number of islands. Number of dimensions = 1K. The DVFS governor employed is performance.



(f) Dynamic energy consumption against number of islands. Number of dimensions = 1K. The DVFS governor employed is performance.

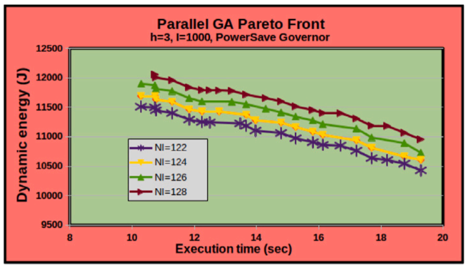
Fig. 6. The execution time and dynamic energy profiles of the multicore CPU and A40 GPU processors involved in the execution of the hybrid parallel GA application. Since the two A40 GPU accelerators are identical, only one profile is shown in each plot. The top four plots are for experiments where the multicore CPU employs powersave governor. The bottom two plots are for experiments where the multicore CPU operates at the maximum frequency (performance governor).

numbers of islands. CPU's energy consumption rises significantly with the number of islands, showing a steep increase, particularly beyond 60 islands. In contrast, the GPU maintains a relatively stable and lower energy consumption profile. This stark difference underscores the GPU's superior efficiency in handling larger datasets and the more extensive parallelism inherent in the 5K dimension setup. The CPU's energy consumption spikes could be attributed to increased context-switching overheads and less efficient scaling in powersave mode.

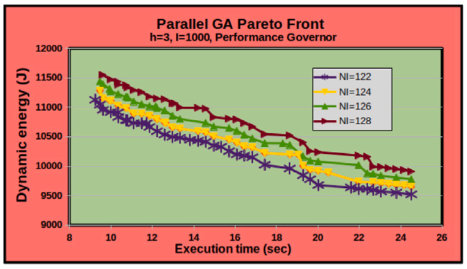
When conducting our experiments using the performance governor at maximum frequency, we observed that the GPU consistently uses less dynamic energy and has shorter execution times than the CPU as the number of islands increases. Initially, the GPU shows a more substantial energy advantage, which diminishes slightly as the number of islands increases but still remains below the CPU energy consumption. This suggests that GPUs are more energy-efficient for larger-scale computations in this setup, benefiting from their parallel processing capabilities which are well-suited for the workload distribution in a PGA.

Fig. 6 illustrates that, under both DVFS governors, GPU execution time scales almost linearly with the increasing number of islands. This behavior arises from the GPU's ability to launch vast numbers of lightweight threads in hardware, incurring constant per-kernel overhead and requiring minimal inter-thread synchronization. Consequently, the GPU maintains robust performance, particularly under the power-save governor and at lower island counts. By contrast, the CPU exhibits pronounced non-linear scaling caused by two factors. The first is the inherent complexity of modern multicore CPUs, including software-managed scheduling, multi-level cache hierarchies, NUMA-distributed memory, and shared interconnects. These introduce performance variability that disrupts uniform linear scaling. This factor is particularly influential in cases of lower computational intensity and is the dominant cause of non-linearity in the experiment with 1000 dimensions. The second factor is underutilization when the number of islands is less than the total number of available CPU cores (64). This becomes the dominant factor in more computationally intensive experiments with 5000 dimensions, especially under the powersave mode. As the number of islands increases and more cores become involved in computation, CPU utilization improves and performance increases. However, beyond a critical point, further increases in the number of islands lead to escalating synchronization delays and resource contention, which moderate further performance gains.

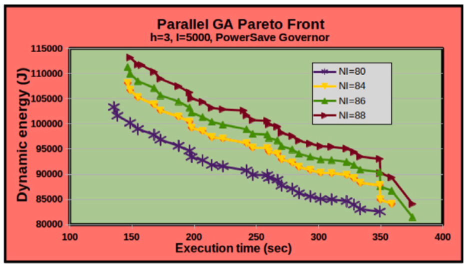
Overall, these results highlight a trade-off between energy efficiency and execution time when choosing between CPU and GPU for running parallel GA applications. GPUs are notably more energy-efficient, particularly for larger datasets and higher dimensions, but this comes at the cost of longer execution times in some configurations. The choice of governor mode also plays a significant role, with powersave governors



(a) Pareto fronts for the hybrid parallel GA application for powersave governor for I = 1000.



(b) Pareto fronts for the hybrid parallel GA application for performance governor for I = 1000.



(c) Dynamic energy versus performance Pareto fronts for the hybrid parallel GA application for powersave governor for I = 5000.

Fig. 7. Comparison of Pareto fronts for different governors and iteration limits in a parallel genetic algorithm: (1) Powersave with I = 1000, (2) Performance with I = 1000, and (3) Powersave with I = 5000.

potentially leveling the playing field between CPU and GPU energy consumption, albeit with varying impacts on performance scaling.

Fig. 7 shows the Pareto front results from our experiment under different governors. They illustrate the trade-off between dynamic energy consumption and execution time for various dimensions (I) and governor settings (powersave and performance). The decision variable is the workload distribution (vector of the number of islands assigned to the computing devices).

The presented figures illustrate multiple Pareto fronts, which address the bi-objective optimization problem for 1000-dimensional and 5000-dimensional configurations on the hybrid server. For the 1000-dimensional configuration, the experiments evaluated various numbers of islands (NI) in the set, {122, 124, 126, 128}. For the 5000-dimension configuration, the NI values are 80, 84, 86, and 88. The Pareto fronts provide a broad spectrum of options for optimal execution of the HPIGA on the accelerator server. The average number of solutions in the Pareto fronts is 25. The number of solutions employing the performance governor is greater than the number for the powersave governor. Consider the Pareto front for $NI = 122$ for the 1000-dimension configuration. The Pareto front provides 20 solutions (tradeoffs). Doubling the execution time only reduces the dynamic energy consumption by 10% utilizing the powersave governor. Similarly, the dynamic energy savings for the same NI value are 17% for the 5000-dimension configuration.

The figures continue to provide valuable insights into the optimization of dynamic energy and execution time, revealing several key trends. In the first two figures (7(a), 7(b)), comparing the powersave and performance governor settings for 1000 iterations, we observe that the performance governor generally achieves lower dynamic energy consumption for the same execution times compared to the powersave governor. This indicates that the performance governor is more effective in optimizing energy usage while maintaining computational speed. A similar trend is observed for the iteration limit set to 5000 (Fig. 7(c)) under the powersave governor. However, the longer execution times and higher iteration count lead to a more pronounced reduction in dynamic energy, suggesting that increasing the number of iterations allows for more refined optimizations. This also indicates the potential benefits of prolonged optimization processes in achieving better energy efficiency. The different values of NI (number of islands) also impact the results, with higher NI typically leading to better energy optimization but longer execution times. These trends emphasize the importance of selecting appropriate parameters for balancing energy efficiency and computational speed.

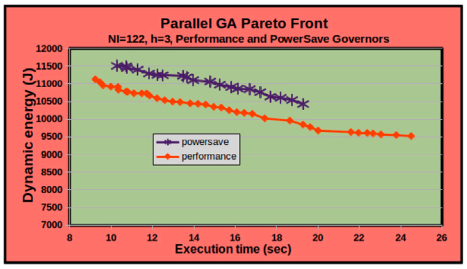
Fig. 8 illustrates the dynamic energy versus performance Pareto fronts for the hybrid parallel GA application using the performance and powersave governors, with 122 and 128 islands, respectively. These Pareto fronts represent the trade-off between minimizing execution time and reducing dynamic energy consumption. A comparative examination of both figures (8(a), 8(b)) reveals that the performance governor consistently achieves lower execution times at the cost of higher dynamic energy consumption compared to the powersave governor. As the number of islands increases, there is a noticeable trend where both execution time and energy consumption decrease. This reduction is more pronounced with the performance governor, indicating a more efficient use of resources.

Additionally, the distinction between the performance and powersave governors is evident in the clustering of data points, where the powersave governor shows higher energy usage for a given execution time. This suggests that while the powersave governor is designed to reduce power consumption, it does not optimize the trade-off as effectively as the performance governor under the tested configurations.

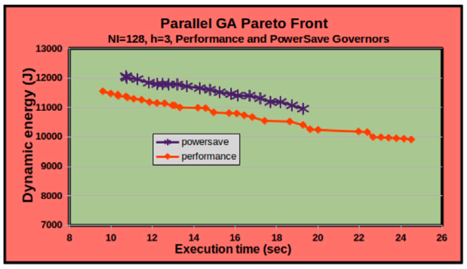
These results provide valuable insights into how different power management strategies affect the efficiency and scalability of PGAs on multicore systems. They highlight the need to carefully select the appropriate governor based on the specific performance and energy requirements of the application, with the number of islands playing a crucial role in determining the overall efficiency.

8. Conclusions and future work

This article presents two comprehensive experiments that explore the energy consumption and performance of PGAs on multi-core CPUs and accelerators using different parallelization strategies. The findings (including energy consumption, performance, Pareto front figures, and discussions) provide a valuable resource for future researchers to design efficient parallel search algorithms.



(a) Pareto front for number of islands, NI = 122, I = 1000.



(b) Pareto front for number of islands, NI = 128, I = 1000.

Fig. 8. Dynamic energy versus performance Pareto fronts for the hybrid parallel GA application for performance and powersave governors for $NI = 122$ and $NI = 128$, respectively.

For an approach to finding an adequate PGA strategy, we presented the first experiment. We compared four distinct parallel strategies running on CPUs and accelerators. Our discussions highlighted interesting performance and energy consumption profiles of the different algorithms, comparing the characteristics of energy consumption when running on CPUs and accelerators. The results serve as novel resources for researchers to select optimal configurations for designing and running intelligent energy consumption algorithms within hybrid heterogeneous systems.

To gain a clearer understanding, Experiment 2 was conducted with the application executed under two CPU and GPU frequency scaling governors: powersave and performance. The analysis reveals that while the GPU maintains consistent energy efficiency and linear execution scaling, CPU becomes increasingly competitive in powersave mode as the number of islands grows. These insights emphasize the dynamic interplay between architecture, workload size, and DVFS settings in optimizing PGA performance. The resulting data was utilized to plot Pareto fronts, illustrating the trade-offs between dynamic energy consumption and performance for each governor. This analysis elucidates the impact of the powersave and performance governors on optimizing the balance between computational efficiency and energy consumption.

Our experiments present innovative comparative studies that examine the performance and energy consumption of GAs on CPUs, accelerators, and hybrid servers. Through our analyses and discussions, we uncover interesting energy consumption profiles for various algorithms, highlighting their distinctive features when executed on CPUs and accelerators. To further enhance our findings, we supplement the results with comprehensive Pareto front graphs for the parallel GAs under examination. The insights derived from these studies serve as crucial benchmarks guiding the future utilization and advancement of efficient, energy-conscious optimization techniques across diverse computational devices. In future work, we plan to model energy consumption as a function of the parameters using regression models, such as random forests. We aim to extend our study by evaluating the impact of different CPU and GPU types and generations on performance and energy consumption.

CRediT authorship contribution statement

Amr Abdelhafez: Writing – review & editing, Writing – original draft, Visualization, Software, Methodology, Investigation, Formal analysis. **Ravi Reddy Manumachu:** Writing – original draft, Software. **Alexey Lastovetsky:** Writing – review & editing, Software, Methodology.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Amr Abdelhafez reports financial support was provided by Science Foundation Ireland Public Fellowship Programme. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This publication has emanated from research conducted with the financial support of Science Foundation Ireland (SFI) under the SFI Frontiers for the Future Programme 20/FFP-P/8683.

Data availability

Data will be made available on request.

References

[1] Michael C. Burkhart, Gabriel Ruiz, Neuroevolutionary representations for learning heterogeneous treatment effects, J. Comput. Sci. 71 (2023) 102054.
[2] Janko Straßburg, Christian Gonzàlez-Martel, Vassil Alexandrov, Parallel genetic algorithms for stock market trading rules, Procedia Comput. Sci. 9 (2012) 1306–1313, Proceedings of the International Conference on Computational Science, ICCS 2012.
[3] El-Ghazali Talbi, Amir Nakib, Metaheuristics for medicine and biology, Studies in Computational Intelligence, vol. 704, Springer, 2017.
[4] Abdullah Konak, David W. Coit, Alice E. Smith, Multi-objective optimization using genetic algorithms: A tutorial, Reliab. Eng. Syst. Saf. 91 (9) (2006) 992–1007.
[5] Ying Liu, Haibo Dong, Niels Lohse, Sanja Petrovic, A multi-objective genetic algorithm for optimisation of energy consumption and shop floor production performance, Int. J. Prod. Econ. 179 (2016) 259–272.
[6] Chunfeng Yang, Haijiang Li, Yacine Rezgui, Ioan Petri, Baris Yuce, Biaosong Chen, Bejay Jayan, High throughput computing based distributed genetic algorithm for building energy consumption optimization, Energy Build. 76 (2014) 92–101.
[7] Olivier Devos, Gerard Downey, Ludovic Duponchel, Simultaneous data preprocessing and SVM classification model selection based on a parallel genetic algorithm applied to spectroscopic data of olive oils, Food Chem. 148 (2014) 124–130.
[8] Vincent Roberge, Mohammed Tarbouchi, Francis Okou, Strategies to accelerate harmonic minimization in multilevel inverters using a parallel genetic algorithm on graphical processing unit, IEEE Trans. Power Electron. 29 (10) (2014) 5087–5090.
[9] Juan Porta, Jorge Parapar, Ramón Doallo, Francisco F. Rivera, Inés Santé, Rafael Crecente, High performance genetic algorithm for land use planning, Comput. Environ. Urban Syst. 37 (2013) 45–58.
[10] A. Shayeghi, D. Götz, J.B.A. Davis, R. Schäfer, R.L. Johnston, Pool-BCGA: a parallelised generation-free genetic algorithm for the ab initio global optimisation of nanoalloy clusters, Phys. Chem. Chem. Phys. 17 (2015) 2104–2112.
[11] Cheng-Jin Ye, Min-Xiang Huang, Multi-objective optimal power flow considering transient stability based on parallel NSGA-II, IEEE Trans. Power Syst. 30 (2) (2015) 857–866.
[12] Zhiyuan Liu, Qiang Meng, Shuaian Wang, Speed-based toll design for cordon-based congestion pricing scheme, Transp. Res. Part C: Emerg. Technol. 31 (2013) 83–98.
[13] Mohamed Kurdi, An effective new island model genetic algorithm for job shop scheduling problem, Comput. Oper. Res. 67 (2016) 132–142.
[14] Frédéric Pinel, Bernabé Dorronsoro, Pascal Bouvry, Solving very large instances of the scheduling of independent tasks problem on the GPU, J. Parallel Distrib. Comput. 73 (1) (2013) 101–110, Metaheuristics on GPUs.
[15] Rasoul Faraji, Hamid Reza Naji, An efficient crossover architecture for hardware parallel implementation of genetic algorithm, Neurocomputing 128 (2014) 316–327.
[16] Liucheng Guo, David B. Thomas, Ce Guo, Wayne Luk, Automated framework for FPGA-based parallel genetic algorithms, in: 2014 24th International Conference on Field Programmable Logic and Applications, FPL, 2014, pp. 1–7.

[17] Zhenyu Wen, Renyu Yang, Peter Garraghan, Tao Lin, Jie Xu, Michael Rovatsos, Fog orchestration for internet of things services, IEEE Internet Comput. 21 (2) (2017) 16–24.

[18] Message Passing Interface Forum, MPI: A message-passing interface standard, 2015.

[19] Simon Farrelly, Ravi Reddy Manumachu, Alexey Lastovetsky, OpenH: A novel programming model and API for developing portable parallel programs on heterogeneous hybrid servers, IEEE Access 12 (2024) 23666–23694.

[20] C.-N. Fiechter, A parallel tabu search algorithm for large traveling salesman problems, Discrete Appl. Math. 51 (3) (1994) 243–267.

[21] Stella C.S. Porto, João Paulo F.W. Kitajima, Celso C. Ribeiro, Performance evaluation of a parallel tabu search task scheduling algorithm, Parallel Comput. 26 (1) (2000) 73–90.

[22] Amr Abdelhafez, Enrique Alba, Gabriel Luque, Performance analysis of synchronous and asynchronous distributed genetic algorithms on multiprocessors, Swarm Evol. Comput. 49 (2019) 147–157.

[23] Amr Abdelhafez, Gabriel Luque, Enrique Alba, A component-based study of energy consumption for sequential and parallel genetic algorithms, J. Supercomput. 75 (10) (2019) 6194–6219.

[24] Tomohiro Harada, Enrique Alba, Gabriel Luque, A fresh approach to evaluate performance in distributed parallel genetic algorithms, Appl. Soft Comput. 119 (2022) 108540.

[25] Amr Abdelhafez, Enrique Alba, Gabriel Luque, Parallel execution combinatorics with metaheuristics: Comparative study, Swarm Evol. Comput. 55 (2020) 100692.

[26] Jan Gmys, Tiago Carneiro, Nouredine Melab, El-Ghazali Talbi, Daniel Tuyttens, A comparative study of high-productivity high-performance programming languages for parallel metaheuristics, Swarm Evol. Comput. 57 (2020) 100720.

[27] Jianyong Jin, Teodor Gabriel Crainic, Arne Løkketangen, A cooperative parallel metaheuristic for the capacitated vehicle routing problem, Comput. Oper. Res. 44 (2014) 33–41.

[28] Eugenio J. Muttio, Wulf G. Dettmer, Jac Clarke, Djordje Perić, Zhaoxin Ren, Lloyd Fletcher, A supervised parallel optimisation framework for metaheuristic algorithms, Swarm Evol. Comput. 84 (2024) 101445.

[29] E.Y. Seliverstov, A.P. Karpenko, Hierarchical model of parallel metaheuristic optimization algorithms, Procedia Comput. Sci. 150 (2019) 441–449, Proceedings of the 13th International Symposium "Intelligent Systems 2018" (INTELS'18), 22-24 October, 2018, St. Petersburg, Russia.

[30] Hung-Kai Wang, Yu-Chun Lin, Che-Jung Liang, Ya-Han Wang, Multi-subpopulation parallel computing genetic algorithm for the semiconductor packaging scheduling problem with auxiliary resource constraints, Appl. Soft Comput. 142 (2023) 110349.

[31] Yongbin Yu, Jiehong Mo, Quanxin Deng, Chen Zhou, Biao Li, Xiangxiang Wang, Nijing Yang, Qian Tang, Xiao Feng, Memristor parallel computing for a matrix-friendly genetic algorithm, IEEE Trans. Evol. Comput. 26 (5) (2022) 901–910.

[32] Guanghui Zhang, Bo Liu, Ling Wang, Dengxiu Yu, Keyi Xing, Distributed co-evolutionary memetic algorithm for distributed hybrid differentiation flowshop scheduling problem, IEEE Trans. Evol. Comput. 26 (5) (2022) 1043–1057.

[33] Wojciech Bożejko, Mariusz Uchroński, Mieczysław Wodecki, Parallel hybrid metaheuristics for the flexible job shop problem, Comput. Ind. Eng. 59 (2) (2010) 323–333.

[34] Wojciech Bożejko, Zdziław Hejducki, Mariusz Uchroński, Mieczysław Wodecki, Solving the flexible job shop problem on multi-GPU, Procedia Comput. Sci. 9 (2012) 2020–2023, Proceedings of the International Conference on Computational Science, ICCS 2012.

[35] Eyder Rios, Luiz Satoru Ochi, Cristina Boeres, Vitor N. Coelho, Igor M. Coelho, Ricardo Farias, Exploring parallel multi-GPU local search strategies in a metaheuristic framework, J. Parallel Distrib. Comput. 111 (2018) 39–55.

[36] Yanhong Zhuo, Tao Zhang, Feng Du, Ruilin Liu, A parallel particle swarm optimization algorithm based on GPU/CUDA, Appl. Soft Comput. 144 (2023) 110499.

[37] Mohamed A. Alqarni, Mohamed H. Mousa, Mohamed K. Hussein, Task offloading using GPU-based particle swarm optimization for high-performance vehicular edge computing, J. King Saud Univ. - Comput. Inf. Sci. 34 (10, Part B) (2022) 10356–10364.

[38] Manoj Kumar, Aryabartta Sahu, Pinaki Mitra, A comparison of different metaheuristics for the quadratic assignment problem in accelerated systems, Appl. Soft Comput. 100 (2021) 106927.

[39] Rui Zhang, Yanan Sun, Mengjie Zhang, GPU based genetic programming for faster feature extraction in binary image classification, IEEE Trans. Evol. Comput. (2023) 1–1.

[40] Jia Luo, Shigeru Fujimura, Didier El Baz, Bastien Plazolles, GPU based parallel genetic algorithm for solving an energy efficient dynamic flexible flow shop scheduling problem, J. Parallel Distrib. Comput. 133 (2019) 244–257.

[41] Martín Letras, Alicia Morales-Reyes, René Cumplido, María-Guadalupe Martínez-Peñaloza, Claudia Feregrino-Uribe, A novel partition strategy for efficient implementation of 3D cellular genetic algorithms, Microprocess. Microsyst. 104 (2024) 104986.

[42] Devrim Akgün, Pakize Erdoğmuş, GPU accelerated training of image convolution filter weights using genetic algorithms, Appl. Soft Comput. 30 (2015) 585–594.

[43] Mohammad Beheshti Roui, Mariam Zomorodi, Masoomeh Sarvelayati, Moloud Abdar, Hamid Noori, Paweł Pławiak, Ryszard Tadeusiewicz, Xujuan Zhou, Abbas Khosravi, Saeid Nahavandi, U. Rajendra Acharya, A novel approach based on genetic algorithm to speed up the discovery of classification rules on GPUs, Knowl.-Based Syst. 231 (2021) 107419.

[44] David Radford, David Calvert, A comparative analysis of the performance of scalable parallel patterns applied to genetic algorithms and configured for NVIDIA GPUs, Procedia Comput. Sci. 114 (2017) 65–72, Complex Adaptive Systems Conference with Theme: Engineering Cyber Physical Systems, CAS October 30 – November 1, 2017, Chicago, Illinois, USA.

[45] John Runwei Cheng, Mitsuo Gen, Accelerating genetic algorithms with GPU computing: A selective overview, Comput. Ind. Eng. 128 (2019) 514–525.

[46] Juan José Escobar, Pablo Sánchez-Cuevas, Beatriz Prieto, Rukiye Savran Kızıltepe, Fernando Díaz del Río, Dragi Kimovski, Energy–time modelling of distributed multi-population genetic algorithms with dynamic workload in HPC clusters, Future Gener. Comput. Syst. 167 (2025) 107753.

[47] Bruno Miguel Silva, Luiz Guerreiro Lopes, Fábio Mendonça, Parallel GPU-acceleration of metaphorless optimization algorithms: Application for solving large-scale nonlinear equation systems, Appl. Sci. 14 (12) (2024) 5349.

[48] Bruno Silva, Luiz Guerreiro Lopes, Fábio Mendonça, Multithreaded and GPU-based implementations of a modified particle swarm optimization algorithm with application to solving large-scale systems of nonlinear equations, Electronics 14 (3) (2025) 584.

[49] Jinpeng Han, Haobo Zhang, Baichuan Gao, Jingui Yu, Peng Jin, Jianzhong Yang, Zhaohui Xia, Efficient isogeometric topology optimization via multi-GPUs and CPUs heterogeneous architecture, Optim. Eng. 26 (2) (2025) 1317–1363.

[50] Mohd Arfian Ismail, A GPU accelerated parallel genetic algorithm for the traveling salesman problem, J. Soft Comput. Data Min. 5 (2) (2024) 137–150.

[51] Vincent Roberge, Katerina Brooks, Mohammed Tarbouchi, Parallel algorithm on multicore processor and graphics processing unit for the optimization of electric vehicle recharge scheduling, Electronics 13 (2024) 1783.

[52] Zhuoran Sun, Ying Ying Liu, Parimala Thulasiraman, Ruppa Thulasiram, Parallel co-evolutionary algorithm and implementation on CPU-GPU multicore, in: Proceedings of the Genetic and Evolutionary Computation Conference Companion, in: GECCO '24 Companion, Association for Computing Machinery, New York, NY, USA, 2024, pp. 109–110.

[53] El-Ghazali Talbi, Metaheuristics from Design to Implementation, John Wiley & Sons, 2009.

[54] Enrique Alba, Parallel Metaheuristics: A New Class of Algorithms, John Wiley & sons, 2005.

[55] Amr Abdelhafez, Gabriel Luque, Enrique Alba, Analyzing the energy consumption of sequential and parallel metaheuristics, in: 2019 International Conference on High Performance Computing & Simulation, HPCS, 2019.

[56] Amr Abdelhafez, Enrique Alba, Speed-up of synchronous and asynchronous distributed genetic algorithms: A first common approach on multiprocessors, in: 2017 IEEE Congress on Evolutionary Computation, CEC, 2017.

[57] Hao-Chun Lu, F.J. Hwang, Yao-Huei Huang, Parallel and distributed architecture of genetic algorithm on apache hadoop and spark, Appl. Soft Comput. 95 (2020) 106497.

[58] Tomohiro Harada, Enrique Alba, Parallel genetic algorithms: A useful survey, ACM Comput. Surv. 53 (4) (2020).

[59] Sourabh Katoch, Sumit Singh Chauhan, Vijay Kumar, A review on genetic algorithm: Past, present, and future, Multimedia Tools Appl. 80 (5) (2020) 8091–8126.

[60] Hamidreza Khaleghzadeh, Ravi Reddy Manumachu, Alexey Lastovetsky, Efficient exact algorithms for continuous bi-objective performance-energy optimization of applications with linear energy and monotonically increasing performance profiles on heterogeneous high performance computing platforms, Concurr. Comput.: Pr. Exp. 35 (20) (2023) e7285.

[61] H. Khaleghzadeh, M. Fahad, A. Shahid, R.R. Manumachu, A. Lastovetsky, Bi-objective optimization of data-parallel applications on heterogeneous HPC platforms for performance and energy through workload distribution, IEEE Trans. Parallel Distrib. Syst. 32 (3) (2021) 543–560.

[62] Francisco Chicano, Andrew M. Sutton, L. Darrell Whitley, Enrique Alba, Fitness probability distribution of bit-flip mutation, Evol. Comput. 23 (2) (2015) 217–248.

[63] M. Ruciński, D. Izzo, F. Biscani, On the impact of the migration topology on the island model, Parallel Comput. 36 (10–11) (2010) 555–571.

[64] Amr Abdelhafez, Ravi Reddy Manumachu, Alexey Lastovetsky, Code repository, 2024, csgitlab.ucd.ie/amra/parallel-genetic-algorithms-on-hybrid-servers.git. (Accessed 01 December 2024).

[65] Christoph Riesinger, Tobias Neckel, Florian Rupp, Non-standard pseudo random number generators revisited for GPUs, Future Gener. Comput. Syst. 82 (2018) 482–492.

[66] Muhammad Fahad, Arsalan Shahid, Ravi Reddy, Alexey Lastovetsky, A comparative study of methods for measurement of energy of computing, Energies 12 (11) (2019).