Parallel Computing 46 (2015) 98–119

Contents lists available at ScienceDirect

Parallel Computing

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

A scalable parallel genetic algorithm for the Generalized Assignment Problem

Yan Y. Liu [a,b,c,d,*], Shaowen Wang [a,b,c,d,e,f]

[a] CyberGIS Center for Advanced Digital and Spatial Studies, University of Illinois at Urbana-Champaign, Urbana, IL 61801, United States
[b] CyberInfrastructure and Geospatial Information Laboratory (CIGI), University of Illinois at Urbana-Champaign, Urbana, IL 61801, United States
[c] Department of Geography and Geographic Information Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, United States
[d] National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign, Urbana, IL 61801, United States
[e] Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, United States
[f] Department of Urban and Regional Planning, University of Illinois at Urbana-Champaign, Urbana, IL 61801, United States

ARTICLE INFO

Article history:
Received 13 February 2013
Received in revised form 4 December 2013
Accepted 23 April 2014
Available online 9 May 2014

Keywords:
Generalized Assignment Problem
Genetic algorithm
Heuristics
Parallel and distributed computing
Scalability

ABSTRACT

Known as an effective heuristic for finding optimal or near-optimal solutions to difficult optimization problems, a genetic algorithm (GA) is inherently parallel for exploiting high performance and parallel computing resources for randomized iterative evolutionary computation. It remains to be a significant challenge, however, to devise parallel genetic algorithms (PGAs) that can scale to massively parallel computer architecture (also known as the mainstream supercomputer architecture) primarily because: (1) a common PGA design adopts synchronized migration, which becomes increasingly costly as more processor cores are involved in global synchronization; and (2) asynchronous PGA design and associated performance evaluation are intricate due to the fact that PGA is a type of stochastic algorithm and the amount of computation work needed to solve a problem is not simply dependent on the problem size. To address the challenge, this paper describes a scalable coarse-grained PGA–PGAP, for a well-known *NP*-hard optimization problem: Generalized Assignment Problem (GAP). Specifically, an asynchronous migration strategy is developed to enable efficient deme interactions and significantly improve the overlapping of computation and communication. Buffer overflow and its relationship with migration parameters were investigated to resolve the issues of observed message buffer overflow and the loss of good solutions obtained from migration. Two algorithmic conditions were then established to detect these issues caused by communication delays and improper configuration of migration parameters and, thus, guide the dynamic tuning of PGA parameters to detect and avoid these issues. A set of computational experiments is designed to evaluate the scalability and numerical performance of PGAP. These experiments were conducted for large GAP instances on multiple supercomputers as part of the National Science Foundation Extreme Science and Engineering Discovery Environment (XSEDE). Results showed that, PGAP exhibited desirable scalability by achieving low communication cost when using up to 16,384 processor cores. Near-linear and super-linear speedups on large GAP instances were obtained in strong scaling tests. Desirable scalability to both population size and the number of processor cores were observed in weak scaling tests. The design strategies applied in PGAP are applicable to general asynchronous PGA development.

* Corresponding author at: Department of Geography and Geographic Information Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, United States. Tel.: +1 2172449315.
E-mail addresses: yanliu@illinois.edu (Y.Y. Liu), shaowen@illinois.edu (S. Wang).

http://dx.doi.org/10.1016/j.parco.2014.04.008
0167-8191/

1. Introduction

Inspired by natural selection, Genetic Algorithm (GA) represents a generic heuristic method for finding near-optimal or optimal solutions to difficult search and optimization problems [1]. GA mimics iterative evolutionary processes with a set of solutions encoded into a population at the initialization stage. Through GA operators (e.g., selection, crossover, mutation, and replacement) that are often stochastic, the population evolves based on the rule of "survival of the fittest" [2,3]. Such an evolutionary process stops when the population converges to solutions of specified quality. The computational challenges of GA are attributed to both problem-specific characteristics (e.g., problem 'difficulty' (e.g., *NP*-hard), problem size, the complexity of fitness function, and distribution characteristics of solution space specific to problem instances), and runtime efficiency of stochastic search [4].

High performance and parallel computing has been extensively studied to tackle the aforementioned computational challenges in GA as GA has inherent parallelism embedded in the evolutionary process [5]. For example, a population can be naturally divided into a set of sub-populations (also called demes) that evolve and converge with a significant level of independence. Various types of parallel genetic algorithms (PGAs) have been developed and broadly applied in a rich set of application domains [5–8]. More interestingly, previous work by Alba and Troya [9] showed that PGA computation not only improves computational efficiency over sequential GAs, but also facilitates parallel exploration of solution space for obtaining more and better solutions. In fact, Hart et al. [10] showed that running PGA even on a single processor core outperformed its sequential counterpart. Therefore, PGA is often considered and evaluated as a different algorithm rather than just the parallelization of its corresponding sequential GA.

This paper describes a scalable PGA (PGAP) to exploit massively parallel high-end computing resources for solving large problem instances of a classic combinatorial optimization problem – the Generalized Assignment Problem (GAP). GAP belongs to the class of *NP*-hard 0–1 Knapsack problems [11–13]. Numerous capacity-constrained problems in a wide variety of domains can be abstracted as GAP instances [14] such as the job-scheduling problem in computer science [15] and land use optimization in geographic information science and regional planning [16]. Various exact and heuristic algorithms have been developed to solve GAP instances of modest sizes [17]. However, in practice, problem instances often have larger sizes while the problem solving requires quick solution time and the capability for finding a set of feasible solutions of specified quality, which compounds the computational challenges.

Our PGA approach focuses on the scalability to massively parallel processor cores (referred to as cores hereafter) available from high-end computing resources such as those provided by the National Science Foundation XSEDE [18] cyberinfrastructure. PGAP is a coarse-grained steady-state [19] PGA that searches solution space in parallel based on independent deme evolution and periodical migrations among connected demes. Scalability is a key to efficiently exploiting a large number of cores in parallel. Previous PGA implementations mostly rely on synchronization to coordinate parallelized operations (e.g., the migration operation in coarse-grained PGAs and the selection operation in fine-grained PGAs), primarily because the computation of PGA is an iterative process and it is straightforward to implement iteration-based synchronization. Synchronization is often needed at two places: (1) waiting for all PGA processes to rendezvous before migration operations; and (2) using synchronous communication to exchange data. While success on scaling PGA to many cores has been achieved based on hardware instruction-level synchronization supported by SIMD architectures [20], we argue that the computational performance of PGA is under-achieved through synchronizing iterations across massively parallel computing resources with MIMD architecture [21].

Therefore, an asynchronous migration strategy is designed to achieve scalable PGA computation through a suite of non-blocking migration operators (i.e., export and import) and buffer-based communications among a large number of demes connected through regular grid topology. The asynchrony of migration is effective to not only remove the costly global synchronization on deme interactions, but also allows for the overlapping of GA computation and migration communication. Addressing buffer overflow issues caused by inter-processor communications and understanding their relationship to the configuration of asynchronous PGA parameters are crucial to design scalable PGA on high-end parallel computing systems. Through algorithmic analysis on PGAP, we identified two buffer overflow problems in exporting and importing migrated solutions, respectively. In export operations, the overflow of the outgoing message buffer used by the underlying message-passing library may cause runtime failure and abort the PGA computation. In import operations, the overflow of the import pool maintained for receiving solutions from neighboring demes may cause the loss of good solutions. Through algorithmic analysis, we derive two conditions to guide the setting of PGAP parameters, including migration parameters, topology, and buffer sizes based on the underlying message passing communication library in order to detect and/or avoid aforementioned buffer overflows. To the best of our knowledge, our work is the first to explicitly consider the relationship between the configuration of asynchronous PGA parameters and underlying system characteristics so as to improve the reliability of asynchronous PGAs.

Experiment results showed that, with the asynchronous migration strategy, our scalable PGA is able to efficiently utilize 16,384 cores with significantly reduced communication cost. Specific strong and weak scaling tests were designed to evaluate the scalability and numerical performance of PGAP because conventional weak and strong scaling methods are not directly applicable to PGA. For example, the problem size used in conventional weak scaling is not a good indicator of the amount of computation needed to achieve a certain solution quality in PGA. PGA and sequential GA also have different

algorithmic behaviors. More importantly, population size plays a crucial role in PGA performance as the number of cores increases. Therefore, instead of problem size, we chose population size to study PGAP scalability and compared its performance with the corresponding sequential algorithm. In strong scaling tests, linear and super-linear speedups were observed in solving large GAP instances. PGAP also outperforms the best-so-far sequential GA in solving modest problem instances. Numerical performance study on large instances showed promising results on the capability of PGAP to exploit massive computing power for more effective exploration of search space for finding alternative solutions, faster convergence, and obtaining improved solution quality.

The reminder of the paper is organized as follows. Section 2 reviews related work on PGA and GAP. Section 3 describes the design and implementation of PGAP for solving computationally intensive GAP instances. Section 4 details computational experiments designed for evaluating PGAP scalability and numerical performance and analyzes experiment results. Section 5 summarizes the findings of the research and discusses future work.

2. Related work

The Generalized Assignment Problem (GAP) is defined to find an optimal assignment of n items to m bins (knapsacks) such that the total cost of the assignment is minimized and the weight capacity constraint of each bin is satisfied. The cost and weight of an item depend on both the item itself and the assigned bin [13]. GAP belongs to the class of *NP*-hard 0–1 Knapsack problems [12]. It is the generalization of the well-known Knapsack problem (single bin) and Multiple Knapsack Problem (MKP) [13]. A detailed survey of sequential algorithms for solving GAP can be found in [17]. Canonical work on GAP heuristic algorithms include genetic algorithms [22–26], simulated annealing [27], tabu search [28], and hybrid search that combines heuristics and local search strategies [29–31]. Previous work on GA approach to GAP showed that standard GA operations (i.e., selection, crossover, mutation, and replacement) often produce infeasible solutions that violate the capacity constraint of GAP, therefore needs additional processing. For example, Chu and Beasley [22] developed two additional operators, feasibility improvement and quality improvement, to convert infeasible solutions to feasible ones and attempt to improve each resulted feasible solution, respectively. Wilson's method [23] allows for infeasible solutions in GA computation, but improves the feasibility of all solutions after GA stops. The method by Feltl and Raidl [24] also allows for infeasible solutions, but with a penalty-based mechanism for adjusting the fitness value of infeasible solutions.

As a well-known approach to increasing the computational efficiency of GA, PGAs differ in where parallelization is exploited: *fine-grained* PGAs [32] (also referred to as *cellular GA (cGA)* or *Diffusion Models* [33]) parallelize the selection operator to select parents from directly connected neighbors on a PGA topology in each iteration; *coarse-grained* PGAs (also referred to as *island model GA (iGA)*) migrate a portion of local solutions to connected demes periodically [34]; *global parallelization*-based PGAs parallelize the computation of the fitness evaluation function if the function is computing intensive [35]. PGA surveys can be found in literature [5,9,36–39]. Tanese [40] showed that a PGA exhibits different algorithmic behavior than its corresponding sequential GA. Interestingly, even without communications, PGA could improve GA performance with independent deme evolution. This is because multiple independent running instances of the same GA likely produce different quality results because of different randomness introduced by each instance for GA operations. Given a fixed solution quality requirement, the probability that at least one instance finds a solution of designated quality earlier would increase as more cores are used. Even without communications among demes, therefore, running multiple demes on a single core often performs better than running a single deme with the same global population size. On the other hand, communications, i.e., migration of a portion of local solutions to other demes, greatly improves the effectiveness of GA computation through the propagation of good solutions and the injection of new randomness.

Previous work on designing scalable PGAs often employs the fine-grained PGA model on SIMD architecture [20,41–46]. For example, Shapiro et al. [20] implemented a PGA for RNA folding on the MasPar SIMD supercomputer with 16,384 processing units. Chen et al. [41] developed a hybrid cellular GA on the same machine. These fine-grained PGA implementations exhibit good performance on gene diffusion, i.e., the propagation of high-quality genes across the entire population. Hart et al. [10], Alba and Troya [19], and Prabhu et al. [45] pointed out that SIMD architecture is suitable for the development of fine-grained PGAs because fine-grained PGAs require synchronization in each iteration to select parents from directly connected demes and SIMD systems provide hardware- and/or system-level synchronization support that can be directly adopted to coordinate the selection operation within each iteration. It is more challenging to synchronize GA iterations on MIMD systems in which processors are often connected through network. On MIMD architecture, synchronization can be a costly task at system level, especially when it involves a large number of connected cores. In message passing models which MIMD systems often use for communications, the overhead of synchronization has been well studied [47–48]. Such synchronization cost can cause serious performance degradation on PGAs. In the synchronous mode, a PGA iteration can be considered as a sequence of computations followed by a global barrier to rendezvous all processes. Any delay at one process, caused from various sources (e.g., network, operating system, memory, I/O, or computation itself in GA) is propagated to all of the participating processes, and the probability and duration of such delay increases as the number of processes increases. In fact, the evolutionary process in PGA may not require synchronization because: (1) a PGA process can evolve independently; (2) communications with other processes only involve neighboring processes, without the need for global-level synchronization; and (3) sending information to and receiving information from neighboring processes do not need to be coupled.

In general, GA, as a type of stochastic computing model, exhibits massive parallelism and holds tremendous potential to reap the benefits of large-scale parallel computation [49]. As massively parallel computing resources become available [21,50] as a consequence of active development of multi-core/many-core computing and extreme-scale supercomputing [51], it is promising to harness an unprecedented amount of parallel computing resources for scalable GA computation.

It has been recognized that asynchronous inter-deme interactions are desirable for designing scalable PGAs [10,19,52,53]. Most of previous PGA work chose to synchronize GA computation primarily for the purpose of straightforward parallel programming implementation. Early work by Hart et al. [10] showed that delays exhibited in asynchronous fine-grained PGAs helped achieve better solution quality because some demes were allowed more time for the local evolution to be more convergent toward better solutions. Lin et al. [53] observed the effectiveness of exchanging solutions asynchronously and proposed a solution exchange strategy based on population similarity instead of the fixed connection topology on coarse-grained PGAs. The CAGE framework [54] employed non-blocking message passing functions for the development of fine-grained GA programming library on MIMD clusters configured with up to 256 computing nodes. We argue that the availability of massively parallel computing resources now offers a major driver toward the design and development of asynchronous PGAs. In this paper, an asynchronous migration strategy is developed as the mainstay for enabling scalable PGA for GAP.

As a complex and dynamic process, PGA computation is sensitive to a set of algorithmic and computational factors. Alba and Troya [19] and Hart et al. [10] pointed out that conventional parallel computing performance measures such as speedup and efficiency, need careful consideration when applied to computational performance evaluation of PGA. Since PGA and GA are algorithmically different, direct performance comparison between PGA and sequential GA may not lead to appropriate conclusions. Alba and Troya [19] further suggest using the same parallel version on different number of cores for speedup measurement. Also, a sufficient number of trials should be conducted to yield statistically confident evidence on obtained results [10,37]. Identifying the sources of performance variation in asynchronous PGA is another issue. Gordon and Whitley [55] observed that even on a single core, emulated PGAs were often more efficient than many sequential GAs for solving various types of problems. In addition, injected computational noise changes asynchronous PGA algorithmic performance [10], making performance measurement more complicated.

In high-performance computing, weak and strong scaling are two typical ways to measure scalability of high-performance computing algorithms [56,57]. As for weak scaling, problem size per core is kept constant as the number of cores increases. Therefore, weak scaling allows us to look at the capability of an algorithm to solve larger or more complicated problems in conjunction with the use of more cores. Strong scaling keeps the overall problem size a constant as the number of cores varies, and measures speedup, calculated by dividing the execution time of the best sequential algorithm by that of the parallel algorithm. However, neither of them can be directly applied to analyze the scalability of PGAs for solving combinatorial optimization problems (elaborated in 4.2). This research, therefore, has designed specific scaling test methods for the comprehensive evaluation of the performance of PGAP.

3. Algorithm

The mathematical formulation of GAP is as follows:

$$\text{Objective}: \quad min \sum_{i=1}^{m}\sum_{j=1}^{n} c_{ij}x_{ij}$$

$$such\ that:$$

$$\sum_{j=1}^{n} w_{ij}x_{ij} \leqslant b_i, \quad i = 1,2,\ldots,m \tag{1}$$

$$\sum_{i=1}^{m} x_{ij} = 1, \quad j = 1,2,\ldots,n \tag{2}$$

$$x_{ij} \in \{0,1\}, \quad i = 1,2,\ldots,m,\ j = 1,2,\ldots,n$$

Where matrix $C_{m\times n}$ and $W_{m\times n}$ denote the cost and weight requirements of assigning each item $j,\ j = 1,2,\ldots,n$, to each bin $i,\ i = 1,2,\ldots,m$, respectively. Constraint (1) represents the weight capacity constraint indicating that the total weights of the items assigned to a bin cannot exceed the bin's capacity. $X_{m\times n}$ is the assignment matrix with each entry valued either 0 or 1. Constraint (2) is the assignment constraint that allows an item to be assigned to exactly one bin. Each problem instance has the cost and weight matrices as input. The output is the assignment matrix $X_{m\times n}$ that minimizes the cost defined in the objective function.

3.1. Sequential genetic algorithm

The sequential GA developed for this GAP formulation is an extension to the GA developed by Chu and Beasley [22]. A solution is encoded as a chromosome that is represented as a binary string s of size n, where the value of s_j denotes the index of the bin to which item j is assigned (Fig. 1). A population is formed by a set of encoded solutions. Population size is a run-time parameter. This GA follows a *steady-state* [3] reproduction process in which each iteration selects two parents to

generate one child for replacement. The GA operators to be performed within each iteration are adapted from Chu and Beasley [22] and listed below for the completeness of this paper:

- *Selection*. Two parents are selected based on selection strategies such as binary tournament selection or rank-based tournament selection [58].
- *Crossover*. The simple single cut-point crossover operator randomly decides a cut point at which the first part of a parent is combined with the second part of the other to form the child solution.
- *Mutation*. The mutation operator is performed on the child solution obtained from the crossover operator to exchange the bin assignment of two randomly chosen items.
- *Feasibility improvement*. This operator looks at those over weighted bins after fitness evaluation for a reassignment that could keep the weight sum of the bin below its capacity. This is done by moving an item in an over weighted bin to an under weighted bin.
- *Quality improvement*. This operator looks at each item for a possible reassignment (to another bin) such that the solution's fitness value could be improved.

The fitness of a solution is evaluated as two values: fitness value that is equivalent to the value of GAP objective function, and 'unfitness' value which is the sum of exceeded weights in each bin. For feasible solutions, the unfitness value is always zero. Replacement strategies use the unfitness value to choose the solutions to be replaced. The execution of GA is stopped if its stopping criterion is met. The stopping criterion can be a fixed number of iterations, a time limit, or a given solution quality threshold.

Chu and Beasley [22] and Feltl and Raidl [24] indicate that the feasibility and quality improvement operators are critical to keep the search close to feasible regions in the solution space and help GA converge to near-optimal or optimal solutions quickly. We further refine these two operators for better lookup efficiency. Both operators include a linear scan of bins to look up for either over weighted bins for feasibility improvement or an under weighted bin for reassigning an item to improve solution quality. In [22], the order in which bins are checked is fixed, which means that the capacity of each bin is also examined in a fixed order. Considering capacity variations among bins, such examination with the fixed order may limit the search efficiency for possible candidate choices for item reassignment. For example, bins checked first are always considered for reassignment first despite of the existence of alternatives. Such limitation becomes significant in PGA, as the size of global population is often large. In our algorithm, the bin lookup sequence is randomly decided to make the bin lookup operation independent of the order of bins coded in problem instances. The performance impact of this technique is discussed in Section 4.

3.2. Parallel genetic algorithm

Our coarse-grain PGA–PGAP – is designed based on asynchronous migration because: (1) synchronizing each GA iteration is not necessary in a coarse-grained PGA; (2) the overhead of synchronization gets worse when more cores are used (see Section 2); and (3) non-blocking migration operators increase the overlapping of computation and communication (local evolution can go to the next iteration without waiting for response messages from receiving demes before the next round of migration) and, thus, may significantly improve the computational performance of PGAP.

In PGAP, each deme initializes and maintains a local population. The size of a local population is referred to as deme size. The number of demes is determined based on the number of cores available at runtime. Demes are connected with a regular 2-D toroidal grid topology (Fig. 2). On the grid, the start and end of a row/column are connected. Therefore, the connectivity degree, d, is four for each deme. The algorithm runs concurrently on all cores until a stopping criterion is satisfied at any of the cores.

The following PGA operators are designed to exchange a portion of a deme's population with its directly connected neighbors.

Export. A deme exports a fixed number of solutions to its neighbors periodically. The export operator defines two parameters: migration rate r (i.e., the number of solutions to be exported) and export interval M_{expt} (defined as the number of

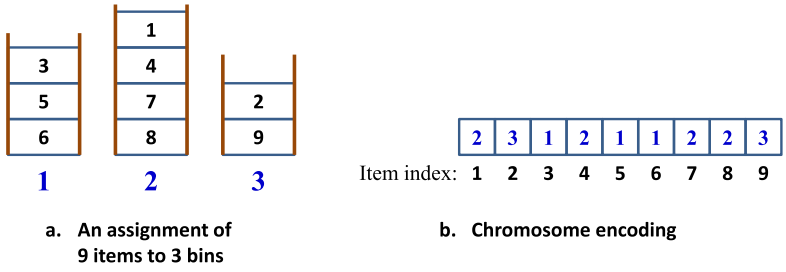


Fig. 1. GAP encoding.

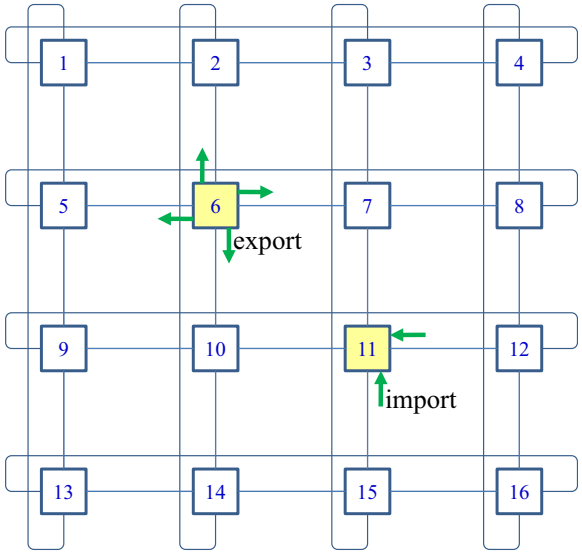


Fig. 2. Topology of PGAP.

iterations). Our strategy for exporting solutions considers both elitism and diversifying deme population by exporting random solutions. When choosing r solutions for exporting, elite solutions generated during the previous export interval are always included. Remaining spots, if any, are filled by randomly picked solutions from local population. If no elite solutions were found in the previous export interval, a holding strategy is applied to probabilistically delay the export.

Import. Each deme maintains an import pool, implemented as a cyclic first-in-first-out (FIFO) queue with the queue header pointing to the first imported solution and queue tail pointing to the next available buffer space, to hold external solutions migrated from neighboring demes. A parameter, import interval M_{impt} (defined as the number of iterations), is used to control how often an import operation is performed. An import operation copies all incoming solutions from the underlying communication system to the import pool. When the pool is full, new incoming solutions override older ones. The size of the import pool is determined by a buffer management method described later (see Section 3.3) to avoid this overriding scenario.

Inject. The inject operator merges migrated solutions from the import pool into local population. Elite solutions from neighboring demes are alway incorporated into local population if they are superior than local elite solutions. Random solutions from outside are considered as candidates to be selected as one of the two parents for standard GA operations.

Fig. 3 illustrates above PGA operators and their interactions with local GA and network. The pseudo code of PGAP is provided in Fig. 4. The introduction of PGA operators affect the selection and replacement operators in the sequential algorithm. After the inject operator is performed, the selection operator takes an injected random solution as one of the two parents to

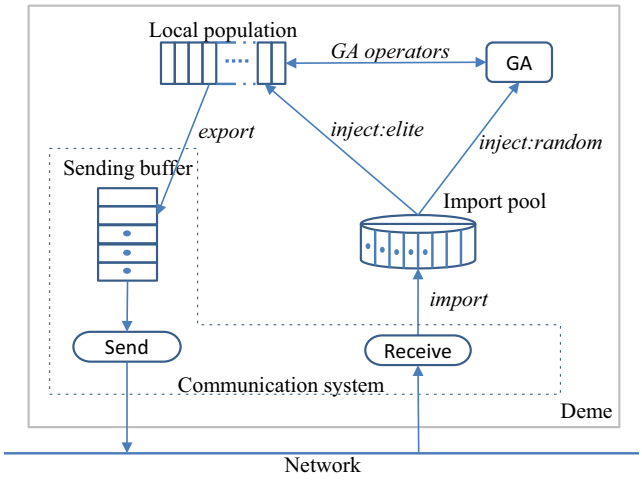


Fig. 3. PGAP operators and their interactions.

subsequent GA operators. The replacement operator directly merges a remote elite solution into local population. By doing so, local population at a deme is able to evolve with better solutions already found by other demes, therefore saving local computation to reach the same level of solution quality. Furthermore, combined with the selection strategy in the export operator for sending out local elite solutions, good solutions can be propagated to other demes in a hop-by-hop fashion until they are overtaken by better solutions. This is similar to the diffusion model in fine-grained PGAs [59]. The reason to export random solutions is that these solutions appear as noise to local evolution on the receiving deme and may influence local

```
/* On each deme */:
Individual P[d]; // deme population of size d
Individual p1, p2, c; // parent 1 and 2, and child
Individual Pool[bufsize]; // import buffer
Generate and evaluate initial population P;
iter := 0;
do /* evolutionary iteration */
    c := ∅;
    select p1 and p2 from P using binary tournament selection; // selection
    if (there are new solutions in Pool) // migration: inject
        t := the first unprocessed solution in Pool;
        if (t is an elite solution from a neighboring deme)
            c := t; // directly-inject this elite solution into local deme
        else // a random solution from neighbor
            p2 := t; // select as a parent
    if (c is empty)
        cutpoint := rand(); // randomly select a cutoff point for p1 and p2;
        c := combine(p1, p2, cutpoint); // crossover
        /* mutation: swap bin assignment of two randomly selected items */
        c := swap(c, rand(), rand());
    get a random lookup order O;
    foreach bin in O // feasibility improvement on c
        calculate weight sum for bin in c;
        if (bin is overweighted)
            try to move one item from bin to another
            such that both bins are under capacity;
    foreach item in c // quality improvement
        try to move the item to another bin such that
        the fitness value of c will be less;
    calculate the fitness value for c; // fitness evaluation
    if (c is feasible and unique) // replacement
        if (there exists an individual with unfitness value > 0)
            replace the unfit individual;
        else replace the individual with the worst fitness value;
    if (iter % M_expt = 0) // migration: export
        if no improvement since last export and should hold
            skip this export operation;
        else wait until previous export operation is finished;
            select and send r solutions to direct neighbors;
    if (iter % M_impt = 0) // migration: import
        probe incoming solutions;
        foreach new incoming solution insoln
            Pool[++ bufhead ] := insoln;
    iter++;
until stopping rule is met;
```

Fig. 4. Parallel genetic algorithm for GAP.

evolution paths in a positive way [10]. When a local evolution process is trapped in local optima or stays within a premature state, such 'noise' from outside may help jump-start the evolution process to search new solution space, therefore providing the diversification effect for local evolution. On the other hand, the probability of holding is kept sufficiently low to avoid excessive interference on any receiving deme's evolution process.

Compared to synchronous PGA design in which all demes need to stop for communication within the selection operator (in the case of fine-grained PGA) or export and import operators (in the case of coarse-grained PGA), there is no global communication barrier for migration-related communication in PGAP. On each deme, an export operation starts every M_{expt} iterations, returns immediately without waiting for acknowledgement messages from receiving demes, and completes when exported solutions are passed to the underlying communication system. The import operator on each deme checks for incoming solutions every M_{impt} iterations. If no incoming solutions are found, the deme proceeds to the next iteration without waiting. The check operation, called probe, is lightweight. The inject operator injects imported solutions from the import pool into the local evolution process, one at a time. The benefit of using asynchronous migration is two-fold as follows. Globally, asynchronous migration eliminates the costly global coordination among all demes. Locally, the overlap between computation and communication, enabled by the non-blocking export and import operators and buffer-based export communication, increases significantly to allow for better computational performance. Fig. 2 illustrates a runtime topology of PGAP on which migration events and computation take place at the same time.

3.3. Buffer management for asynchronous migration

Buffer-based migration was used before for developing asynchronous PGAs. For example, Andre and Koza [60] implemented an asynchronous PGA by separating export, import, and other GA operators as independent processes and used per-neighbor buffers to receive exported solutions from corresponding neighbors. To the best of our knowledge, no previous work has considered the issue of buffer overflow and its relationships with PGA parameters. In PGAP, buffer-based communication and non-blocking communication functions are applied together to implement the asynchronous migration strategy and achieve desirable overlapping of computation and communication in export and import. Buffer is used at two places: (1) the import pool is implemented as a cyclic buffer; and (2) PGAP allocates memory to the underlying communication system as outgoing message buffer (referred to as sending buffer hereafter) for buffer-based non-blocking export operation. Buffer overflow at both places must be considered and avoided whenever possible to make PGAP reliable. The overflow of the import pool will override solutions received in previous import operations, and thus cause the loss of imported neighbor solutions if they have not been injected yet. The overflow of the sending buffer has more severe impact: the underlying message-passing library often terminates the execution of all processes even when the overflow occurs within one process. We argue that inappropriate configuration of PGA parameters may lead to serious buffer overflow issues. An algorithmic analysis is conducted on PGAP to study the two buffer overflow issues caused by communication delays. Two algorithmic conditions are then derived to detect and avoid buffer overflow issues by setting PGAP parameters appropriately.

PGAP uses message passing (specifically, MPI [61]) as underlying parallel programming model and implements the buffer-based non-blocking communication as follows, which is a common choice suggested in [61]:

- Message send operation completes without the need to wait for the post of matching receive at destination process. Instead, the sending operation is considered complete after the message is handed to the sending buffer. Send operation is implemented as non-blocking function.
- The buffered send operation calls a non-blocking non-buffered send for sending each message queued in the buffer, which will not complete until the matching receive is posted or the receiving process starts receiving. This is suggested by MPI standard (see chapter 3 in [61]).
- The sending buffer is managed by the underlying communication system, but the memory of this buffer is allocated explicitly by PGAP.
- Before the message receive operation starts, a probe operation is called to check new messages. Probing introduces negligible cost. If probing does not find new messages, the receive operation is postponed. A receive is considered complete when an incoming solution is copied into the import pool. The receive operation is implemented as blocking function.
- A import operation, if called, receives all of incoming solutions into the import pool.

Table 1 lists the PGAP parameters considered in our analysis of buffer overflow. Our analysis is based on a common PGA execution environment where each deme has the same setting for PGAP parameters listed on Table 1. This analysis focuses on the overflow scenarios caused by communication delays among demes. For simplicity purpose, the holding strategy mentioned in Section 3 is not considered. Computing time for an iteration is assumed to be consistent across all demes. This assumption indicates that each participating core may have similar CPU and memory characteristics, which is the case for the supercomputers we used for experiments.

3.3.1. Sending buffer overflow analysis

The sending buffer overflows when there are too many pending send operations to handle (Fig. 3). In PGAP, communication delays in the underlying communication system and/or a long import interval (M_{impt}) on receiving end can prevent a send operation from being complete. Communication delays are caused by network congestion or communication system

Table 1
PGAP parameters considered in the algorithmic analysis.

d	Connectivity: the number of directed connected neighboring demes
r	Migration rate: the number of local solutions selected for each export operation
M_{expt}	Export interval: the number of iterations between two consecutive export operations
M_{impt}	Import interval: the number of iterations between two consecutive import operations
$K_{sendbuf}$	Size of the sending buffer allocated as the system outgoing message buffer. $K_{sendbuf} \geqslant r$
K_{impt}	Size of the import pool

resource limitations, which we have no control of in algorithm design. Allocating a large $K_{sendbuf}$ helps, but does not guarantee the avoidance of overflow. The purpose of our analysis is to avoid the overflow scenarios caused by inappropriate configuration of PGAP parameters.

We consider the worst case scenario in which no send operation can be completed. Suppose it takes x iterations for export operations to fill the sending buffer with $K_{sendbuf}$ solutions. The rate of buffer filling is $\frac{r}{M_{expt}}$ per iteration. By letting $x \times \frac{r}{M_{expt}} = K_{sendbuf}$, we have $x = \frac{K_{sendbuf}}{r} \times M_{expt}$. If M_{impt} at receiving deme is set to be less or equal to x, we can then avoid the sending buffer overflows caused by inappropriate configuration of PGAP. Equivalently, this condition can be written as:

$$\frac{M_{impt}}{M_{expt}} \leqslant \frac{K_{sendbuf}}{r} \tag{1}$$

$K_{sendbuf}$ is usually fixed for a PGAP run. Condition (1) can then be used to guide the appropriate configrations of M_{impt}, M_{expt}, and r.

3.3.2. Import pool overflow analysis

If the solutions in the import pool of a deme are not injected into local population in time because of computational noise or outpaced communication between the deme and its neighbors, the pool may overflow. In this overflow situation, one strategy is to allow solution overriding. Such overriding is 'harmless' if new solutions are elite solutions from the same neighbor because a newly arrived elite solution always has equal or better quality. However, since we allow random solutions to be exported and elite solutions from different neighbors may have different quality, overriding current solutions in the import pool may result in overriding better solutions that have not been injected into local population. Sophisticated methods can be developed for import pool management to handle such overriding, but at the price of slowing down the processing of the import operator. By looking at overflow scenarios and their relationship to PGAP parameters, a sufficient condition is derived to guide the configuration of K_{impt}, M_{expt}, d, and r in order to avoid the overflow of the import pool.

The import pool will overflow if and only if:

1. The rate of producing solutions to the pool is faster than the rate of consuming from it, if we consider the import operator as the producer and the inject operator as the consumer; or
2. The import pool is not large enough to hold imported solutions received in a single import operation.

The first condition means if the rate of import operations (in iterations), denoted by R_{impt}, is larger than the rate of inject operations (in iterations), denoted by R_{inject}, the import pool will overflow eventually, regardless of the size of the pool (K_{impt}). Let us denote the event of import pool overflow as O, the number of received solutions in a single import operation as k_{impt}, and the number of solutions remained in the import pool at the beginning of an iteration as k. Therefore, when an import operation is complete, there are $k + k_{impt}$ solutions in the import pool. From the notations defined in Table 1, Above overflow conditions can be written equivalently as:

$$A : R_{impt} \leqslant R_{inject},$$
$$B : k + k_{impt} \leqslant K_{impt},$$
$$A \wedge B \leftrightarrow \neg O$$

The left part ($A \wedge B$) is the sufficient and necessary condition to avoid overflow. An exporting deme sends r solutions every M_{expt} iterations to a receiving deme. The receiving rate R_{impt} is the same as sending rate $\frac{d \times r}{M_{expt}}$, otherwise the sending buffer from at least one neighoring deme will overfollow eventually. PGAP algorithm (Fig. 4) injects one imported solution per iteration if the import pool is not empty, indicating $R_{inject} = 1$. A is then equivalent to:

$$\frac{d \times r}{M_{expt}} \leqslant 1 \tag{a}$$

Condition (a) indicates that, in each iteration, there is at most one incoming solution sent by neighboring demes. Note that condition (a) is implied in B because K_{impt} is a constant. For B, between any two consecutive import operations, neighboring demes constantly sends $d \times \frac{r}{M_{expt}} \times M_{impt}$ solutions to the receiving deme. Communication delays could hold

the receiving of these solutions to later import operations. However, since we must avoid the overflow of any sending buffers, the receive operation cannot be postponed infinitely. In fact, by applying condition (1) derived in Section 3.3.1, we get an upper bound for K_{impt}:

$$k_{impt} \leqslant K_{sendbuf} \times d \tag{b}$$

Furthermore, $K_{sendbuf} \times d$ is the largest possible number of solutions in the impool pool. This is simply because $R_{impt} \leqslant R_{inject} = 1$, meaning that the inject operator consumes solutions in the import pool no slower than the import operator producing solutions into the pool. Therefore, delayed receive operations in an import operation is the only possible scenario to make the number of solutions in the import pool following an import operation to be larger than that of the previous import. An import operation with delayed receive operations can receive up to $K_{sendbuf} \times d$ solutions. Meanwhile, since it takes at least $K_{sendbuf} \times d$ iterations to gather these many receiving requests (condition (a)), by the time the import operation eventually happens, there will be zero solutions in the pool. Thus, condition $K_{impt} \geqslant K_{sendbuf} \times d$ satisfies B.

Combing (a) and (b), we get:

$$\left(\frac{d \times r}{M_{expt}} \leqslant 1\right) \wedge (K_{impt} \geqslant K_{sendbuf} \times d) \longrightarrow \neg O \tag{2}$$

Condition (2) shows the correlations among PGAP parameters listed on Table 1 in order to avoid the overflow of the import pool. Based on conditions (1) and (2), buffer overflow issues caused by inappropriate configuration of PGAP parameters can be avoided.

3.4. Implementation

PGAP is implemented in C. The export and import operators are implemented using the MPI message-passing programming model. MPI's non-blocking point-to-point communication functions *MPI_Ibsend ()* and *MPI_Iprobe ()* are used for implementing non-blocking features of the export and import, respectively. To enable flexible control of the system outgoing message buffer, we use MPI's *MPI_Buffer_attach ()* to explicitly allocate memory for buffer-based message send operations. The communication topology is generated dynamically by making the sizes of the two dimensions of the 2-D grid as close as possible. The runtime configuration of PGAP parameters is based on the algorithmic anlaysis results in Section 3.3. Since severe communication delays can still cause the overflow of the sending buffer, PGAP implementation skips an export operation if the sending buffer is full.

A random number generator is used to provide stochastic choices on GA operators. Given the fact that GA often requires a large amount of iterations before converging to specified solution quality, the same random number sequence cannot be used on all demes even with different initial seeds. A parallel random point generator, SPRNG, is employed to generate a unique random number sequence on each deme [62].

4. Performance evaluation

Experiments were designed to evaluate the scalability of PGAP algorithm by using up to 16,384 cores. Both strong and weak scaling tests were performed, but they have to be tailored to PGA performance evaluation. Strong scaling tests were conducted to measure speedups in a set of large-scale PGAP runs, each using a different number of cores. However, two issues related to PGA performance evaluation have to be resolved: defining base case runs and comparing speedups in runs that achieve different solution quality. In weak scaling tests, population size, instead of problem size, is used as scaling factor in order to appropriately measure the computational effort required in relation to the increase of computing power. For both tests, results were compared with the corresponding synchronous implementation. PGA algorithmic capabilities such as numerical performance, solution quality improvement, and convergence were evaluated to understand the improved problem-solving capabilities by using the asynchronous migration strategy.

4.1. Experiment design

Two MIMD high-performance computing (HPC) systems on XSEDE (i.e., the Lonestar and Ranger clusters at the Texas Advanced Computing Center (TACC)) were used for the experiments. Lonestar has 22,656 cores with 0.302 petaflop peak performance. Each node on Lonestar has two 3.3 GHz Intel Xeon hexa-core 64-bit Westmere processors (12 cores) and 24 GB memory. Ranger has 62,976 cores with 0.579 petaflop peak performance. Each node on Ranger contains four 2.3 GHz AMD Opteron quad-core processors (16 cores) and 32 GB memory. Both systems use InfiniBand as interconnect. Up to 2048 cores on Lonestar were used to study the convergence of PGAP and track the number of unique solutions found in PGAP runs. Up to 16,384 cores on Ranger were used for scalability tests and numerical performance evaluation in solving large GAP instances.

Table 2
PGAP default configuration.

Parameters	Settings
Population size per deme	100
Initial population generation	Random with feasibility improvement or constraint-based improvement [24]
Selection	Binary tournament
Crossover	1-Point. Probability: 0.8
Mutation	1-Item mutation. Probability: 0.2
Replacement	Replacing the unfittest or worst
Elitism	Yes
Stopping rules	No solution improvement, bounded solution quality reached, or fixed number of iterations
Connectivity d	4
Migration rate r	2
Export interval M_{expt}	50
Import interval M_{impt}	25
Probability of holding	1/20 (the probability to export when no better solution found during a previous export interval)
Sending buffer size $K_{sendbuf}$	20 Solutions. Actual memory requirement is $(20 \times n \times 4 + buffer\_overhead)$ bytes
Import pool size K_{impt}	80 Solutions. Actual memory requirement is $(80 \times n \times 4)$

Five types of public GAP benchmark instances available from OR-LIB[1] and Yagiura's website[2] have been used in literature as benchmark datasets. Small-sized type D, E, and F instances from OR-LIB are used in small-scale experiments to verify the quality of solutions found by our baseline GA algorithm. Type D and E instances from Yagiura are large instances and considered more difficult because costs are inversely correlated with weights [63]. E801600, one of the largest instances of type E with 1600 items and 80 bins, was used for PGAP scaling tests and performance evaluation.

Table 2 shows the default configuration of PGAP parameters based on condition (1) and (2) derived in Section 3.3. The ratio M_{impt}/M_{expt} is configured to be much smaller than $K_{sendbuf}/r$ in order to reduce the impact of sporadic network congestions on the sending buffer. We also follow the guidelines by Hart et al. [10] and Alba and Troya [19] to make sure that results from different PGAP runs are comparable. For example, the stopping rule for speedup analysis is finding the best solution found by the base case runs, instead of setting a walltime or the number of iterations. A synchronous PGA is developed based on PGAP as the reference implementation for performance comparison purpose. This is done by adding global barriers for export and import operators to PGAP and using blocking communication functions and synchronous communication mode in MPI. Therefore, synchronization cost is the only source of performance difference.

4.2. Scalability analysis

It is not appropriate to directly use conventional weak and strong scaling testing methods to analyze the scalability of PGAP for the following reasons. First, the amount of computation work required to solve a combinatorial optimization problem depends as much on the problem difficulty as on the problem size. For example, a small-size (in terms of the number of items and/or bins) type E GAP instance may be 'harder' than a large-size type C instance because type E instances were generated by inversely correlate the cost and weight of each item [29]. Therefore, varying problem size only, as done in typical weak scaling experimentation, may not be sufficient to reveal how well PGA can solve large problems using more computational resources. Second, PGA is a type of randomized algorithm, multiple runs of the same PGA configuration may take different amount of execution time to achieve the same level of solution quality. Furthermore, while the overall problem size is fixed in strong scaling, changing the number of cores also changes deme size and the number of demes handled by each core. Such changes have profound impact on local evolution and global migration effect, together complicating the overall evaluation of computational performance.

4.2.1. Strong scaling test

Our strong scaling experiment measures the speedup of PGAP by increasing the number of cores, but keeping the size of global population constant. Each core runs one deme and thus deme size, which is equal to the global population size divided by the number of cores, varies accordingly. Speedup is calculated as the ratio of the execution time of two PGAP runs with the denominator being a reference (referred to as base case) using a small number of cores. The reason to use PGAP itself as the reference algorithm is that PGA and GA are not suitable for direct comparison as they are considered to have different algorithmic behaviors. Such definition of speedup is referred to as *relative speedup* by Sun and Ni [64] or *type I speedup* by Alba and Troya [19]. Since the global population size is kept constant, the increase of the number of cores leads to the decrease of deme size and the increase of the number of demes. It is known that PGA performance is highly influenced by deme size and migration. Specifically, large deme size makes GA search on a core toward a random search and injects solutions from fewer demes, while small deme size makes local GA search "premature" but injects solutions from more demes. Our strong scaling test is designed to compare the speedup between PGAP and its synchronous implementation.

[1] http://people.brunel.ac.uk/~mastjjb/jeb/info.html
[2] http://www-or.amp.i.kyoto-u.ac.jp/~yagiura/gap/

In the experiment, global population size was kept at 1,638,400. As the number of cores used in the experiment increased from 512 to 16,384, population size on each deme decreased from 3200 to 100. Since comparing with sequential algorithm performance would produce misleading results, we used 512-core run of PGAP as the base case for PGAP. 512-core run of its synchronous version is then used as the base case for speedup calculation for the synchronous version. Therefore, the linear speedup in our experiment would be 32 when 16,384 cores are used. When a specified solution quality threshold could not be reached within the maximum walltime allowed using 512 cores, the execution time of the base case run is set to be the maximum walltime (i.e., 16 h (57,600 s)). In such scenario, using the maximum walltime in base case is a conservative estimation of the actual speedup in speedup comparison because the resulted speedup is the lower bound of the actuall speedup. Fig. 5(a) and (b) illustrate the speedup measurements against different solution quality thresholds, quantified as the upper bound of fitness value. Results show that, for both synchronous and asynchronous runs, using more cores resulted in better speedup, even superlinear speedups. By looking at speedups achieved at multiple solution quality thresholds, more comprehensive view of PGAP numerical performance was obtained by examining the relationship between the levels of difficulty to reach a specified solution quality and the number of cores used. For example, for the type E instance, both synchronous and asynchronous runs experienced difficulty to achieve solution quality threshold 188,000, indicated by sublinear speedup achieved. Asynchronous PGAP runs (Fig. 5(a)) exhibited superlinear speedup at 8 out of 11 solution quality thresholds when using 16,384 cores, while 3 out of 8 were observed in synchronous runs (Fig. 5(b)). Synchronous runs could not reach the three tightest solution quality thresholds reached by PGAP, while asynchronous runs were also able to find solutions better than the tightest threshold 183,500 when using 8192 and 16,384 cores. Beyond 8192 cores, speedup increase in the synchronous runs became insignificant, while PGAP scales well to 16,384 cores.

Note that Fig. 5(a) and (b) cannot be directly used to compare speedups between PGAP and its synchronous version. This is because they used different base cases for speedup calculation (512-core runs of PGAP and its synchronous version, respectively). Instead, we define *ratio of speedup* as below for comparing speedup differences between asynchronous and synchronous migration strategies. It is measured as the ratio of execution time of the synchronous version and that of PGAP:

$$ratio\_of\_speedup = \frac{speedup_{async}}{speedup_{sync}} = \frac{T_{base}/T_{async}}{T_{base}/T_{sync}} = \frac{T_{sync}}{T_{async}} \tag{3}$$

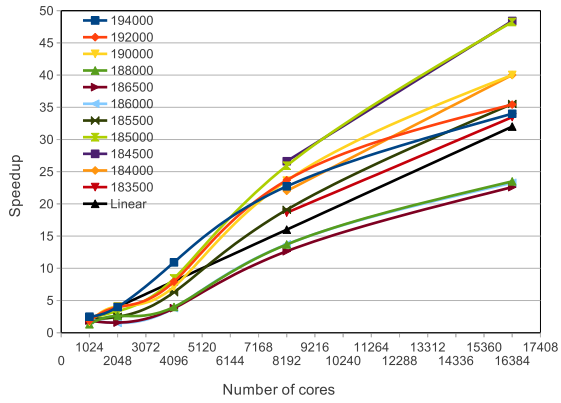
where T_{base} is the execution time of a common base case. Fig. 5(c) shows the ratio of speedup, calculated based on Eq. (3) for all of the solution quality thresholds that the synchronous version achieved in at least one setting of the given numbers of cores. For all measurable cases, the values of the ratio are larger than 1, meaning that PGAP achieved better speedup than its synchronous version. Particularly, when using 16,384 cores, the ratio ranges from 1.90 to 2.47.

4.2.2. Weak scaling test

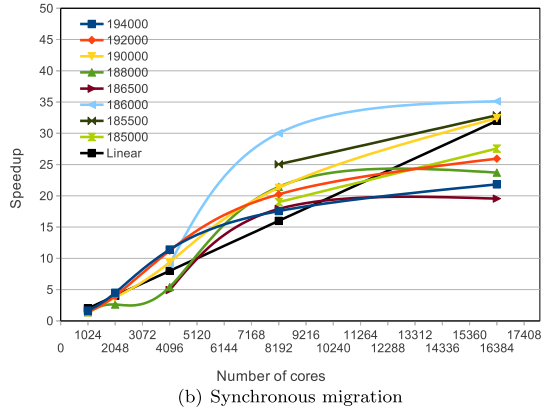
Because increasing problem size may alter problem difficulty and the amount of computation needed to solve the problem, problem size, as used in typical weak scaling test, is not appropriate to use in weak scaling test of PGAs. Instead, our weak scaling test varies the size of global population in order to measure how PGAP leverages larger global population numerically, enabled by the use of more cores, to diversify the search in solution space and achieve a designated solution quality more effectively. We refer to this weak scaling test method as *population scaling*. The benefit of using a large population is straightforward because it provides a much larger sampling solution space for GA. But using a large population can easily turn a GA evolutionary process into a random search that can hardly converge on a single core. With PGA, however, the evolutionary process can be kept effective at deme level, but much more solution space can be searched by running a large number of demes simultaneously.

It is worth noting that population scaling adds more cores to do additional work on the same problem, similar to the "new-era" weak scaling proposed by Sarkar et al. [65]. Leveraging larger global population, population scaling is designed to evaluate the capability of PGAP to obtain better solutions in a shorter amount of time. The test was done on the Ranger supercomputer. We use the time taken to achieve a certain solution quality as the measure of population scaling. Deme size was set to 200. The number of cores used ranges from 1024 to 16,384. As the number of cores doubles, global population size also doubles from 204,800 to 3,287,400. We ran this test on both PGAP and its synchronous version. Each run was given one hour to finish.

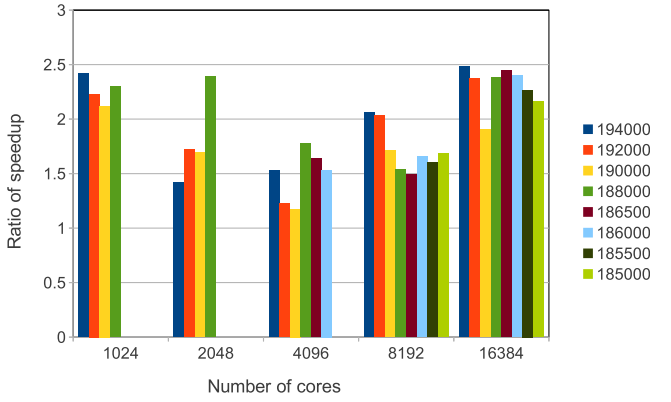
The benefit of using large population PGAP can be shown in Fig. 6(a). The time taken to achieve certain solution quality threshold, quantified again as the upper bound of fitness value, was measured in accordance to the increase of the number of cores. Overall, as the bound became tighter, it took more time to find solutions whose fitness values are equal to or better than the bound. For a particular bound, it is obvious that using more cores reduces the time taken to find solutions with equal or better quality. This trend became more significant for tighter bounds. In fact, for those bounds tighter than 184,000, runs using smaller number of cores started running out of time to find solutions of specified quality. For example, solutions with bound 182,500 or better were only found by using 8192 and 16,384 cores. For all of the runs, using 16,384 cores significantly reduced the time taken to achieve the specified solution quality. This can be explained as follows: (1) running massive demes simultaneously can greatly increase the overall probability of finding new elite solutions; and (2) migration operators are able to propagate good solutions to all demes if they are globally better, stimulating local evolutionary processes at deme level. There were some variations observed. For the test problem instance, improving solution quality from threshold 188,000 to 186,500 was difficult with the given PGA configuration in both asynchronous and synchronous



(a) Asynchronous migration



(b) Synchronous migration

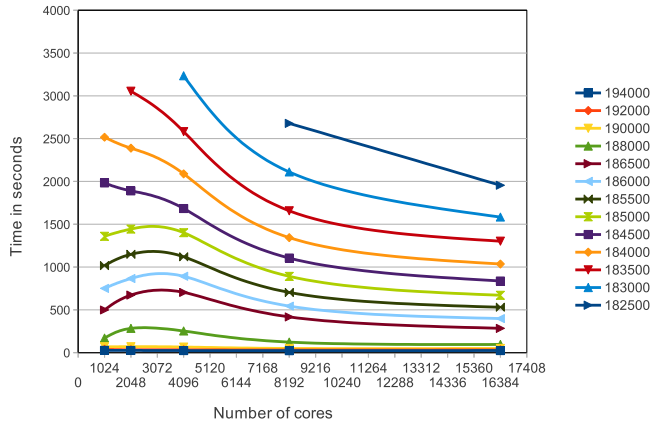


(c) Ratio of speedup comparison

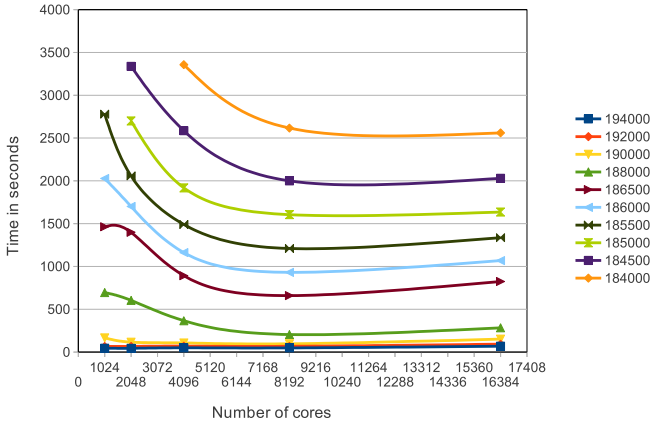
Fig. 5. Strong scaling results. Speedups were measured against different solution quality thresholds. Missing points/columns were the runs exceeding maximum walltime.

versions. In the asynchronous version, higher fluctuations were observed in one of the runs of using 2048 and 4096 cores, respectively, and had subsequent effect on reaching tighter bounds. Such variation was due to the dynamics in stochastic computation specific to each run.

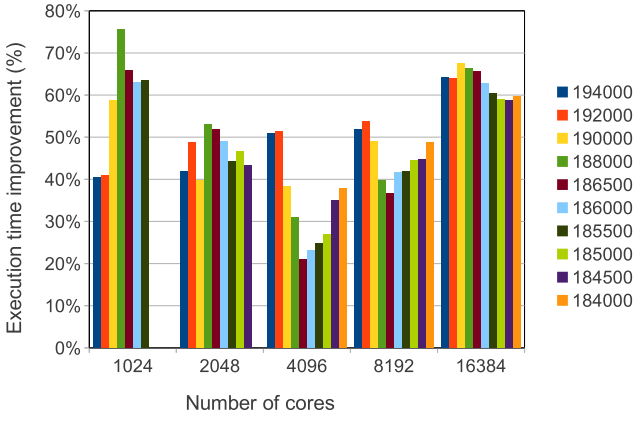
In contrast, Fig. 6(b) shows the weak scaling performance of the synchronous version. Consistently, not only the time taken to reach a bound was longer than PGAP, the synchronous runs could not find solutions better than 184,000 given the same amount of execution time (1 h). Fig. 6(c) shows this trend more clearly by comparing the execution time improvement, measured as the percentage of $\frac{execution\_time_{sync} - execution\_time_{async}}{execution\_time_{sync}}$. When using 16,384 cores, the execution time improvement was consistently around 60%. In summary, the weak scaling experiment shows that asynchronous migration was able to exploit large population size more effectively and showed significant numerical performance advantages over the corresponding synchronous version as more cores were used.



(a) Aynchronous migration



(b) Synchronous migration



(c) Execution time improvement

Fig. 6. Weak scaling results. Execution times were measured against different solution quality thresholds. Missing points were the runs exceeding maximum walltime (one hour).

4.3. Communication to computation ratio

One advantage of using asynchronous migration in PGA is improved communication to computation ratio. Such advantage becomes more obvious when using a large number of cores. In any synchronous PGA implementation, a delay at one core could affect all, while such delays in asynchronous migration only affect direct neighbors (in receiving solutions) and can even be offset by the overlapping of computation and communication.

In this experiment, the E801600 problem instance was solved by both asynchronous and synchronous versions of PGAP with the same settings as specified in the weak scaling experiment. Fig. 7 shows the communication cost as percentage of total execution time. PGAP's communication cost (on average 15.5%) was significantly lower than the synchronous version (on average 54%). For the synchronous version, the communication cost increased steadily as the number of cores doubled, mainly due to the increase cost of MPI_Barrier () calls. While for PGAP, the communication cost decreased as more cores were used. The decrease can be explained as follows. In a GA execution, as the evolutionary process continues, it becomes harder to find better solutions. Therefore, random solutions are more likely to be selected and migrated. A holding strategy is applied in both PGAP and its corresponding synchronous version to delay the export operation in this situation in order to avoid excessive injection of randomness. As more cores are used, PGAP achieves a solution bound earlier, beyond which getting better solutions takes longer time and the holding strategy is applied more frequently. This scenario applies to the synchronous version, too. But the expensive MPI_Barrier () cost at large scale is more significant than the reduction of communication caused by the holding operations. This observation indicates that optimal search strategies should be employed to keep searching efficiency at pace with the increase of computing power.

To better understand the communication variation in asynchronous migration, a snapshot of communication cost on each core is plotted for a run using 16,384 cores, shown in Fig. 8. Most of cores' communication cost was around 13%, with a few cores ranging between 8.5% and 18%. This means that communication cost in this large-scale PGAP run was consistent across cores despite that migration operations were not synchronized. Also, the asynchronous migration strategy was able to manage the 9.5% communication cost variation. This experiment further illustrated that asynchronous PGA, if designed properly, can also be reliable while introducing significantly lower communication cost than synchronous PGAs.

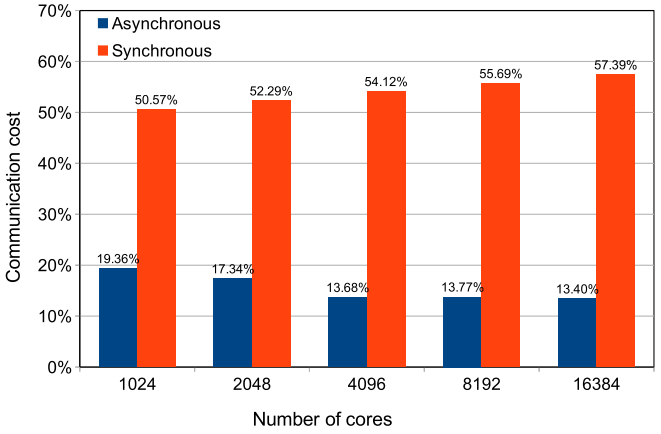


Fig. 7. Distribution of communication cost in weak scaling test.

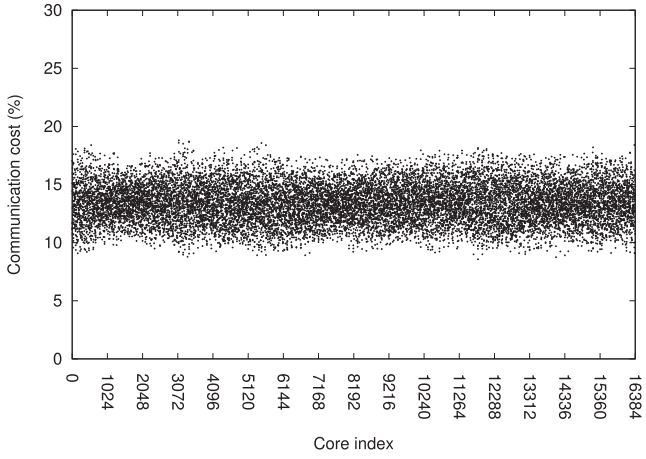


Fig. 8. Snapshot of communication cost at each core in a PGAP run using 16,384 cores.

4.4. Solution quality and numerical performance

4.4.1. Solution quality of baseline GA

This experiment was designed to evaluate solution quality of the baseline sequential GA of PGAP, which enhanced the feasibility and quality improvement operators used in Chu and Beasley [22]. As mentioned in Section 3, randomness was introduced in the feasibility and solution quality improvement operators as a way to diversify search patterns for item reassignment as PGAP scales to use large amount of demes. This algorithmic change improves PGA performance in both spatial and temporal contexts. For all of the demes spatially distributed in PGA computation, the improved operators enable more flexible search patterns on all demes with additional help from using a unique random number sequence per deme. Temporally, they allow each iteration to use a different search order from previous iterations for finding alternative item reassignments. Therefore, the change we made to introduce more randomness also improves the baseline sequential GA. In this experiment, GA parameters were set as the same as in [22], i.e., population size was 100 and the stopping rule was without improvement in consecutive 500,000 iterations. We ran the baseline algorithm and the two-deme PGAP cases only once. Results were compared with the best-so-far GA results [24] to the authors' knowledge. Table 3 lists the best solutions found and compares the gap to the maximum lower bounds (found by the linear programming package (IP/LP Copt)) found among CPLEX commercial software [66], CRH-GA algorithm [24], and our baseline GA. Experiment results showed that our baseline GA outperforms Feltl's GA in 31 out of 39 small-scale type D, E, and F instances. The solution quality improvement is significant because results reported in [24] show the best among multiple runs.

Table 3 also illustrates the immediate benefit of using two demes with migration on a single core (column 'Two-deme PGAP'). In the two-deme case, the single core take turns to run two demes with migration. Each deme is assigned with a different random number sequence that generates different solution search paths. The benefit of using multiple demes with migration is obvious for the GAP problem. Even though the two demes ran on a single core, Table 3 shows that it effectively improved solution quality and outperformed Feltl's GA in 37 instances, and our sequential GA in 31 instances.

4.4.2. Numerical performance

Using asynchronous migration has two major impacts on the numerical performance of PGA: (1) reduced communication cost allows PGA to run more iterations within the same amount of time; and (2) migration intervals become more dynamic among spatially distributed demes. In the case of synchronous migration, a migration operation is followed by a 'silent' time among all demes, the length of which is determined by the parameters of migration interval. But in the case of asynchronous migration, as computation goes on, the timing for migration shifts gradually among demes because of runtime dynamics such as system clock and communication delays. Hart et al. [10] demonstrated the benefits of such dynamics on improving PGA performance in terms of the total number of function evaluations required to achieve a designated solution quality. Alba et al. [67] demonstrated the effect of network latency of LAN (local area network) and WAN (wide area network) on PGA efficiency when they found that WAN executions could be more efficient due to longer isolation times. We used the result of the weak scaling experiment to evaluate numerical performance of PGAP.

To understand the benefit of reduced communication cost, we use the number of iterations performed per second as a measure (referred to as *iteration rate*) to evaluate PGAP's numerical efficiency. In a multi-deme case, iteration rate is measured as the average value across all demes. Fig. 9 shows the result of using 1024–16,384 cores. The result on the synchronous version is easy to understand. The decrease of iteration rate is due to the increasing cost of MPI_Barrier () as more cores are involved. Additionally, because of the use of global barrier, the standard deviation is marginal. Compared to the synchronous case, the benefit of using asynchronous migration is significant, indicated by much higher iteration rate. We also observed that, as more cores were used, the iteration rate did not decrease. Instead, the rate increased from 441.84 to 471.84. Such increase was due to more frequent invocations of the holding strategy. The standard deviation for the asynchronous case decreased from 25.39 to 18.14 as the number of cores increases from 1024 to 16,384. This may not suggest to use more cores in order to get more stable iteration rate. Instead, the decreasing standard deviation was, again, due to more frequent holding operations caused by earlier PGA convergence in asynchronous runs. In summary, the iteration rate analysis clearly shows that the asynchronous migration strategy in PGAP is highly scalable to the number of cores and the consistent iteration rate allows PGAP to finish more evolution iterations collectively by using more computing power.

By performing more iterations on a problem instance, the improvement of solution quality is expected for PGAP. Fig. 10 shows how much PGAP can outperform the synchronous version in obtaining better solutions within one hour of computation. We measured solution quality gain over the synchronous version as the percentage of $\frac{fitness\_value_{sync} - fitness\_value_{async}}{fitness\_value_{sync}}$. For all of the cases we evaluated, solution quality of the asynchronous version is better than the synchronous one. The largest improvement of 1.08% occurred when using 16,384 cores, which improved fitness value by 1999. The percentage of improvement increases as the number of cores doubled.

While the improvement of numerical performance from reduced communication cost is apparent, the influence of the migration strategy itself on PGAP's problem-solving capability can be intricate due to the high variation on the timing of migration operations among a large number of demes. A preliminary study was conducted to measure such runtime influence in PGAP by calculating the number of iterations needed to achieve a set of solution quality thresholds on Ranger, instead of measuring the execution time which is highly correlated with communication cost. Fig. 11 shows the result obtained from the weak scaling experiment. The number of iterations needed in each run is calculated as the minimum number of iterations taken among all of the demes that reached the solution bound. For the cases of 1024, 2048, and 16,384 cores, PGAP took

Table 3
Solution quality comparison [1,2,3,4] among CPLEX software, CRH-GA, the baseline sequential GA, and two-deme PGAP.

Prob. type	Size		IP/LP Copt	CPLEX	CRH-GA	Baseline GA		Two-deme PGAP	
	m	*n*		Gap (%)	Gap (%)	Fitness value	Gap (%)	Fitness value	Gap (%)
D	5	400	25670	opt	**0.44**	25790	**0.47**	25737	**0.26**
D	10	400	25274.8	0.18	**1.31**	25509	**0.93**	25435	**0.63**
D	20	400	24546.8	0.51	**1.97**	24903	**1.45**	24855	**1.26**
D	40	100	6092	3.96	**3.72**	6327	**3.86**	6265	**2.84**
D	40	200	12244.9	2.22	**3.06**	12561	**2.58**	12520	**2.25**
D	40	400	24371.8	1.1	**2.81**	24851	**1.97**	24801	**1.76**
D	80	100	6110.5	6.6	**7.01**	6526	**6.80**	6501	**6.39**
D	80	200	12132.3	2.87	**3.76**	12490	**2.95**	12454	**2.65**
D	80	400	24177	2	**3.02**	24748	**2.36**	24587	**1.70**
E	5	100	7757	opt	**0.24**	7774	**0.22**	7760	**0.04**
E	5	200	15611	opt	**0.23**	15632	**0.13**	15626	**0.10**
E	5	400	30794	opt	**0.28**	30872	**0.25**	30826	**0.10**
E	10	100	7387.8	0.61	**0.91**	7454	**0.90**	7436	**0.65**
E	10	200	15039.8	0.25	**0.96**	15135	**0.63**	15126	**0.57**
E	10	400	29977.9	0.09	**0.94**	30135	**0.52**	30142	**0.55**
E	20	100	7348.2	1.32	**1.74**	7463	**1.56**	7448	**1.36**
E	20	200	14765.2	0.89	**1.7**	14923	**1.07**	14918	**1.03**
E	20	400	29500.3	0.34	**1.6**	29845	**1.17**	29810	**1.05**
E	40	100	7316.1	3.32	**3.11**	7497	**2.47**	7522	**2.81**
E	40	200	14630.4	1.85	**2.2**	14835	**1.40**	14858	**1.56**
E	40	400	29186.6	0.69	**2.14**	29586	**1.37**	29593	**1.39**
E	80	100	7650	opt	**0.78**	7670	**0.26**	7668	**0.24**
E	80	200	14566.7	2.17	**2.93**	14833	**1.83**	14846	**1.92**
E	80	400	29161.3	1.57	**2.49**	29631	**1.61**	29567	**1.39**
F	5	100	2755	opt	**0.41**	2761	**0.22**	2761	**0.22**
F	5	200	5294	opt	**0.35**	5304	**0.19**	5315	**0.40**
F	5	400	10745	opt	**0.25**	10776	**0.29**	10765	**0.19**
F	10	100	2276.8	1.99	**3.95**	2364	**3.83**	2348	**3.13**
F	10	200	4644.6	1.13	**3.12**	4794	**3.22**	4759	**2.46**
F	10	400	9372.7	0.46	**2.78**	9631	**2.76**	9604	**2.47**
F	20	100	2145.1	8.15	**8.38**	2339	**9.04**	2301	**7.27**
F	20	200	4310.1	4.73	**6.55**	4585	**6.38**	4552	**5.61**
F	20	400	8479.4	2.38	**7.15**	9050	**6.73**	8950	**5.55**
F	40	100	2110.1	21.28	**18.27**	2476	**17.34**	2501	**18.53**
F	40	200	4086.5	10.14	**12.86**	4522	**10.66**	4578	**12.03**
F	40	400	8274.3	4.05	**10.36**	9105	**10.04**	8907	**7.65**
F	80	100	2064.4	31.37	**26.77**	2655	**28.61**	2583	**25.12**
F	80	200	4123.4	19.05	**17.33**	4869	**18.08**	4837	**17.31**
F	80	400	8167.1	9.18	**12.55**	9248	**13.23**	9127	**11.75**

Columns in bold font indicate the gap between the fitness value obtained from the method of the column and the IP/LP Copt bound.
[1] Column IP/LP: lower bound found by linear programming.
[2] Column CPLEX: best solutions found by CPLEX. *opt*: optimal solution found.
[3] Gap (%): the percentage of $\frac{fitness\_value_{best\_solution} - lower\_bound}{lower\_bound}$.
[4] IP/LP, CPLEX, and CRH-GA results are from Feltl et al. [24]. They are included for comparison purpose.

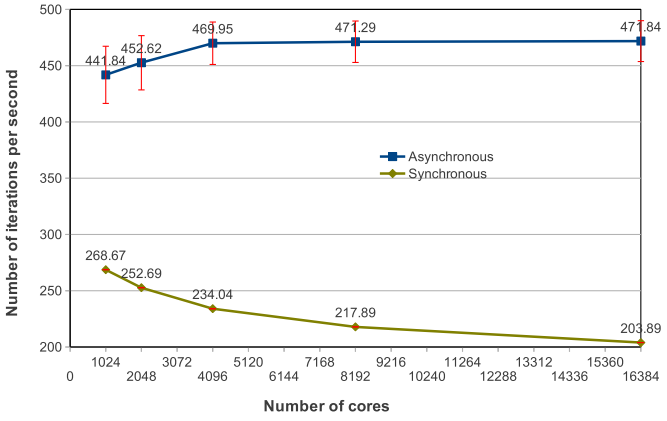


Fig. 9. Iteration rate comparison. The standard deviation is also plotted.

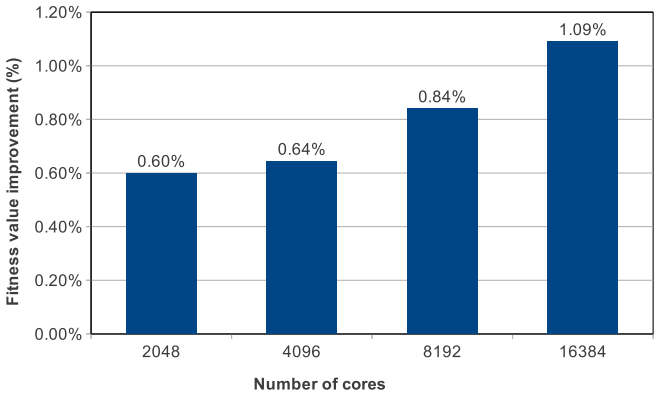
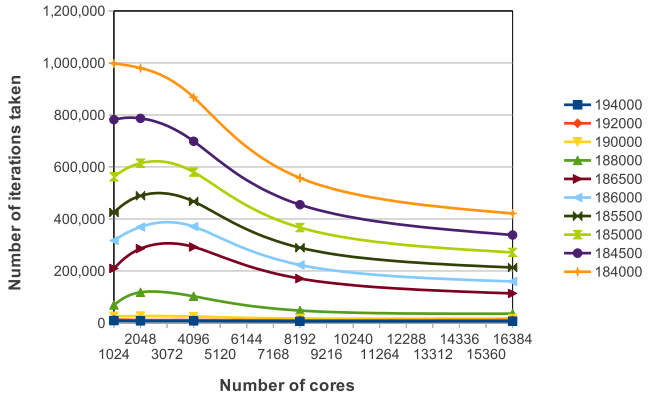
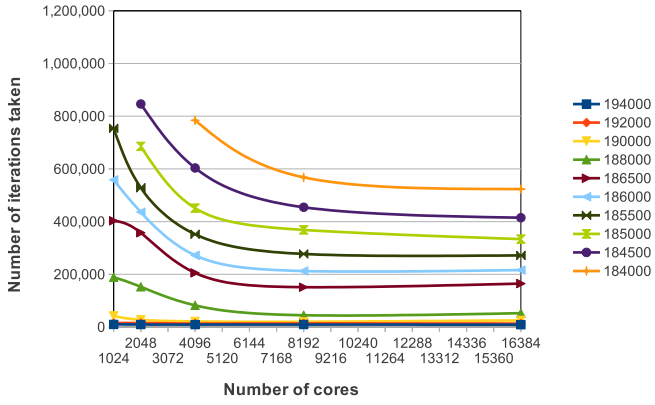


Fig. 10. Solution quality comparison, measured as the percentage of fitness value improvement in PGAP over the synchronous version. Each run was given one hour to finish.



(a) Aynchronous migration



(b) Synchronous migration

Fig. 11. Number of iterations required to achieve specified solution quality. Missing points were the runs exceeding maximum walltime (one hour).

less number of iterations to reach any of the ten solution quality thresholds. But for the cases of 4096 and 8192 processors, more iterations were needed than the synchronous version to improve from threshold 188,000 to 186,500. Combined with Fig. 6, the numerical performance gain from PGAP seems to mainly attribute to the significant reduction of communication cost. But for the case of 16,384, we consistently observed better performance of PGAP in both execution time and the number of iterations in achieving a specified solution quality.

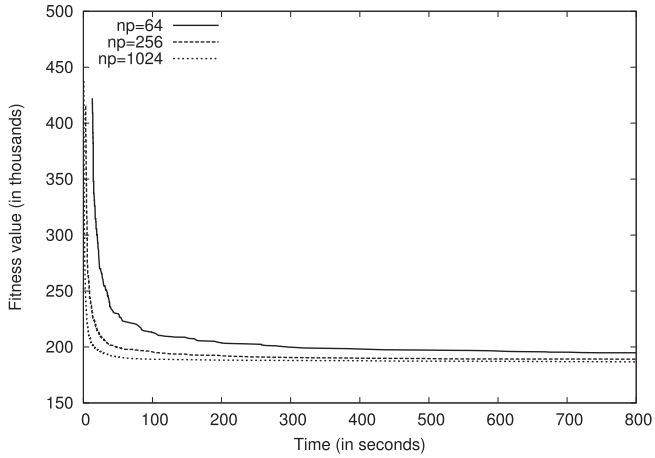
The advantage of using asynchronous migration is also illustrated by the convergence experiment. Fig. 12 depicts snapshots of the fitness value of the best solutions found at each timestamp in a set of runs of PGAP (Fig. 12)) and its synchronous version (Fig. 12(b)) using 64, 256, and 1024 cores on the Lonestar supercomputer. PGAP converged much faster in all cases.

4.4.3. Finding feasible solutions

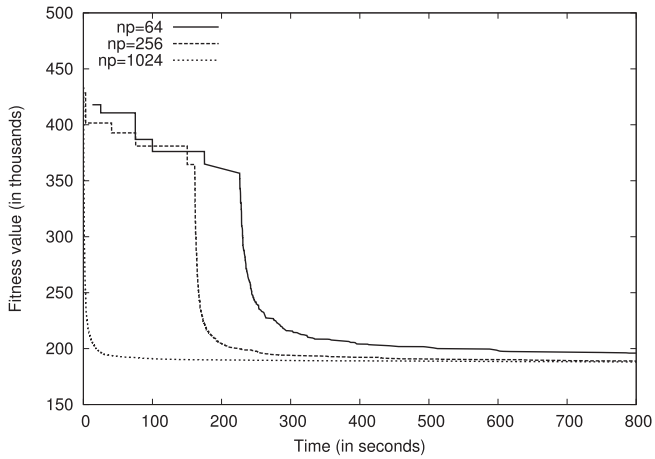
The capability to find more feasible solutions by leveraging massive computing resources is an important factor to illustrate how large-scale PGAP runs explore solution space more efficiently than its synchronous version. Obtaining more feasible solutions of specified solution quality is often one of the problem-solving goals for a lot of GAP applications. By investigating the landscape of found feasible solutions, better problem-specific heuristics can be developed to guide solution space search toward promising directions while avoid being trapped in local optima. The capability of PGAP for finding feasible solutions is measured by counting the number of unique solutions found with equal or better solution quality than the specified quality thresholds within one hour of weak-scaling test runs. PGAP runs using 16,384 cores were compared with the runs of its synchronous version. Results showed that, on average, PGAP found 11,093 unique solutions with fitness value better than 194,000, 19.98% more than the synchronous version. This trend became more obvious as solution quality thresholds became tighter. At thresholds of 188,000 and 186,000, PGAP found 40.62% and 75.90% more unique solutions than the synchronous version, respectively. Such improvement on problem-solving capability mainly attributes to the fact that, given the same amount of execution time, the asynchronous migration strategy allows PGAP to perform significantly more iterations.

5. Concluding discussions

This paper described a scalable parallel genetic algorithm for solving large GAP instances by leveraging massively parallel computing. Realizing that synchronizing massive cores imposes significant performance penalty, an asynchronous migration strategy is developed to improve the numerical performance of PGAP. This strategy has three migration operators: export,



(a) Asynchronous migration



(b) Synchronous migration

Fig. 12. PGAP convergence. np: number of cores.

import, and inject. By using buffer-based communication and non-blocking message passing between sending and receiving demes, migration communication can be overlapped with GA computation without adding any global barrier to synchronize demes. Compared to corresponding synchronous implementation, experiments showed significant improvement of PGAP on communication and computation ratio, speedup, and the capability to explore solution space efficiently. Superlinear speedups were observed in strong-scaling tests against large problem instances. Parallel efficiency study of PGA is also important to explore how we can efficiently use computing resources to achieve a specified solution threshold given a particular problem. Cantu-Paz and Goldberg [68] studied the speedup and efficiency of a simplified global parallelization-based PGA from theoretical perspective and found interesting results on how to determine the optimal number of cores for solving a given problem instance. However, the study of parallel efficiency in coarse-grained PGA is more complicated and remains to be an open research problem because an optimal configuration of PGA may depend on the problem, the difficulty of reaching the specified solution quality, and runtime dynamics since PGA is a type of stochastic algorithm. By using the asynchronous migration strategy, the communication cost of PGAP is greatly reduced. As a result, each core is efficiently kept busy for local evolutionary computation. But computing resources can still be wasted if one deme largely repeats the work done by another deme. Similarity analysis on inter-deme populations, as part of our future work, may help reveal such phenomenon and develop runtime strategies to avoid it.

Experiment results in the weak scaling experiment showed that PGAP exhibited desirable scalability for leveraging large populationsize enabled by using massive cores in solving large problem instances. To avoid runtime failure and the loss of good solutions observed in asynchronous PGAs, two buffer overflow problems are identified and addressed in PGAP by establishing two sufficient conditions for appropriate PGAP parameter configuration. The design of the asynchronous migration strategy is generic and independent of the targeted GAP problem. Therefore, the two sufficient conditions can be further applied as general guidelines to the development of asynchronous coarse-grained PGAs.

We will further study PGA algorithmic behaviors using larger scale parallel computing environments, developing GAP-specific solution space search strategies, and extending the asynchronous PGA design on heterogeneous computing platforms. We will further analyze migration strategies of PGAP by looking at alternatives on topology and migration parameters for more efficient GA search process control. We will continue to study the scalability of PGAP at larger scale (e.g., exascale [57]). The findings presented in this paper have been incorporated in PGAP software library. This library has been deployed on and tuned for the petascale Blue Waters supercomputer at the National Center for Supercomputing Applications (NCSA). Desirable scalability up to 131,072 cores, similar to the results presented in this paper, was obtained in solving very large problem instances. To further improve PGAP, the feasibility and quality improvement operators will be enhanced by exploring neighboring functions such as described in [69]. Problem characteristics illustrated in [29] will be leveraged to design GAP instance-specific migration strategies by changing migration intervals, migration rate, and strategies for selection and injection dynamically at runtime. Hybrid computing resources become increasingly common on supercomputers, such as Blue Waters (CPU + GPU) and Stampede at TACC (CPU + MIC + GPU). On these resources, the cost of synchronization may be more significant due to variations of individual computer nodes and underlying network. Asynchronous migration is a suitable solution for such resources. However, the migration strategy developed in this paper needs to be extended to take advantage of hybrid computing resources. The associated algorithmic analysis needs to consider high variations of computing characteristics on hybrid computing platforms.

Acknowledgments

This work is supported in part by the National Science Foundation (NSF) under Grant No. OCI-1047916. Computational experiments used the Extreme Science and Engineering Discovery Environment (XSEDE) (resource allocation Award Number SES090019), which is supported by the National Science Foundation Grant No. OCI-1053575. This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (award number OCI 07-25070) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications. The authors thank Dr. Alberto Maria Segre at the University of Iowa for insightful discussions that led to this research topic and are grateful for the insightful comments received from CIGI members: Yizhao Gao, Anand Padmanabhan, and Mengyu Guo.

References

[1] J.H. Holland, Adaptation in Natural and Artificial Systems, MIT Press, Cambridge, MA, USA, 1992.
[2] S. Wright, The roles of mutation, inbreeding, crossbreeding and selection in evolution, in: Proc. Sixth Int. Cong. Genet., vol. 1, 1932, pp. 356–366.
[3] D.E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.
[4] P.S. Oliveto, J. He, X. Yao, Time complexity of evolutionary algorithms for combinatorial optimization: a decade of result, Int. J. Automat. Comput. 4 (3) (2007) 281–293.
[5] E. Alba, M. Tomassini, Parallelism and evolutionary algorithms, IEEE Trans. Evol. Comput. 6 (5) (2002) 443–462, http://dx.doi.org/10.1109/TEVC.2002.800880.
[6] Z. Konfrst, Parallel genetic algorithms: advances, computing trends, applications and perspectives, Parallel Distrib. Process. Symp. Int. 7 (2004) 162b. http://doi.ieeecomputersociety.org/10.1109/IPDPS.2004.1303155.
[7] C.-H. Huang, S. Rajasekaran, High-performance parallel bio-computing, Parallel Comput. 30 (910) (2004) 999–1000, http://dx.doi.org/10.1016/j.parco.2004.01.003.

[8] J.I. Hidalgo, F. Fernandez, J. Lanchares, E. Cant-Paz, A. Zomaya, Parallel architectures and bioinspired algorithms, Parallel Comput. 36 (1011) (2010) 553–554, http://dx.doi.org/10.1016/j.parco.2010.09.001.
[9] E. Alba, J.M. Troya, A survey of parallel distributed genetic algorithms, Complexity 4 (4) (1999) 31–52.
[10] W.E. Hart, S.B. Baden, R.K. Belew, S.R. Kohn, Analysis of the numerical effects of parallelism on a parallel genetic algorithm, in: IPPS '96: Proceedings of the 10th International Parallel Processing Symposium, IEEE Computer Society, Washington, DC, USA, 1996, pp. 606–612.
[11] M.L. Fisher, R. Jaikumar, L.N.V. Wassenhove, A multiplier adjustment method for the generalized assignment problem, Manage. Sci. 32 (9) (1986) 1095–1103.
[12] C. Chekuri, S. Khanna, A PTAS for the multiple knapsack problem, in: SODA '00: Proceedings of the 11th Annual Symposium on Discrete Algorithms, 2000, pp. 213–222.
[13] R. Borndorfer, R. Weismantel, Relations among some combinatorial programs, Tech. rep., Konrad-Zuse-Zentrum fr Informationstechnik Berlin, 1997.
[14] D.G. Cattrysse, L.N.V. Wassenhove, A survey of algorithms for the generalized assignment problem, Eur. J. Oper. Res. 60 (3) (1992) 260–272, http://dx.doi.org/10.1016/0377-2217(92)90077-M.
[15] V. Balachandran, An integer generalized transportation model for optimal job assignment in computer networks, Oper. Res. 24 (4) (1976) 742–759.
[16] R.G. Cromley, D.M. Hanink, Coupling land use allocation models with raster GIS, J. Geograph. Syst. 1 (2) (1999) 137–153, http://dx.doi.org/10.1007/s101090050009.
[17] A. Freville, The multidimensional 0–1 knapsack problem: an overview, Eur. J. Oper. Res. 155 (1) (2004) 1–21.
[18] Xsede, <http://xsede.org>, 2013.
[19] E. Alba, J.M. Troya, Improving flexibility and efficiency by adding parallelism to genetic algorithms, Stat. Comput. 12 (2) (2002) 91–114.
[20] B.A. Shapiro, J.C. Wu, D. Bengali, The massively parallel genetic algorithm for RNA folding: MIMD implementation and population variation, Bioinformatics 17 (2) (February 2001) 137–148.
[21] M. McCool, Scalable programming models for massively multicore processors, Proc. IEEE 96 (5) (2008) 816–831, http://dx.doi.org/10.1109/JPROC.2008.917731.
[22] P.C. Chu, J.E. Beasley, A genetic algorithm for the generalised assignment problem, Comput. Oper. Res. 24 (1) (1997) 17–23, http://dx.doi.org/10.1016/S0305-0548(96)00032-9.
[23] J.M. Wilson, A genetic algorithm for the generalised assignment problem, J. Oper. Res. Soc. 48 (8) (1997) 804–809.
[24] H. Feltl, G.R. Raidl, An improved hybrid genetic algorithm for the generalized assignment problem, in: SAC '04: Proceedings of the 2004 ACM Symposium on Applied Computing, ACM, New York, NY, USA, 2004, pp. 990–995, http://dx.doi.org/10.1145/967900.968102.
[25] T. Lau, E. Tsang, The guided genetic algorithm and its application to the generalized assignment problem, in: Proceedings Tenth IEEE International Conference on Tools with Artificial Intelligence (10–12 Nov 1998), 1998, pp. 336–343. http://dx.doi.org/10.1109/TAI.1998.744862.
[26] L. Lorena, M. Narciso, J. Beasley, A constructive genetic algorithm for the generalized assignment problem, Evol. Optim. 5 (2002) 1–19.
[27] F. Qian, R. Ding, Simulated annealing for the 01 multidimensional knapsack problem, J. Chin. Univ. Numer. Math. 16 (4) (2007) 320–327.
[28] J.A. Diaz, E. Fernandez, A tabu search heuristic for the generalized assignment problem, Eur. J. Oper. Res. 132 (1) (2001) 22–38.
[29] M. Yagiura, T. Ibaraki, F. Glover, A path relinking approach with ejection chains for the generalized assignment problem, Eur. J. Oper. Res. 127 (2) (2006) 548–569.
[30] V. Jeet, E. Kutanoglu, Lagrangian relaxation guided problem space search heuristics for generalized assignment problems, Eur. J. Oper. Res. 127 (3) (2007) 1039–1056.
[31] A.P. French, J.M. Wilson, An lp-based heuristic procedure for the generalized assignment problem with special ordered sets, Comput. Oper. Res. 34 (8) (2007) 2359–2369, http://dx.doi.org/10.1016/j.cor.2005.09.008.
[32] L.D. Whitley, Cellular genetic algorithms, in: Proceedings of the Fifth International Conference on Genetic Algorithms, 1993, pp. 658.
[33] S.E. Eklund, A massively parallel architecture for distributed genetic algorithms, Parallel Comput. 30 (56) (2004) 647–676, http://dx.doi.org/10.1016/j.parco.2003.12.009.
[34] M. Ruciski, D. Izzo, F. Biscani, On the impact of the migration topology on the island model, Parallel Comput. 36 (1011) (2010) 555–571, http://dx.doi.org/10.1016/j.parco.2010.04.002.
[35] D. DAmbrosio, W. Spataro, Parallel evolutionary modelling of geological processes, Parallel Comput. 33 (3) (2007) 186–212, http://dx.doi.org/10.1016/j.parco.2006.12.003.
[36] T.C. Belding, The distributed genetic algorithm revisited, in: Proceedings of the Sixth International Conference on Genetic Algorithms, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1995, pp. 114–121.
[37] D. Whitley, An overview of evolutionary algorithms: practical issues and common pitfalls, Inf. Softw. Technol. 43 (14) (2001) 817–831.
[38] W. Rivera, Scalable parallel genetic algorithms, Artif. Intell. Rev. 16 (2) (2001) 153–168, http://dx.doi.org/10.1023/A:1011614231837.
[39] E. Cantu-Paz, A survey of parallel genetic algorithms, 1997.
[40] R. Tanese, Distributed genetic algorithms, in: Proceedings of the Third International Conference on Genetic Algorithms, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1989, pp. 434–439.
[41] H. Chen, N.S. Flann, D.W. Watson, Parallel genetic simulated annealing: a massively parallel SIMD algorithm, IEEE Trans. Parallel Distrib. Syst. 9 (2) (1998) 126–136, http://dx.doi.org/10.1109/71.663870.
[42] T. Kalinowski, Solving the mapping problem with a genetic algorithm on the maspar-1, in: Proceedings of the First International Conference on Massively Parallel Computing Systems, 1994, pp. 370–374. http://dx.doi.org/10.1109/MPCS.1994.367057.
[43] S. Baluja, A massively distributed parallel genetic algorithm (MDPGA), Tech. Rep., Carnegie Mellon University, 1992.
[44] H. Juille, J.B. Pollack, Co-evolving intertwined spirals, in: Proceedings of the Fifth Annual Conference on Evolutionary Programming, MIT Press, 1996, pp. 461–468.
[45] D. Prabhu, B.P. Buckles, F.E. Petry, A SIMD environment for genetic algorithms with interconnected subpopulations, Scalable Comput. Pract. Exp. 7 (2) (2006) 65–86.
[46] J.T. Ngo, J. Marks, Physically realistic motion synthesis in animation, Evol. Comput. 1 (3) (1993) 235–268, http://dx.doi.org/10.1162/evco.1993.1.3.235.
[47] W. Jiang, J. Liu, H.-W. Jin, D. Panda, W. Gropp, R. Thakur, High performance mpi-2 one-sided communication over infiniband, IEEE International Symposium on Cluster Computing and the Grid, 2004, pp. 531–538. <http://doi.ieeecomputersociety.org/10.1109/CCGrid.1336648>.
[48] A. Faraj, P. Patarasuk, X. Yuan, A study of process arrival patterns for mpi collective operations, in: ICS '07: Proceedings of the 21st Annual International Conference on Supercomputing, ACM, New York, NY, USA, 2007, pp. 168–179. <http://doi.acm.org/10.1145/1274971.1274996>.
[49] R. Cledat, T. Kumar, J. Sreeram, S. Pande, Opportunistic computing: a new paradigm for scalable realism on many-cores, in: HOTPAR '09: USENIX Workshop on Hot Topics in Parallelism. doi:http://dx.doi.org/10.1109/IPDPS.2008.4536264.
[50] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, K.A. Yelick, The landscape of parallel computing research: a view from Berkeley, Tech. Rep., EECS Department, University of California, Berkeley, Dec. 2006.
[51] D. Bader, Petascale Computing: Algorithms and Applications, first ed., Chapman & Hall, CRC, 2007.
[52] E. Alba, J.M. Troya, An analysis of synchronous and asynchronous parallel distributed genetic algorithms with structured and panmictic islands, in: Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing, Springer-Verlag, London, UK, 1999, pp. 248–256.
[53] S.-C. Lin, W.F. Punch, E.D. Goodman, Coarse-grain parallel genetic algorithms: categorization and new approach, Proceedings. Sixth IEEE Symposium on Parallel and Distributed Processing (26–29 Oct 1994), 1994, pp. 28–37. http://dx.doi.org/10.1109/SPDP.1994.346184.
[54] G. Folino, C. Pizzuti, G. Spezzano, A scalable cellular implementation of parallel genetic programming, IEEE Trans. Evol. Comput. 7 (1) (2003) 37–53, http://dx.doi.org/10.1109/TEVC.2002.806168.

[55] V.S. Gordon, L.D. Whitley, Serial and parallel genetic algorithms as function optimizers, in: Proceedings of the Fivfth International Conference on Genetic Algorithms, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993, pp. 177–183.
[56] J.R. Clausen, D.A. Reasor, C.K. Aidun, Parallel performance of a lattice-Boltzmann/finite element cellular blood flow solver on the IBM blue gene/p architecture, Comput. Phys. Commun. 181 (6) (2010) 1013–1020, http://dx.doi.org/10.1016/j.cpc.2010.02.005.
[57] J. Dongarra, D. Gannon, G. Fox, K. Kennedy, The impact of multicore on computational science software, CTWatch Q. 3 (1) (2007) 817–831.
[58] D.E. Goldberg, K. Deb, A comparative analysis of selection schemes used in genetic algorithms, in: G.J.E. Rawlins (Ed.), Foundations of Genetic Algorithms, Morgan Kaufman, San Francisco, CA, 1991, pp. 69–93.
[59] B. Manderick, P. Spiessens, Fine-grained parallel genetic algorithms, in: Proceedings of the Third International Conference on Genetic Algorithms, Morgan Kaufmann Publishers Inc., 1989, pp. 428–433.
[60] D. Andre, J.R. Koza, Parallel genetic programming: a scalable implementation using the transputer network architecture, in: K.E. Kinnear, L. Spector, P.J. Angeline (Eds.), Advances in genetic programming, vol. 2, MIT Press, Cambridge, MA, 1996, pp. 317–337.
[61] MPI-Forum, MPI: A Message-Passing Interface Standard, Version 3.0, High Performance Computing Center Stuttgart (HLRS), Stuttgart, Germany, 2012.
[62] M. Mascagni, A. Srinivasan, Algorithm 806: Sprng: a scalable library for pseudorandom number generation, ACM Trans. Math. Softw. 26 (3) (2000) 436–461. <http://doi.acm.org/10.1145/358407.358427>.
[63] M.M. Amini, M. Racer, A rigorous computational comparison of alternative solution methods for the generalized assignment problem, Manage. Sci. 40 (7) (1994) 868–890.
[64] X.-H. Sun, L.M. Ni, Another view on parallel speedup, in: Proceedings of the 1990 ACM/IEEE Conference on Supercomputing, Supercomputing '90, IEEE Computer Society Press, Los Alamitos, CA, USA, 1990, pp. 324–333. <http://dl.acm.org/citation.cfm?id=110382.110450>.
[65] V. Sarkar, W. Harrod, A.E. Snavely, Software challenges in extreme scale systems, J. Phys.: Conf. Ser. 180 (1) (2009) 012045.
[66] Cplex, <http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/>, 2013.
[67] E. Alba, G. Luque, J. Troya, Parallel LAN/WAN heuristics for optimization, Parallel Comput. 30 (56) (2004) 611–628, http://dx.doi.org/10.1016/j.parco.2003.12.007.
[68] E. Cantu-Paz, D.E. Goldberg, Efficient parallel genetic algorithms: theory and practice, Comput. Methods Appl. Mech. Eng. 186 (2) (2000) 221–238.
[69] R.K. Ahuja, O. Ergun, J.B. Orlin, A.P. Punnen, A survey of very large-scale neighborhood search techniques, Discrete Appl. Math. 123 (2002) 75–102.