Soft Comput
DOI 10.1007/s00500-013-1140-5

METHODOLOGIES AND APPLICATION

Asynchronous and implicitly parallel evolutionary computation models

Domagoj Jakobović · Marin Golub · Marko Čupić

Abstract This paper presents the design and the application of asynchronous models of parallel evolutionary algorithms. An overview of the existing parallel evolutionary algorithm (PEA) models and available implementations is given. We present new PEA models in the form of asynchronous algorithms and implicit parallelization, as well as experimental data on their efficiency. The paper also discusses the definition of speedup in PEAs and proposes an appropriate speedup measurement procedure. The described parallel EA algorithms are tested on problems with varying degrees of computational complexity. The results show good efficiency of asynchronous and implicit models compared to existing parallel algorithms.

Keywords Evolutionary algorithms · Parallelization · Asynchronous algorithms

1 Introduction

Evolutionary algorithms (EAs) are search algorithms inspired by natural selection that have been shown to be very successful in many applications and in different domains. The use of EAs and other metaheuristics meets the need to generate acceptable solutions for hard optimization problems, where the exact level of satisfiable solution quality must be determined for each application in question.

When utilizing EAs, however, some problems may arise which can be effectively solved with some form of *parallelization*: for some applications the data structures involved in order to evaluate solutions may be very large and need to be distributed among processing units for efficient computation. Solution evaluation can be (and usually is) time consuming, which presents a problem for methods that rely on frequent sampling in search space.

This paper proposes new models of parallel evolutionary algorithms and compares their efficiency, in terms of evolution speedup, with existing algorithms. The described models allow *asynchronous* execution where the same data structure - individuals - may be modified by multiple processing elements at the same time. The distinction is also made between explicit parallel algorithms, which correspond to all existing models, and implicit parallelization, in which some portions of a *sequential* algorithm are executed in parallel. We investigate these models on applications with varying computational demands in different parts of the evolutionary algorithm. These applications pose contradictory requirements regarding parallelization, which allows the comparison of various parallelization methods. The results show that asynchronous and implicit parallelization methods exhibit performance that is not worse, and in some conditions better, than the existing models.

Furthermore, measurement of the speedup of evolutionary algorithms is discussed and an adapted speedup measuring procedure is proposed. The presented models are implemented in an EA framework which allows the deployment of different parallel models without recompilation or code adaptation simply by choosing different configuration parameters.

The paper is organized as follows: the next section discusses categorization of evolutionary algorithms and existing parallel models and architectures. Section 3 describes new parallel models and outlines their implementation. In Sect. 4 the applications being solved are presented, and Sect. 5

Communicated by G. Acampora.

D. Jakobović (✉) · M. Golub · M. Čupić
Faculty of Electrical Engineering and Computing,
University of Zagreb, Zagreb, Croatia
e-mail: domagoj.jakobovic@fer.hr

Journal: 500 MS: 1140 ☐ TYPESET ☐ DISK ☐ LE ☐ CP Disp.:2013/9/30 Pages: 12 Layout: Large

discusses the speedup measuring methodology. Section 6 shows the obtained experimental results and Sect. 7 gives a short conclusion and some perspectives for further work.

2 Parallel evolutionary algorithms

2.1 Evolutionary algorithm models

Evolutionary algorithms, whether serial or parallel, can be divided in two main subclasses: panmictic and structured EAs Alba and Tomassini (2002). In the case of *panmictic* or *global* evolutionary algorithms, selection takes place globally and any individual can compete and mate with any other. Unlike the panmictic one, *structured* evolutionary algorithms deal with subpopulations, where the population is divided into several subpopulations which may or may not overlap.

Two popular classes of panmictic EAs are generational and steady-state algorithms. In a generational model a whole new population of N individuals replaces the old one. Steady-state EA, on the other hand, at every step creates one new individual which is inserted back into the population. Those models may be viewed as two extremes of *generation gap* algorithms: in generation gap algorithms a given number of the individuals M (mortality) are replaced with new ones (generational EAs have a mortality of $M = N$ and steady-state EAs a mortality of $M = 1$).

Widely known types of structured EAs are distributed (DEAs) and cellular evolutionary algorithms (CEAs). DEAs are also called island models or coarse-grained as they deal with isolated subpopulations which exchange individuals. On the other hand, in a CEA, or fine-grained EA, an individual has its own pool of potential mates defined by neighboring individuals (one individual may belong to many pools).

2.2 Parallel evolutionary algorithms and applications

Parallel evolutionary algorithms (PEAs) can be classified into following classes: master-slave or global PEAs, coarse grained or fine grained distributed EAs and hybrid models.

Master-slave or *global PEA* has a single population and usually executes only evaluation in parallel. Nevertheless, slaves can perform all or some of genetic operators on the subset of population which is defined by the master. Each individual may compete (selection) and mate (reproduction) with any other, as in the serial EA. If the master does not access (use in any way) the subset of the population that a slave currently accesses (e.g. evaluates), then the algorithm is *synchronous*. A synchronous master-slave EA has the same properties as a serial EA, except the speed of execution.

Distributed EAs have many names: they are also called multiple-deme parallel EAs, island EAs or coarse grained EAs. DEAs have a relatively small number of demes with many individuals which are occasionally migrated. The migration mechanism requires several additional parameters to be defined: communication topology, migration condition, number of migrants, migrant selection and integration method. The demes themselves may overlap so the same set of individuals belongs to more than one deme Nowostawski and Poli (1999).

Hybrid (or hierarchical) parallel EAs combine some of previously described methods in a single algorithm: examples include multiple-deme models with master-slave algorithms run on each deme, demes divides into smaller subpopulations etc.

2.3 Related work

The above models have been used extensively in a number of applications. Global PEAs have been used on various problems but almost always distributing the evaluation phase only (Cantú-Paz 2007; Cantú-Paz 1998; Borovska 2006), assuming it to be the most time consuming. Distributed EAs are a viable choice with numerous computing nodes being highly available (Park et al. 2008; He et al. 2007; Melab et al. 2006; Alba et al. 2002; Nowostawski and Poli 1999; Alba et al. 2004), and in this model high speedups were easily attainable.

Extended models are presented recently: in (Acampora et al. 2011) the authors apply hierarchical distribution among processor cores and develop a distributed memetic (hybrid) solution for the e-learning experience binding problem, and in (Acampora et al. 2011) apply the model on a distributed system with Gaussian-based migration operator. Without concentrating primarily on speedup, the structured population algorithms may help improve the convergence properties, as shown for distributed differential evolution in (Weber et al. 2011). Rather than the population, the local search operators can also be structured in parallel (Caraffini et al. 2013) by randomly choosing the appropriate operator, although experiments with concurrent execution were not reported.

The fine-grained parallel EAs, on the other hand, appear in a smaller number, as for their implementation often a specialized hardware platform is needed, such as an FPGA programmable array (Eklund 2004).

Considering the available literature, the asynchronous PEA models have not been extensively investigated nor used in practice. To avoid confusion, we will try to clarify the distinction between the synchronous and asynchronous algorithm behavior. In a *synchronous* algorithm, when a data structure (individual) is being accessed by a processing element, e.g. its fitness is being evaluated or its genetic material changed, this data structure cannot be changed by any other processing element. A common example occurs when the master waits for all the workers to finish evaluating the individuals.

Journal: 500 MS: 1140 ☐ TYPESET ☐ DISK ☐ LE ☐ CP Disp.:2013/9/30 Pages: 12 Layout: Large

An *asynchronous* algorithm, on the other hand, is any algorithm that doesn't comply with the above rule, i.e. if multiple processing elements (threads, processes) may access the same data structure at the same time. For example, this may be the use of an individual in crossover or selection (by the master) while its fitness is currently being evaluated (by a worker). This definition should not be confused with asynchronous *migration* between islands in distributed parallel EAs (Alba and Troya (2001)), where a a synchronous algorithm is still employed.

The described behavior is counterintuitive at the first glance, but the motivation may lie in the reduction of *idle time*, which inevitably occurs when a processing element must wait for the data structure to become available. The justification of this approach must therefore be experimentally verified for any given type of asynchronous behavior.

The usability of a parallel model is often increased if a software implementation is available which the user may apply to the specific optimization problem. At the moment, there are a number of frameworks available for parallel evolutionary algorithms; some of the more prominent are shown in Table 1. Most of the existing frameworks offer parallelism only in the form of island models and/or master-slave evaluation parallelization. The deployment of a hybrid (hierarchical) parallel model is generally not readily available with the frameworks, at least not without additional intervention in code. Furthermore, the only object of parallelization offered in master-slave models is the evaluation.

3 New models of parallel evolutionary algorithm

This section describes new parallel EA models and states main differences to the existing ones. We distinguish two parallel algorithm types: explicitly parallel and implicitly parallel algorithms.

3.1 Explicit parallelism

An *explicitly parallel* algorithm presumes execution in more than one instance (more than one process), it may assign different *roles* to different processes and may use 'send' or 'receive' data operations. Explicit parallel algorithms are usually expressed with message passing paradigm using primitives such as 'send/receive individuals', 'send/receive fitness values', 'send/receive control message', 'synchronize' etc. They also have predefined roles for which the total number may be constrained – e.g. only one master and multiple slave processes. The explicit parallelism is the most common way of defining a PEA behavior.

The *implicit parallelism* concept, on the other hand, employs a *sequential* algorithm and, unlike the explicit parallelization, it does not define how the evolution is parallelized, but states what parts of the algorithm should be executed in parallel. The latter approach is described in Sect. 3.2.

For illustration and comparison purposes, we first give an example of an explicitly parallel algorithm, *synchronous generational global parallel EA* (SGenGPEA), listed as Algorithm 1. This algorithm is a 'standard' master-slave PEA, found in almost all PEA implementations, which is coupled with generational selection and distributes only evaluation among slave processes.

SGenGPEA defines two roles: one process as the master and one or more processes as workers. The algorithm is considered *synchronous* since its behavior is equivalent to the sequential generational algorithm it encapsulates (note that it can be used with different variants of generational EAs). It is also considered global (panmictic) since any individual it operates on may interact with any other. The SGenGPEA algorithm will be used as a baseline for further efficiency analysis.

In this context, we present a new explicitly parallel algorithm that is intended to be used with a steady-state replacement mechanism and is denoted as *asynchronous elimina-*

Table 1 Overview of parallel EA implementations

Framework	Parallel models
ParadisEO (Melab et al. (2006), Cahon et al. (2004)) http://paradiseo.gforge.inria.fr/	distributed EA, master-slave with parallel evaluation, single solution parallel evaluation
MALLBA http://www.lsi.upc.es/~mallba/	distributed EA
Distributed BEAGLE (Gagne et al. 2003) http://beagle.gel.ulaval.ca/distributed/	distributed EA, master-slave with parallel evaluation
JDEAL http://laseeb.isr.ist.utl.pt/sw/jdeal/	master-slave with parallel evaluation
ECJ http://www.cs.gmu.edu/~eclab/projects/ecj/	distributed EA, master-slave with parallel evaluation, multithreaded evaluation or operators
EvA2 http://www.ra.cs.uni-tuebingen.de/software/EvA2/	distributed EA
DREAM http://www.dr-ea-m.org/	distributed EA

Journal: 500 MS: 1140 ☐ TYPESET ☐ DISK ☐ LE ☐ CP Disp.:2013/9/30 Pages: 12 Layout: Large

Algorithm 1 Synchronous generational global parallel EA - SGenGPEA

Role: MASTER (single)
while evolution not done **do**
  perform one generation of a generational EA (without evaluation)
  send *jobsize* individuals to each WORKER;
  while individuals to evaluate **do**
    receive fitness values from a WORKER;
    send *jobsize* individuals to the WORKER;
  end while
  while all fitness values not received **do**
    receive fitness values from a WORKER;
  end while
end while

Role: WORKER (many)
while evolution not done **do**
  receive individuals from MASTER;
  evaluate individuals;
  send fitness values to MASTER;
end while

Algorithm 2 Asynchronous elimination global parallel EA - AEliGPEA

Role: MASTER (single)
while evolution not done **do**
  while generation not done **do**
    repeat
      perform one iteration of steady-state EA, without evaluation (i.e. produce and replace one individual)
    until *jobsize* iterations performed;
    receive fitness values from a WORKER;
    send *jobsize* individuals to the WORKER;
  end while
end while

Role: WORKER (many)
signal ready status to MASTER;
while evolution not done **do**
  receive individuals from MASTER;
  evaluate individuals;
  send fitness values to MASTER;
end while

tion global parallel EA (AEliGPEA), defined in listing as Algorithm 2. The algorithm is *asynchronous* since the master does not wait for the worker process to return the fitness values of new individuals. Hence, the selection operator (in a steady-state EA) may use *inconsistent* individuals that are not yet evaluated, i.e. whose fitness is not yet received by the master process.

This raises the question of the correctness of the algorithm: there will obviously be situations in which an individual with good genotype may be eliminated and an individual with bad genotype may be selected for mating. The motivation behind this is the reduction of idle time in worker processes with respect to the synchronous version: while some of the evaluations performed by the workers may indeed be wasted (since the individuals in question may already be changed/eliminated at the master), the greater amount of time that workers spend computing could remedy the slowed evolution process. The effectiveness of this approach is experimentally verified in this work.

A similar concept, but with multiple threads on a shared-memory parallel architecture, was described in (Golub and Budin 2000 and Golub et al. 2001). This approach is also different from steady-state distributed evaluation as implemented in ECJ framework and described in (Sullivan et al. 2008), where the individuals to be evaluated do not replace existing ones immediately, but only when they are received from the workers, which is suitable for high latency environments. Furthermore, the experiments with steady-state parallelization were not performed in (Sullivan et al. 2008).

The problems that could benefit from both these algorithms are the ones in which the evaluation phase has a high time complexity, which is a common feature of many EA applications. If this is not the case, we propose the use of implicit parallelism, as described in the following section.

3.2 Implicit parallelism

The *implicit parallel* model uses a *sequential* algorithm, but with certain predefined parts of the algorithm being executed in parallel. This is an extension of the master-worker model where the user is expected only to decide what part(s) of the evolution should be performed by the workers. The parallelization is implicit since the object of parallelization is not defined within the evolutionary algorithm.

This approach is both an algorithmic and an implementation issue, where the goal is that the technique be independent of the chosen type of evolutionary algorithm. In the EC framework implementation accompanying this work, this is made possible with a high level of abstraction used to implement an algorithm: all evolutionary operators are realized with constructs such as 'mate individuals', 'mutate', 'evaluate', 'replace with' etc. The operator details, such as individual structure, types of crossover and mutation rates, are specified in the configuration file (algorithm independent).

The levels of implicit parallelism explored in this work are:

- **evaluation**: the calculation of fitness is distributed among worker processes;
- **operators** (whether mutation, crossover or a local search operator): the desired genetic operators and subsequent evaluation are executed in parallel.

In both cases the procedure is similar: the parallel subsystem intercepts the function calls for evaluation or genetic operators that the sequential algorithm uses. The individuals (data structures) included in the operation are then sent to workers in groups defined by the *jobsize* parameter (or its

Journal: 500 MS: 1140 ☐ TYPESET ☐ DISK ☐ LE ☐ CP Disp.:2013/9/30 Pages: 12 Layout: Large

default value). When the individuals are returned, their fitness and/or genotype is updated accordingly. This method can be used with any selection mechanism. For example, if the evaluation is implicitly parallelized by choice, Algorithm 3 illustrates what happens every time a new individual is sent to implicit evaluation:

Algorithm 3 Implicit parallelization - evaluation level

```
store new individual;
if jobsize individuals are stored then
  if a worker is ready then
    receive evaluated individuals;
    send new individuals to worker;
  else
    evaluate one individual locally;
  end if
end if
```

The described scheme is obviously asynchronous, since the sequential algorithm may use the affected individuals in the meantime. However, a *synchronous* implicit parallelism is also possible: in this mode the individuals the workers will operate on are temporarily removed from the population (so the population size temporarily decreases). Removing the individuals actually means that they cannot be chosen by any selection operator available to the algorithm until they are returned to the population. The removal and subsequent insertion of individuals take place only during the intercepted call to evaluation or genetic operators.

Of course, the implicit asynchronous method with only evaluation in parallel may be functionally equivalent to the explicitly asynchronous algorithm if the same steady-state EA is used as the base.

The motivation for implicit parallelism can be justified with the reasoning that the average EC user may not always be familiar with the details of the evolutionary process, other than the applied genetic operators and the fitness function. However, if the user has the basic knowledge of the *time complexity* of the problem components – i.e. the evaluation, crossover, mutation or local operators – then he may simply choose the component that should be executed in parallel.

Furthermore, the user may encounter a situation where one variant of sequential evolutionary algorithm achieves better results than the other available evolutionary algorithms. In that case the user may want to parallelize that particular algorithm, for which there may not exist an appropriate explicit parallel version. With implicit parallelization we can thus avoid writing a customized parallel algorithm for a specific problem, and instead just select which algorithm components should be distributed. The asynchronous nature may in this case obviously lead to a deterioration in the rate of evolution, but it can still be useful if the distribution of time consuming operations is effective enough, which is investigated in the Results.

Table 2 Parallel evolutionary algorithm models

PEA property	variants
algorithm	- globally parallel EA (master/slave)
	- distributed EA (coarse grained)
	- massively parallel EA (fine grained)
	- hybrid PEA
model	- panmictic (single deme)
	- structured (multiple demes)
synchronicity	- synchronous
	- asynchronous
parallelization	- explicit
	- implicit

3.3 Extension to multiple deme models

The models described in previous sections presume execution on a single *deme*, i.e. where the population consists of a single set of individuals that may interact freely. In contrast, EAs have been extensively used in multiple demes (structured EA) where each deme evolves independently but with the inclusion of the migration operator that exchanges individuals between different demes.

Each of the previously described single deme models can be employed with a multiple-deme population: with a *sequential* algorithm that runs on each deme, we get what is widely known as a distributed evolutionary algorithm (DEA). If we use an explicitly parallel algorithm, then the same parallel algorithm operates on each deme. Finally, a sequential algorithm may be specified along with an implicit parallelization option, which results in that algorithm being parallelized on each deme. The last two cases correspond to a hybrid distributed EA with the deme or island model at the higher and a master-worker algorithm at the lower level. A concise overview of parallel EA properties and corresponding variants is given in Table 2.

3.4 Evolutionary computation framework

All the parallel models presented in this work are implemented as components of the Evolutionary Computation Framework (ECF), a C++ framework for the use and development of various EC methods. Current version of the framework is available at http://gp.zemris.fer.hr.

4 Test problems

This section covers the applications on which the described parallel models were tested.

Journal: 500 MS: 1140 TYPESET DISK LE CP Disp.:2013/9/30 Pages: 12 Layout: Large

The distinctive elements of the problems are the genotypes, associated operators and the evaluation function, and it is these that in most cases guide the selection of an appropriate parallelization method.

4.1 Evolution of scheduling heuristics with genetic programming

The first application is an example of machine learning with the goal of finding a suitable scheduling heuristics. Due to inherent problem complexity and variability, a large number of scheduling systems employ heuristic scheduling methods. Among many available heuristic algorithms, the question arises of which heuristic to use in a particular environment, given different performance criteria and user requirements. A solution to this problem may be provided using genetic programming to create problem specific scheduling algorithms.

In this application the *priority scheduling* paradigm is used: the jobs (activities) are selected to start based on their priority value. Priority values are, in turn, determined with a *priority function* that the user must choose and it is this choice that has the greatest impact on the effectiveness of scheduling process. The task of genetic programming is to find a priority function which is best suited for given user-defined criteria and scheduling environment (the solution is represented with a tree that embodies the priority function). A single priority function, once evolved, is used to schedule unseen sets of scheduling problems and thus compared with the existing human-made heuristics.

Although we have experimented with many different scheduling criteria and environments (Jakobovic and Budin 2006; Jakobovic et al. 2007; Jakobovic and Marasovic 2012), here we employ static scheduling on one machine with minimization of weighted tardiness (this is an NP-complete problem whose solution can also be used in more complex multiple machine environments (Morton and Pentico 1993).

This problem is a representative of evaluation-critical applications, since each GP tree must be interpreted many times to generate schedules for all the test cases: every time a job is to be scheduled, the same tree is used to calculate priorities of all the unscheduled jobs. It is expected that this will favor the parallel models that distribute evaluation among the processors.

4.2 Game strategy evolution using genetic programming

In this application the goal of GP is to evolve a game strategy for a card game (blackjack in this case). The automated computer player must decide upon the next action in the game, which may be ‘hit’, ‘stand’, ‘double-down’ or ‘split’. The decision is based on the current state of the player cards and a single visible dealer’s card. Genetic programming builds a separate decision tree for the first two actions (‘hit’ or ‘stand’) and an additional tree for the other options. The tree functions are based on logical and arithmetic operators, whereas the terminals describe the player’s or dealer’s card values. The fitness function is expressed as the normalized score the simulated player achieved in a predefined number of games. Unlike the other two applications, this example is a maximization problem.

This application also spends the most of the processor time in evaluation, although not as much as in the previous example. It is therefore expected that the distribution of evaluation would still be the best option for parallelization.

4.3 Function approximation using genetic algorithm

The third application is a GA example of function approximation (Schneburg et al. 1995; Golub and Posavec 1997). The task to be solved is to interpolate the given function g through an arbitrary time series $T = (x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ where n is the number of points and $y_i < y_{max}$. The approximation function g is given as:

$$g(x) = a_0 + a_1 x + \sum_{i=1}^{N_S} \left[a_{3i-1} \cdot \sin\left(a_{3i} x + a_{3i+1}\right)\right], \qquad (1)$$

where N_S is the number of sine elements.

The problem can be stated as finding the minimal sum of squares of deviations for the function g and the given time series T. Therefore, the goal function to be minimized is:

$$f(a_0, a_1, \ldots, a_{3N_S+2}) = \sum_{i=1}^{n} \left[g(x_i) - y_i\right]^2 \qquad (2)$$

In this example the GA implementation includes a local search procedure that takes place every time after an individual has been mutated. The local search loops over every coefficient of the individual and calculates the fitness values with a small change in each direction (similar to a pattern search). If a better value is found, the coefficient is simply updated.

This problem does not pose great time complexity on fitness evaluation (Golub 2001), but mutation with the local search operator is the most time critical component since it includes additional evaluations.

5 Experimental setup

5.1 Properties of speedup for parallel evolutionary algorithms

One goal of our experiments was to measure the effectiveness of presented parallel models when applied to different problems, and the most usual measure is the achieved speedup.

Although speedup is is a well-accepted way of measuring parallel algorithm's efficiency, its definition and interpretation for evolutionary algorithms has been vague and sometimes controversial. The traditional definition of speedup relates the execution time of the best sequential algorithm T_1 to that of the parallel algorithm being run on m processors T_m as $S_m = \frac{T_1}{T_m}$.

Speedup value equal to the number of processors is considered linear; smaller value indicates sub-linear speedup and greater value super-linear speedup. In evolutionary computation the execution time is considered stochastic, so the obvious adaptation to the traditional definition is the use of average execution times over independent runs. But this definition doesn't cover the following issues: what variants of sequential and parallel EA should be compared and what is the adequate termination condition of those algorithms?

To answer these questions, we must further explore the variants of speedup measure. The traditional speedup is considered *strong* (or absolute) if a parallel algorithm is compared against the *best* available sequential algorithm for the problem and *weak* (or relative) otherwise, i.e. against a sequential algorithm that solves the problem but is not proved to be the optimal. In the context of EC, the only practical way is to use the weak speedup, since the strong definition requires the researcher to be aware of the fastest algorithm solving any of the problems being tackled. In other words, weak speedup usually means comparing against a sequential evolutionary algorithm.

Another point to consider is the type of sequential algorithm: the traditional definition of speedup presumes that the sequential algorithm is *equivalent* to the parallel version being run on a single processor. For instance, if the parallel algorithm is a multiple deme model, than the sequential version should be an identical evolutionary algorithm: in order to have a fair and meaningful speedup, we need to consider the same algorithm and then only change the number of processors from 1 to m.

Finally, we need to decide upon the stopping criterion of the algorithms: a simple approach would impose a predefined number of iterations or a predefined number of evaluations to both. These methods are *not* considered fair (Alba and Tomassini 2002), since they may compare two algorithms that are producing solutions of different quality. The obvious adaptation is to stop the algorithms when a solution of the same quality had been found, usually an optimal solution. This is, in fact, the recommended way of measuring the speedup for parallel EAs (Alba and Tomassini 2002; Alba 2002), where the tested measure is *convergence rate* instead of execution time.

However, since in most real-world problems (including the ones presented in this paper) no known optimal solution may exist, we are left with the option of using a predefined solution quality as a termination criterion. This raises the question of choosing a particular quality for the given problem, since it is obvious that using different quality levels may result in different speedups. Furthermore, having the *same* fitness value as termination condition for *every run* may prove impractical, since for the majority of problems the evolutionary algorithm is not guaranteed to converge to a particular solution in a finite amount of time. In any case, we might be forced to average over greatly varying values of execution times to calculate the speedup.

5.2 Speedup definition and measurement

For these reasons, we propose the following speedup measurement procedure: for each problem being solved and the employed parallel algorithm, we have to define an *acceptable* quality level that we want the algorithm to reach – the one close but usually not equal to the optimum (if it is known), or the one found by previous experiments in the field. Then, the algorithm being tested - either sequential or parallel - is to be started in a number of instances (runs) that are terminated only when the *median* of the current fitness values of *best individuals* in each instance reaches the desired quality level. The obtained termination time is then recorded as algorithm execution time T_m and used to calculate the speedup. In other words, it is not the execution time that is averaged, but the solution quality of the algorithm in question, over multiple algorithm runs.

This definition of speedup measurement does not exclude the one where all the algorithms are stopped when they reach the *same* quality level. Note that different quality levels may be defined for different evolutionary algorithms even on the *same problem*, since the algorithms may exhibit very different convergence properties. For example, one algorithm may not be able to converge to the fitness value that the other achieves - and is therefore not a good choice for the problem – but the weak (relative) speedup can still be determined.

5.3 Speedup measure–a case study

To justify the proposed speedup definition, an example is shown that demonstrates the properties of different speedup measures. The example compares two explicitly parallel algorithms, the synchonous SGenPEA (Algorithm 1) as a baseline and the asynchronous AEliGPEA (Algorithm 2), both applied to the GP scheduling problem (4.1). The SGenPEA is compared to the generational sequential algorithm, and the AEliGPEA to the equivalent elimination one.

The first issue that must be considered is the convergence properties of the algorithms, since for this problem they differ by a considerable margin. The *sequential* algorithms are compared for the same number of evaluations (50000) and in 30 independent runs for both; the generational algorithm achieves the mean best fitness result of 464.1 with a standard deviation $\sigma = 1.9$, whereas the elimination algorithm achieves 460.4 with $\sigma = 0.7$. The t-test on the results rejects the probability of those two populations originating from the same distribution with p value < 0.0001. This may be another indicator in favor of the implicit parallelization approach, since we would obviously like to parallelize the more effective baseline sequential algorithm, for which an equivalent parallel implementation may not be available.

Secondly, we have to define an appropriate fitness value for each of the algorithms. In practice, this includes choosing an acceptable level of quality, but in our experiments we used the following metric: for every parallel algorithm, the average best fitness value of the corresponding sequential algorithm is increased (for minimization problems) by the standard deviation, and the resulting value is used. In our example, the value of 466 is used for the generational, and the value 461.1 for the elimination parallel algorithm.

Finally, the speedup of both algorithms is calculated using the following measures:

1. time ratio for the same number of generations (100);
2. time ratio for the same number of evaluations (50000);
3. the proposed measure based on the median of best individuals.

The additional measure, where each run is stopped only when the algorithm reaches the same fitness value, was not applicable for this problem, since not every algorithm instance was able to converge to the desired level in a reasonable amount of time. Even if we discard the runs that didn't converge, the resulting execution times can vary from less than an hour to several days, which is clearly not a good basis for comparison.

The speedup results for both algorithms are given in Fig. 1 and 2 and in Table 3 (the dotted line in the figures indicates the linear speedup level). For the synchronous generational algorithm, the first two measures give the same results, but may yield slightly different values for the asynchronous one (since the master doesn't wait for the workers, number of generations may not be related to the number of evaluations). It can be seen that for the asynchronous algorithm the first two measures are not adequate and don't show the actual progress in solution quality; the proposed median based measure is in this case more reliable. At the same time, the speedups obtained with the median based measure exhibit similar properties as the traditional measures for the synchronous algorithm.

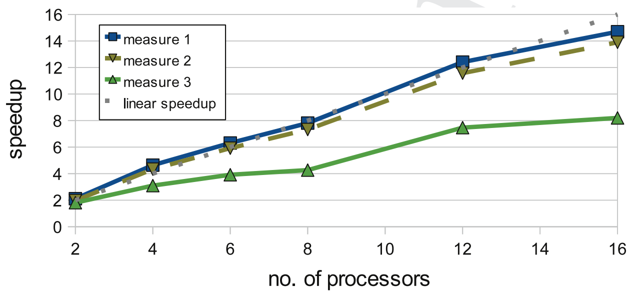

Fig. 1 Speedup results for AEliGPEA, different speedup measures

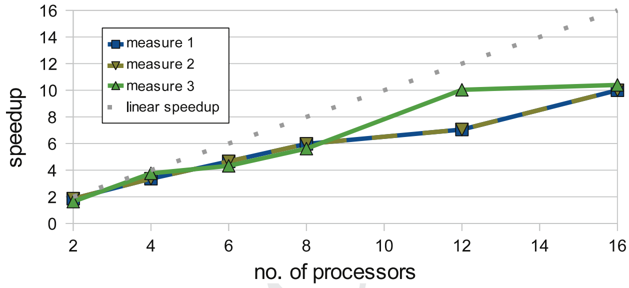

Fig. 2 Speedup results for SGenGPEA, different speedup measures

To further explore the proposed speedup definition, we compared the obtained sets of best values at the moment the median value reaches the designated threshold. In other words, the set of best values that an algorithm produces on a given number of processors is compared to the set that the same algorithm produces on different number of processors, at the moment of reaching the same median. The sets obtained in this way should *not* be statistically different, because they describe the algorithm's convergence rate at a given moment. A pairwise comparison of all sets for both parallel algorithms and for every tested number of processors is performed. In the case of AEliGPEA, the tests show no significant difference with p values of at least 0.25, and for SGenGPEA the corresponding p values are greater than 0.43.

In the remainder of the text, only the median based measure will therefore be used.

6 Results

This section gives the experimental results on the described problems and presented models of parallel algorithms. In all the experiments the implementation has been compiled and executed using the MPICH2 library with the *socket* communication channel. By default, one MPI process is assigned to a single processor in all experiments. In all the experiments the 'jobsize' parameter was set depending on the number of workers so that approximately a quarter of the population is deployed at the workers at any time (e.g. for population size of 500 with 5 workers, the 'jobsize' parameter equals 25). For the speedup measurement, every point in the following graphs is generated from at least 30 algorithm runs on each number of processors.

6.1 GP Scheduling problem

The first set of experiments for this problem considers the two explicitly parallel algorithms (see Sect. 3.1). Using the

Table 3 Case study - different speedup measures

algorithm	SGenGPEA						AEliGPEA					
num. proc.	**2**	**4**	**6**	**8**	**12**	**16**	**2**	**4**	**6**	**8**	**12**	**16**
measure 1	1.9	3.4	4.7	6.0	7.1	10.0	2.1	4.6	6.3	7.8	12.4	14.7
measure 2	1.9	3.4	4.7	6.0	7.1	10.0	1.9	4.1	5.9	7.3	11.6	13.9
measure 3	1.6	3.8	4.3	5.6	9.9	10.4	1.8	3.1	3.9	4.3	7.5	8.2

method described in the previous section, each parallel algorithm speedup is computed with comparison to the appropriate baseline sequential algorithm; synchronous generational algorithm (SGenGPEA) is compared to a generational roulette-wheel algorithm and asynchronous steady-state algorithm (AEliGPEA) to a steady-state 3-tournament worst elimination algorithm.

Different fitness values are used as termination criterion for those algorithms (see Sect. 5.3) since the steady-state algorithm (both sequential and parallel) yielded significantly better results. It is of course possible that with some other combination of parameters the generational algorithm would achieve better results, but we didn't perform a detailed parameter state analysis in this context. Furthermore, the difference in absolute performance may be another motivation for the use of a specific type of parallel algorithm (asynchronous one in this case).

For the speedup measurement, the sequential algorithm is run until the median of the best individuals' fitness values reached the designated quality value and then the same procedure is repeated for the parallel algorithm with different number of processors. The population size for this problem was set to 500. The speedups for both algorithms are presented in Fig. 3 (note that these are the same values as in Fig. 1 and 2, measure 3). It can be seen that the asynchronous parallel algorithm scales similarly to the synchronous version for this problem, besides providing better convergence.

Although in asynchronous algorithm a portion of evaluations performed by the workers is wasted, it makes up for this by reducing the idle time at worker processes. Since the master process doesn't wait for all the workers to finish, it can proceed with evolution and generate new work packets for the workers to evaluate. For instance, at 8 processors, the average worker idle time is about 15 % for the SGenGPEA, and only 3 % for the AEliGPEA.

Another set of experiments for this problem was conducted with implicit parallelization, where parts of the algorithm are conducted in parallel (by choice through a parameter in the configuration file). Since the implicit parallelization requires a *sequential* algorithm, experiments are performed with steady-state algorithm as the basis for comparison. In this problem we applied parallelization of evaluation, which is the most time consuming operation in this example. The implicit parallel generational algorithm is run in asynchronous and synchronous mode (Sect. 3.2) and the results for

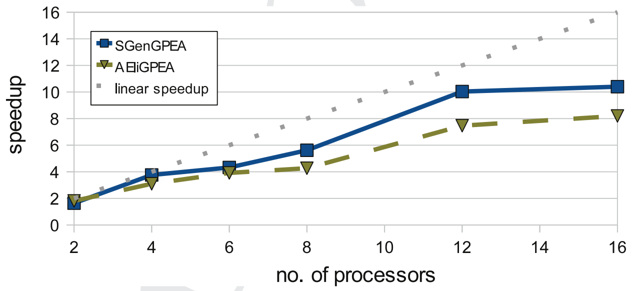

Fig. 3 Speedup results for SGenGPEA and AEliGPEA, GP scheduling

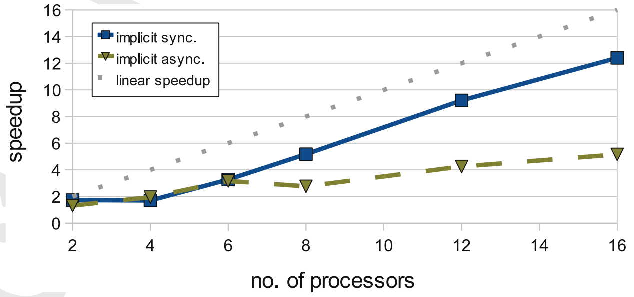

Fig. 4 Speedup results for implicit parallelization, GP scheduling

both versions are shown in Fig. 4. It can be perceived that the implicit parallel evaluation achieved results similar to the AEliGPEA, with an advantage for the synchronous version.

6.2 GP game strategy

For this application we experimented with explicitly parallel algorithms, SGenGPEA and AEliGPEA, and the number of individuals was set to 500 for all algorithms. The first step includes defining the quality levels to be used as median values for algorithm termination. Again the performance was varying depending on the selection method: the generational sequential algorithm achieved mean best value of 19.2 with $\sigma = 1.6$ and the steady-state algorithm yielded the value of 22.4 with $\sigma = 2.1$ (the results exhibit a statistically significant difference with p value <0.001). According to the described approach, we chose the fitness values that correspond to the mean best value, decreased (since this is a maximization problem) by the standard deviation. In other words, the quality level was set at 17.6 for the SGenGPEA and at 20.3 for the AEliGPEA.

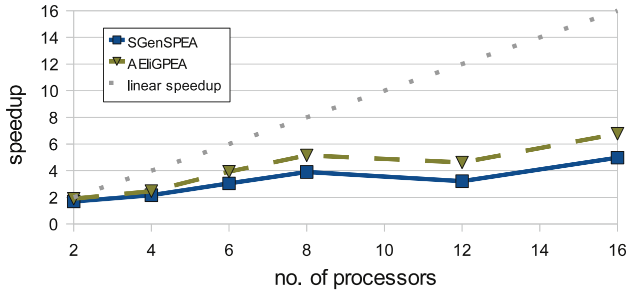

Fig. 5 Speedup results for SGenGPEA and AEliGPEA, GP game strategy

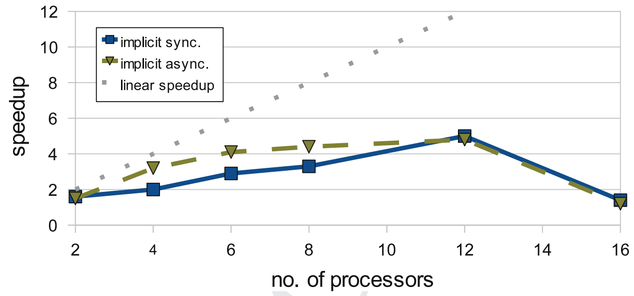

Fig. 6 Speedup results for implicit parallel mutation and local search, GA approximation

The parallel algorithms were run on multiple processors, in at least 30 instances for each number of processors, until the median of best individuals reaches the designated level. The results for both algorithms are shown in Fig. 5. It can be seen that the speedup values are somewhat lower than those in the previous example, which is due to a lesser complexity of the evaluation function. Nevertheless, the asynchronous parallel algorithm scaled similarly to the synchronous variant.

6.3 GA Approximation problem

For the approximation problem, the fitness value is represented as a summed square error over the points of the time series. This problem is an example of application where the evaluation does not stand out in time complexity as compared to other elements of the algorithm. In fact, previous analysis has shown (Golub 2001) that the most part of processor time may be spent on mutation and local search, depending on the parameters.

Although the local search operator slows down the algorithm, its effects are most beneficial: the average best fitness without local search is 128.4 with $\sigma = 45$, whereas with local search the obtained value reaches 19.8 with $\sigma = 41$; statictically significant difference is confirmed with p value <0.0001.

In speedup experiments the desired quality level was hence set to 60, equal to the mean value increased by the deviation. It should be mentioned that the convergence of EA for this problem varies greatly over the runs: we were able to achieve fitness values as low as 0.01, but on some runs the algorithm didn't reach below several hundred.

With these conditions, the explicit algorithms that parallelize only evaluation take a *longer* time than the sequential version of the algorithm. For instance, the synchronous SGenGPEA run on 4 processors shows that an average worker will spend 1.6 % time on communication (including reading and writing the individuals), 93.5 % idle time and only 4.9 % time doing useful work - evaluation. This example serves as a test case to which the usual parallel implementations may not be well adapted.

Therefore, we applied the implicit parallelization method with *mutation and local search* distributed among the worker processes, since these operations were identified as the most time consuming. In this model, the workers also perform evaluation on the received individuals before they are returned to the master. The speedup measurement was made using the steady-state 3-tournament elimination sequential algorithm as the basis for comparison, and the same algorithm was implicitly parallelized. The speedup results are shown in Fig. 6.

Implicit parallelization achieves good results for a smaller number of processors, but its scalability is clearly limited in this case: the population size of 300 and number of processors greater than 8 results in a relatively small amount of computation for a single job and high communication load at the master process. For example, at 8 processors, the workers spend about 10 % time for communication, 37 % time for computation and are idle 53 % of the time.

On the other hand, a hybrid distributed EA may overcome this limitation. For illustration purposes, with a hybrid DEA with 3 demes and implicit parallelization in 4 processes on each deme (a total of 12 processors) we obtained a speedup value of 10.7.

7 Conclusions and future work

This paper describes new parallel evolutionary algorithm models, an asynchronous parallel algorithm and implicit parallelization, that offer additional options for problems where existing master-slave models may not achieve the desired level of efficiency.

An emphasis is put on asynchronous parallel algorithms where the selection can act on individuals whose fitness does not match the current genotype. While this may obviously impair the convergence rate, the experiments show that the overall speedup is comparable to that of the synchronous algorithms. At the same time, an asynchronous algorithm

Journal: 500 MS: 1140 ☐ TYPESET ☐ DISK ☐ LE ☐ CP Disp.:2013/9/30 Pages: 12 Layout: Large

may show better convergence depending on the problem at hand.

The concept of implicit parallelization is also introduced, in which the desired algorithm elements are distributed to worker processes. The main motivation behind the approach is twofold: the user should not be limited with the choice of existing parallel algorithms when he wants to speed up a sequential algorithm with good convergence properties. The second issue is the possibility of identification of the most time consuming element of the algorithm and its automatic parallelization (without additional implementation), whether in synchronous or asynchronous manner.

Both new parallel models have the same disadvantage that all master-slave models share: at some point the master process *will* become a bottleneck as the number of processes is increased. A hybrid DGA with master-slave algorithm at every node may in that case still perform efficiently.

The main contributions of this paper could be summarized as follows: (1) asynchronous parallel algorithms are shown to be a viable alternative to traditional models; (2) the implicit parallelization concept is introduced and tested; and (3) an appropriate speedup measure for evolutionary algorithms is defined based on the convergence rate.

Although in our implementation the parallel algorithms are implemented with message passing between processes, the presented models can also be realized using multithreading technology on multi-core machines that are widely available, which could reduce the communication cost. The combination of multiple demes distributed on workstations where each deme runs a (possibly asynchronous) multithreaded PEA could prove most efficient and is hence a future area of research.

References

Acampora G, Gaeta M, Loia V (2011) Combining multi-agent paradigm and memetic computing for personalized and adaptive learning experiences. Comput Intell 27(2):141–165

Acampora G, Gaeta M, Loia V (2011) Hierarchical optimization of personalized experiences for e-learning systems through evolutionary models. Neural Comput. Appl. 20(5):641–657. doi:10.1007/s00521-009-0273-z

Alba E (2002) Parallel evolutionary algorithms can achieve super-linear performance. Inf Process Lett 82:7–13

Alba E, Luna F, Nebro AJ (2004) Parallel heterogeneous genetic algorithms for continuous optimization. Parallel Comput 14:2004

Alba E, Nebro AJ, Troya JM (2002) Heterogeneous computing and parallel genetic algorithms. J Parallel Distrib Comput 62(9):1362–1385

Alba E, Tomassini M (2002) Parallelism and evolutionary algorithms. IEEE Trans Evol Comput 6:443–462

Alba E, Troya JM (2001) Analyzing synchronous and asynchronous parallel distributed genetic algorithms. Future Gener Comput Syst 17(4):451–465

Borovska, P (2006).: Solving the travelling salesman problem in parallel by genetic algorithm on multicomputer cluster. In: international conference on computer systems and technologies – CompSysTech06, pp. 11–1-11-6.

Cahon, S., Melab, N., Talbi, E.G (2004) Building with paradiseo reusable parallel and distributed evolutionary algorithms. Parallel Computing 30(5–6), 677–697. Parallel and nature-inspired computational paradigms and applications.

Cantú-Paz, E (1998) Designing efficient master-slave parallel genetic algorithms. In: genetic programming 1998: proceedings of the third annual conference, Morgan Kaufmann, University of Wisconsin, USA, pp 455.

Cantú-Paz, E (2007) Parameter setting in parallel genetic algorithms. In: Parameter setting in evolutionary algorithms, pp. 259–276.

Caraffini F, Neri F, Iacca G, Mol A (2013) Parallel memetic structures. Inf Sci 227:60–82

Eklund, S.E (2004) A massively parallel architecture for distributed genetic algorithms. Parallel computing 30(5–6), 647–676. (Parallel and nature-inspired computational paradigms and applications).

Gagne, C., Parizeau, M., Dubreuil, M (2003) Distributed beagle: an environment for parallel and distributed evolutionary computations. In: proceedings 17th annual international symposium of high performance computing systems and applications (HPCS).

Golub, M (2001) Improving the efficiency of parallel genetic algorithms, Ph.D. thesis, Faculty of Electrical Engineering and Computing, Zagreb, Croatia.

Golub, M., Budin, L (2000) An asynchronous model of global parallel genetic algorithms. In: C. Fyfe (ed.) Proceedings of 2nd ICSC Symposium on Engineering of Intertnational Systems, EIS2000, pp. 353–359. ICSC Academic Press, UK.

Golub, M., Jakobovic, D., Budin, L (2001) Parallelization of elimination tournament selection without synchronization. In: proceedings of the 5th IEEE international conference on intelligent engineering systems INES 2001, pp. 85–89. Institute of Production Engineering, Helsinki, Finland.

Golub, M., Posavec, A.B (1997) Using genetic algorithms for adapting approximation functions. In: proceedings of the international conference ITI '97, University Computing Centre, University of Zagreb, Pula, pp. 451–456.

He H, Skora O, Salagean A, Mkinen E (2007) Parallelisation of genetic algorithms for the 2-page crossing number problem. J Parallel Distrib Comput 67(2):229–241

Jakobovic D, Budin L (2006) Dynamic scheduling with genetic programming. Lect Notes Comput Sci 3905:73

Jakobovic D, Jelenković L, Budin L (2007) Genetic programming heuristics for multiple machine scheduling. Lect Notes Comput Sci 4445:321–330

Jakobovic D, Marasovic K (2012) Evolving priority scheduling heuristics with genetic programming. Appl Soft Comput 12(9):2781–2789. doi:10.1016/j.asoc.2012.03.065

Melab, N., Cahon, S., Talbi, E.G (2006) Grid computing for parallel bioinspired algorithms. J Parallel Distrib Comput 66(8), 1052–1061 . (Parallel Bioinspired Algorithms)

Morton, T.E., Pentico, D.W (1993) Heuristic Scheduling Systems. Wiley Inc., USA.

Nowostawski, M., Poli, R (1999) Dynamic demes parallel genetic algorithm. In: KES'99, proceedings of the international conference, IEEE, pp. 93–98.

Nowostawski, M., Poli, R (1999) Parallel genetic algorithm taxonomy. In: Proceedings of the third International Conference knowledge-based intell information engng systems KES'99, pp. 88–92. IEEE.

Park HH, Grings A, dos Santos MV, Soares AS (2008) Parallel hybrid evolutionary computation: automatic tuning of parameters for parallel gene expression programming. Appl Math Comput 201(1–2):108–120

Schneburg, E., Heinzmann, F., Feddersen, S (1995) Genetische Algorithmen und Evolutionsstrategien. 978–3893194933. Addison-Wesley, Verlag.

Journal: 500 MS: 1140 ☐ TYPESET ☐ DISK ☐ LE ☐ CP Disp.:2013/9/30 Pages: 12 Layout: Large

Sullivan, K., Luke, S., Larock, C., Cier, S., Armentrout, S (2008) Opportunistic evolution: efficient evolutionary computation on large-scale computational grids. In: proceedings of the 2008 conference on genetic and evolutionary computation, GECCO '08, ACM, New York, pp 2227–2232 .

Weber M, Neri F, Tirronen V (2011) Shuffle or update parallel differential evolution for large-scale optimization. Soft Comput 15(11):2089–2107