A Partially Asynchronous Global Parallel Genetic Algorithm

Darren M. Chitty
Aston Lab for Intelligent Collectives Engineering (ALICE)
Aston University, Birmingham, UK
darrenchitty@googlemail.com

ABSTRACT

Population-based meta-heuristics such as Genetic Algorithms (GA) are ideal for exploiting multiple processor cores. With parallel architectures now standard computationally intensive methods need to harness them to best effect. A synchronous globally parallel GA creates and evaluates population members in parallel at each generation resulting in considerable processor time spent waiting for threads. An asynchronous approach whereby parallel threads continue evolution without waiting addresses this issue but can result in memory conflicts. This paper introduces an asynchronous global GA model for shared memory CPUs without memory conflicts. Experiments demonstrate performance gains of 1.35 to 12 fold dependant on problem and population sizes. However, an asynchronous model leads to non-uniform evolution reducing accuracy. Consequently, this paper demonstrates that combining synchronous and asynchronous methods into a partially asynchronous model retains a speed advantage whilst improving solution accuracy.

CCS CONCEPTS

• **Mathematics of computing → Evolutionary algorithms**.

KEYWORDS

Genetic Algorithm, Parallelisation, Asynchronously

ACM Reference Format:
Darren M. Chitty. 2021. A Partially Asynchronous Global Parallel Genetic Algorithm . In *2021 Genetic and Evolutionary Computation Conference Companion (GECCO '21 Companion), July 10–14, 2021, Lille, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3449726.3463190

1 INTRODUCTION

In recent years microprocessor manufacturers have been unable to keep pace with Moores Law for increasing processor speeds via nano-scale manufacturing. Consequently, manufacturers have migrated to a parallel methodology with multiple processor cores operating in parallel. Central Processing Units (CPUs) can now have up to 64 processor cores and even lightweight processors such as ARM CPUs have four processor cores. Graphics Processing Units (GPUs) have thousands of processor cores enabling significant speedups. It is key for computationally intensive optimisation methods to harness this processing power to maximum effect.

GECCO '21 Companion, July 10–14, 2021, Lille, France

ACM ISBN 978-1-4503-8351-6/21/07.
https://doi.org/10.1145/3449726.3463190

Evolutionary algorithms are a particular optimisation methodology ideally suited to parallel architectures. Genetic Algorithms (GAs) [14], Ant Colony Optimisation (ACO) [8] and Particle Swarm Optimisation (PSO) [9] are all population-based methods and naturally parallel as population members can be generated and evaluated using differing parallel processes. Indeed, the parallel nature of GAs was quickly recognised and exploited but managing parallel implementations can be problematic and as such considerable work has been performed on methods to improve GA parallelism. Key is ensuring that processor *occupancy* remains high, all processor cores are used to maximum ability. A simple parallel GA approach is a generational synchronous model whereby at each generation population members are created and evaluated by parallel threads before the next generation. This can be inefficient if the time to create and evaluate solutions is low or non-uniform. An asynchronous approach avoids this by parallel threads creating and evaluating population members continuously without waiting for other threads but can be unsafe parallelisation. Thus, the popular asynchronous method is to use independent sub-populations on each processor core with occasional migration between sub-populations. However, if a new processor is faster with double the number of cores this speedup will not be reflected by the parallel GA as further sub-populations are necessary. With CPUs now having up to 64 cores a parallel standard global GA may be preferable.

The paper is laid out as follows. Section 2 will discuss prior parallelisation strategies for GAs. Section 3 will demonstrate synchronous parallisation of GAs and an alternative novel methodology for asynchronous parallisation. Both will be compared in terms of speed and accuracy using a permutation type problem in Section 4. To obtain the best from both models Section 5 will introduce a *partially* asynchronous approach. Finally, Section 6 will draw conclusions from the approaches and highlight scope for improvement.

2 BACKGROUND

Evolutionary algorithms (EAs) are typically population-based consisting of a set of chromosomes, a colony of ants or a swarm of insects. Each member represents a potential solution and new solutions in the population can normally be generated and evaluated independently from others. Therefore, in the context of parallel computation, evolutionary algorithms are an ideal technique.

The earliest EA was the GA [14] using the principles of Darwinian evolution to create populations of chromosomes that represent solutions to a given problem. Naturally parallel by being population-based the first parallel implementations of a GA were investigated by Grefenstette [11] who considered four differing designs; the synchronous master-slave model, the semi-synchronous master-slave, the asynchronous concurrent model and the network model. Synchronous refers to parallel threads matching each other such as operating on the same generation. Asynchronous refers

to parallel threads being not required to match to other threads. The synchronous master-slave only evaluates a single generation of solutions in parallel which wastes time by waiting for threads to complete tasks each generation. The semi-synchronous master-slave addresses this issue with the master thread adding new solutions as the slave processes work. With the asynchronous concurrent model each slave thread both creates and evaluates solutions using a shared memory population. The downside is that threads can read and update the same memory location, a *memory conflict*.

The network model is the dominant methodology for parallelisation of GAs with independent GAs operating on separate processors as sub-populations. Their only collaboration is to broadcast their best solutions to other sub-populations. The advantage is full utilisation of a parallel processor, but in the case of many cores, a large overall population. This methodology is known as the Parallel Genetic Algorithm (PGA). Tanese [22] expanded this concept into an *island* approach with each sub-population referred to as a deme or island and periodic *migration* between these islands, a broadcast. A four dimensional hypercube computer was used with migration occurring uniformly in time with neighbouring populations along a given dimension. Two additional parameters are required, the migration interval, the number of generations between migration, and the migration rate, a percentage of sub-population individuals to migrate. Cohoon et. al. [5] used a similar approach with a fully connected sub-population. However, Mühlenbein [18] considered other topologies such as a ring so solutions take time to percolate to all sub-populations. Alternatively, sub-populations can be placed in a 2D plane such that migration can only occur between direct neighbours known as a cellular GA (cGA) [16].

Regarding synchronous or asynchronous implementations, Alba and Troya [1] perform an anaylsis of synchronous migration when sub-populations wait for migrants and asynchronous migration whereby they are inserted on arrival. The authors found an asynchronous approach executed considerably faster due to no time spent waiting thereby maximising processor use. Mühlenbein [17] considered a fully asynchronous approach for a cGA whereby each population individual acts independently and mates with a neighbouring individual on a 2D grid with the result replacing the current solution if an improvement. However, there is no discussion of avoiding threads simultaneously reading and writing to the same memory location. Luque et. al. [15] implemented an asynchronous cGA with each processor evolving a portion of the 2D grid and at each iteration solutions are sent along processor edges to their neighbour processors. This is migration at every generation but enables processors to evolve their assigned population members asynchronously. Performance similar to an island model was reported. Pinel et. al. [19] also implemented an asynchronous parallel cGA and address the issue of two threads simultaneously reading and writing to the same memory location by using thread locking.

The second Grefenstette model, the asynchronous concurrent model for a standard or *global* GA is less popular. However, Rasheed and Davison [21] implemented a version whereby the master thread immediately gives slaves new solutions when they complete evaluations, a form of centralised control. Golub and Budin [10] considered an asynchronous implementation of a global GA by implementing an asynchronous master-slave. An elimination tournament selection was used with three individuals selected and the weakest discarded. The authors note differing threads can perform invalid iterations requiring increased generations. Depolli et. al. [7] consider a queue-based approach for problems with computationally expensive evaluation whereby the master sends solutions to be evaluated and generates new solutions to be evaluated asynchronously. However, the authors note a *selection lag* whereby a good solution could take longer to evaluate than other less fit solutions which then get selected for reproduction. Harada and Takadama [13] consider a *semi-asynchronous* model whereby slaves pause before generating new solutions until a minimum number of current generation solutions are evaluated. Results indicated better performance than synchronous or asynchronous models.

Recently, GPU implementations of parallel GAs have been considered. A fundamental difference from CPUs is that the cores are grouped under a few individual multiprocessors under which the same instruction must be executed simultaneously. Vidal and Alba [23] implemented a cGA on a GPU reporting a speedup of up to 25 fold when using very large population sizes in the order of 25k. The approach is synonymous with a synchronous master-slave model in that the CPU acts as the master and waits at each generation for the GPU to create and evaluate solutions. Chen et. al. [3] implemented a GA using a GPU but required significant synchronisation for crossover resulting in minimal speedup. Pospichal et. al. [20] implemented an island GA on a GPU with asynchronous migration reporting speedups of up to 8000 fold for numerical optimisation.

3 PARALLEL GENETIC ALGORITHM MODELS

There are two methods for implementing a global GA in parallel using a multi-core CPU, synchronously or asynchronously. This section will profile the synchronous approach and present a simple method to achieve *memory conflict* free asynchronous execution.

3.1 A Synchronous Parallel GA Model

A GA is population-based using the Darwinian principles of natural selection, genetic crossover and mutation to create new generations. This enables evolution to occur with fitter solutions for the given problem under consideration being combined to potentially generate better solutions in subsequent generations. Population members are independent from the others and can be created and evaluated in isolation and in parallel. The previous generation is sampled with natural selection to choose parents and offspring for the next generation created using crossover and mutation.

To implement a synchronous parallel GA at each generation a set of parallel threads create and evaluate new population members for the next generation. Synchronisation requires each thread to complete its creation and evaluation of population members before the next generation can start. This is in effect the master-slave model as defined by Grefenstette [11]. Algorithm 1 provides an overview of a synchronous parallel global GA and Algorithm 2 the work that threads perform on their designated population members. With t parallel threads available each creates $\frac{Population\ Size}{t}$ members. The advantage of a synchronous method is identical operation to a sequential version. A disadvantage is considerable creation of parallel threads and time spent waiting for threads to complete. Note the two nested `for` loops on lines 2 and 3 of Algorithm 1 whereby the outer loop needs to wait for the threads generated in the inner loop to complete their assigned tasks.

Algorithm 1 Synchronous Parallel GA

1: Generate initial population and evaluate
2: **for** each generation **do**
3: **for** each parallel thread t **do**
4: Create parallel slave thread with ID t to generate $\frac{Population\ Size}{Thread\ Count}$ solutions
5: **end for**
6: Wait for all threads to finish generating their solutions
7: **end for**

Algorithm 2 Synchronous Parallel GA Slave Thread

1: **for** each population member in set defined by thread ID t **do**
2: Select two parents from whole population
3: Generate two new solutions using parent chromosomes
4: Apply mutation with random probability
5: Evaluate new solutions
6: **end for**

3.2 An Asynchronous Parallel GA Model

To address the issue of repeated thread creation and waiting an alternative approach is to have each thread begin creating population members for the following generation as soon as they finish their current generation. Threads do not wait for others to complete, they operate as an asynchronous concurrent model [11]. However, a problem is that when combining chromosomes with crossover a parent chromosome could be being updated concurrently by another thread especially when chromosomes are long. This can cause incorrect solutions for permutation type problems such as the Travelling Salesman Problem (TSP), Vehicle Routing Problem (VRP) or sorting problems. With a permutation type problem a given value such as a city to visit must occur once only in a solution. If one thread is reading a chromosome whilst another is updating it a value could be read twice and placed in the new solution with another missing entirely, an invalid solution.

PSO can be easily implemented with asynchronous parallelisation as solutions simply *fly* towards others rather than copying chromosomes. ACO is less easy to implement in parallel asynchronously as a pheromone matrix requires updating after each iteration but population-based ACO can operate asynchronously as other solutions merely recommend paths to take [4, 12]. In that respect a GA is unique in directly copying part of other parent solution chromosomes which have the potential to be updated by other threads in an asynchronous implementation, a *memory conflict*.

One option is to detect and ignore invalid solutions generated by these *memory conflicts* but this requires computational time and loss of evolutionary steps by ignoring. Alternatively, a form of thread locking could be used such that only one thread has priority but this is similar to the synchronous approach with parallel threads waiting to be unlocked. A popular method, as discussed in Section 2, to avoid this issue but ensure an asynchronous implementation is an island model with multiple sub-populations on differing parallel threads. A parallel thread only selects individual solutions from its own island avoiding threads reading and writing to the same chromosome simultaneously. However, with this model each thread requires a minimum population size for sensible evolution to occur. For instance, if there are 32 parallel threads or cores and 20 population members then an overall population size of 640 members is required. Moreover, since CPU manufacturers increase speed by increasing processor cores if the next processor has 64 processor cores a population size of 1280 will be necessary and no speed advantage from the new processor will be achieved.

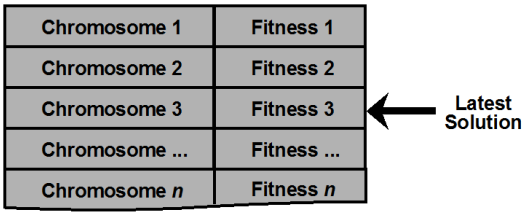

Figure 1: A population member's *working* memory.

Therefore, if a speedup is desired an alternative methodology is required which can operate asynchronously with a fixed size population using CPUs with potentially hundreds of processor cores. As noted, the problem is parallel threads reading and writing from the same chromosome or not being able to sample from the entire population. A solution is to utilise the shared memory available on modern computing systems. A simple method could be to have every single chromosome stored throughout the evolutionary process. Parallel threads then sample from the latest completed chromosomes across the whole population. However, for a chromosome of size 1000, a population size of 100 and evolution of 50,000 generations 18.6GB of memory would be required.

An alterative approach is to provide each population member a small *working* or *short term* memory, it can store the n last solutions and associated fitness. As evolution progresses a population member fills up its *working* memory with solutions. When it becomes full it returns to the beginning overwriting the oldest stored chromosomes. Each population member has a counter indicating which chromosome in the *working* memory is the latest to be completed. Other parallel threads of execution when sampling from the population use the latest chromosomes and fitness to be generated by each population member stored in the memory. Given it could be some time before these chromosomes are overwritten it avoids the problem of memory conflicts. If a *working* memory of 100 historical chromosomes is used then memory consumption for a chromosome of size 1000, a population size of 100 and evolution of 50,000 iterations would now only require 0.04GB memory, a 500 fold reduction. Figure 1 depicts this chromosome *working* memory.

Algorithm 3 provides a high level overview of the asynchronous master thread operation whilst Algorithm 4 demonstrates the simple parallel slave step. Note that Algorithm 3 has only one loop now and the wait for threads to complete outside this loop and the slave thread in Algorithm 4 now has two nested loops. The approach significantly reduces the degree of waiting at every generation used by the synchronous model. To avoid problems with reading and writing to the same chromosome by parallel threads working memory chromosomes of each parent are used as shown on line 4. When solutions are generated they are written into the working memory for each population member and their memory counters incremented as described by line 7.

Algorithm 3 Asynchronous Parallel GA

1: Generate initial population and evaluate
2: Create working memory size m for each population member
3: **for** each parallel thread t **do**
4: Create parallel slave thread with ID t to generate $\frac{Population\ Size}{Thread\ Count}$ solutions for the requisite generations
5: **end for**
6: Wait for threads to complete evolution, output best solution

Algorithm 4 Asynchronous Parallel GA Slave Thread

1: **for** each generation **do**
2: **for** each pop. member in set defined by thread ID t **do**
3: Select two parents from population
4: Generate two new solutions using latest chromosomes from parents working memory
5: Apply mutation with random probability
6: Evaluate new solutions
7: Store in member working memories pointed to by counters and increment
8: **end for**
9: **end for**

4 EXPERIMENTAL RESULTS

To measure the effectiveness of the proposed asynchronous globally parallel GA implementation in contrast to a synchronous model both will be tested using a permutation problem. As previously discussed, when reading a parent chromosome if another thread updates the chromosome at the same time then invalid solutions can be generated. The permutation problem used is the Travelling Salesman Problem (TSP), an NP-Complete combinatorial optimisation problem with the goal to visit all cities once minimising distance travelled. The symmetric TSP is represented as a complete weighted graph $G = (V, E, d)$ where $V = \{1, 2, .., n\}$ is a set of vertices defining each city and $E = \{(i, j)|(i, j) \in V \times V\}$ the edges consisting of the distance d between pairs of cities such that $d_{ij} = d_{ji}$. The objective is to find a Hamiltonian cycle in G of minimal length.

Six TSP instances from the TSPLIB library ranging in size are considered. The crossover operator PMX with be used combining two parent chromosomes to generate two offspring and swap, inversion and insertion mutations will also be used. Experiments are conducted using an AMD Ryzen 2700 processor which has eight physical processor cores and via hyper-threading, an additional eight *virtual* cores. Consequently, the parallel GA implementation will use 16 threads of execution. Results are averaged over 25 random runs with differing seeds for a range of population sizes. The parameters used with the GA approach are shown in Table 1.

A working memory size of 200 solutions is used as experiments found memory conflicts still occurred with less working memory and a population size of 32 for the smallest problem. A processor can *stall* waiting for a memory read of a chromosome from main memory such that the slave *owner* of the chromosome can perform several hundred generations of evolution. Unfortunately, the behaviour of parallel threads is not under direct control. However, it was found that the working memory size is inversely proportional to the ratio of population members to threads and problem size.

Table 1: Genetic Algorithm parameters

Maximum Iterations -	50,000
Mutation Probability -	33%
Elitism Rate -	2 population members
Tournament Size -	10% of population

Table 2 provides the experimental results in terms of relative error to the best known solution, runtime and also the measured processor occupancy. In all instances and population sizes the asynchronous GA model is the faster approach. Speedup varies dependant on both population size and problem size. Hence, for a small population and the smallest problem a 10 fold reduction in execution speed is achieved. However, for the same problem, raising the population size to 256 individuals the speedup reduces to only a two fold gain. The cause is that for the synchronous model the degree of time spent waiting for threads to complete as a proportion of the parallel workload reduces. Therefore, the advantage of the asynchronous model reduces as the population and problem size increase. This hypothesis is borne out by the processor occupancy rates which for the synchronous model increase considerably as population and problem size increase. For the largest problem and 256 individuals the processor occupancy is only slightly less than the asynchronous model resulting in only a 35% speed advantage.

Regarding accuracy, note that in a majority of cases the synchronous model has a small advantage. This is due to the asynchronous model having *non-uniform* or *non-synchronised* evolution. Population members being generated and evaluated on differing processor threads can be several generations apart in evolutionary terms, a *generational gap*. In fact, observations during experimental runs noted this could be hundreds of generations of difference especially for the smaller population sizes. Overall accuracy is poor due to no local search being utilised. This adds a further aspect to comparing synchronous and asynchronous globally parallel GAs. Given the computational cost of local search it is used sparingly and consequently can lead to further unbalanced evolution and, for the synchronous parallel GA model, significantly greater waiting time.

Therefore, the experiments from Table 2 are repeated using a 1% probability of the 2-opt local search operator [6] being used. This operator is computationally expensive as it attempts to exchange every combination of two edges and retain the swap if an improvement. This process repeats until no single improvement can be found. Results are shown in Table 3 whereby it can be observed that the speedups achieved by the asynchronous parallel GA model are larger in this instance compared to the results in Table 2. The cause can be found by analysing processor occupancy. For the asynchronous model the processor occupancies have increased slightly. However, for the synchronous parallel GA model processor occupancies have reduced considerably, especially with larger population sizes, to just 17% usage. The reason is that a single population member can require considerable computational time being improved by 2-opt whilst other threads have finished their assigned tasks but need to wait. This results in significant wasted computer processor time. The asynchronous model retains greater processor occupancy by parallel threads continuing evolution while a population member undergoes 2-opt improvement.

Table 2: Average error, runtime and processor occupancy for synchronous and asynchronous parallel genetic algorithms with varying population sizes applied to a range of TSP instances.

		Synchronous Parallel Genetic Algorithm			Asynchronous Parallel Genetic Algorithm			
TSP Instance	Population Size	Error (%)	Runtime (secs)	CPU Occupancy (%)	Error (%)	Runtime (secs)	CPU Occupancy (%)	Speedup
pr1002	32	28.13 ± 1.52	41.09 ± 0.91	22.13 ± 0.33	31.65 ± 2.75	4.32 ± 0.25	59.61 ± 2.79	9.51x
	64	20.58 ± 1.67	42.41 ± 0.08	25.87 ± 0.30	20.50 ± 1.43	7.25 ± 0.17	62.62 ± 1.47	5.85x
	128	13.83 ± 1.26	44.71 ± 0.12	30.67 ± 0.26	14.77 ± 1.88	13.64 ± 0.23	63.70 ± 0.68	3.28x
	256	10.07 ± 1.10	55.43 ± 0.47	39.81 ± 0.51	10.35 ± 1.06	26.35 ± 0.31	64.87 ± 0.52	2.10x
pcb1173	32	32.36 ± 1.48	43.59 ± 1.18	22.28 ± 0.35	37.87 ± 3.11	4.83 ± 0.19	62.93 ± 2.04	9.02x
	64	25.60 ± 1.41	42.63 ± 0.22	27.75 ± 0.73	25.70 ± 1.65	8.28 ± 0.24	65.38 ± 1.27	5.15x
	128	18.50 ± 1.48	47.14 ± 0.14	32.09 ± 0.32	18.79 ± 1.66	15.40 ± 0.41	66.16 ± 1.10	3.06x
	256	12.71 ± 1.18	55.57 ± 0.30	46.44 ± 0.48	13.79 ± 1.24	29.41 ± 0.28	67.62 ± 0.33	1.89x
fl1400	32	28.88 ± 2.43	42.49 ± 0.51	22.93 ± 0.23	29.49 ± 3.83	5.68 ± 0.34	62.33 ± 2.62	7.48x
	64	23.19 ± 2.08	45.12 ± 0.32	27.32 ± 0.28	23.53 ± 3.38	9.32 ± 0.21	65.76 ± 0.86	4.84x
	128	17.42 ± 2.32	49.56 ± 0.09	34.28 ± 0.34	16.63 ± 3.10	17.37 ± 0.24	67.70 ± 0.78	2.85x
	256	11.89 ± 1.48	56.34 ± 0.29	51.91 ± 1.07	11.75 ± 1.69	33.36 ± 0.37	69.35 ± 0.56	1.69x
u2193	32	44.85 ± 1.47	47.97 ± 0.15	25.60 ± 0.22	47.95 ± 2.11	8.35 ± 0.42	73.06 ± 2.64	5.75x
	64	40.49 ± 1.09	49.05 ± 0.37	32.20 ± 0.26	40.93 ± 1.79	13.33 ± 0.57	73.89 ± 2.07	3.68x
	128	34.91 ± 1.22	56.52 ± 0.77	41.63 ± 0.38	35.26 ± 1.20	24.20 ± 0.39	75.42 ± 0.89	2.33x
	256	27.45 ± 1.13	67.98 ± 0.82	66.06 ± 0.33	28.23 ± 1.01	47.28 ± 2.70	75.25 ± 3.09	1.44x
pr2392	32	42.90 ± 1.31	48.46 ± 0.22	27.20 ± 0.31	44.75 ± 2.23	9.27 ± 0.53	74.17 ± 2.83	5.22x
	64	39.53 ± 1.00	50.45 ± 1.78	32.13 ± 0.54	39.17 ± 1.46	14.37 ± 0.43	76.06 ± 1.55	3.51x
	128	33.33 ± 0.97	57.25 ± 1.30	44.81 ± 0.66	34.55 ± 1.63	27.08 ± 1.28	75.73 ± 2.86	2.11x
	256	26.98 ± 1.17	72.73 ± 0.32	58.96 ± 4.24	27.86 ± 1.43	50.83 ± 2.52	77.13 ± 2.41	1.43x
fl3795	32	56.14 ± 2.39	62.26 ± 1.55	32.21 ± 0.77	56.88 ± 2.24	14.48 ± 0.63	80.01 ± 1.72	4.30x
	64	53.08 ± 1.95	59.51 ± 0.60	41.04 ± 0.42	53.67 ± 3.39	21.51 ± 0.85	80.15 ± 1.25	2.77x
	128	48.73 ± 2.18	67.37 ± 7.55	60.34 ± 2.67	48.36 ± 2.22	38.77 ± 0.85	80.99 ± 0.83	1.74x
	256	42.44 ± 1.48	99.78 ± 0.98	72.21 ± 1.43	43.04 ± 2.50	73.76 ± 2.16	82.22 ± 1.07	1.35x

Table 3: Average error, runtime and processor occupancy for synchronous and asynchronous parallel genetic algorithms with varying population sizes and a 1% probability of using 2-opt applied to a range of TSP instances.

		Synchronous Parallel Genetic Algorithm			Asynchronous Parallel Genetic Algorithm			
TSP Instance	Population Size	Error (%)	Runtime (secs)	CPU Occupancy (%)	Error (%)	Runtime (secs)	CPU Occupancy (%)	Speedup
pr1002	32	3.01 ± 0.28	109.09 ± 1.33	18.26 ± 0.15	4.90 ± 0.68	9.16 ± 0.23	82.03 ± 1.57	11.91x
	64	2.42 ± 0.33	167.65 ± 1.74	17.26 ± 0.20	2.85 ± 0.46	17.63 ± 0.33	84.89 ± 1.33	9.51x
	128	1.83 ± 0.44	284.86 ± 3.22	17.62 ± 0.10	2.02 ± 0.39	34.34 ± 0.45	86.57 ± 0.71	8.29x
	256	1.69 ± 0.42	479.01 ± 4.87	18.51 ± 0.12	1.75 ± 0.41	68.49 ± 0.84	87.02 ± 0.75	6.99x
pcb1173	32	3.78 ± 0.48	127.72 ± 1.57	17.97 ± 0.10	6.25 ± 0.63	11.53 ± 0.29	84.82 ± 1.92	11.07x
	64	2.75 ± 0.54	208.59 ± 5.26	16.79 ± 0.71	3.86 ± 0.52	22.03 ± 0.37	87.29 ± 0.93	9.47x
	128	2.33 ± 0.45	352.44 ± 3.62	17.38 ± 0.15	2.58 ± 0.44	43.07 ± 0.62	88.87 ± 0.79	8.18x
	256	2.13 ± 0.45	598.34 ± 5.77	18.42 ± 0.09	2.13 ± 0.44	85.51 ± 1.17	89.50 ± 0.69	7.00x
fl1400	32	2.29 ± 0.56	143.30 ± 2.08	17.27 ± 0.16	2.95 ± 0.54	13.81 ± 0.32	86.70 ± 1.72	10.38x
	64	1.71 ± 0.24	248.76 ± 3.31	16.72 ± 0.18	1.90 ± 0.41	26.07 ± 2.96	88.96 ± 1.73	9.54x
	128	1.61 ± 0.44	415.71 ± 6.85	16.74 ± 0.12	1.72 ± 0.49	50.67 ± 8.74	90.40 ± 1.36	8.20x
	256	1.36 ± 0.26	729.24 ± 13.69	18.42 ± 0.09	1.41 ± 0.32	105.03 ± 1.96	91.27 ± 0.58	6.94x
u2193	32	6.44 ± 0.60	249.68 ± 3.64	16.97 ± 0.15	9.81 ± 0.74	29.36 ± 0.54	92.44 ± 1.50	8.50x
	64	4.84 ± 0.60	449.98 ± 6.74	16.49 ± 0.13	6.94 ± 0.75	56.57 ± 1.10	93.64 ± 1.60	7.95x
	128	4.04 ± 0.51	823.34 ± 8.37	16.38 ± 0.16	4.55 ± 0.37	114.21 ± 1.81	94.08 ± 0.92	7.21x
	256	3.23 ± 0.32	1482.56 ± 33.67	17.42 ± 0.22	3.55 ± 0.37	223.61 ± 3.68	95.80 ± 0.98	6.63x
pr2392	32	5.62 ± 0.42	306.88 ± 4.30	17.02 ± 0.21	8.55 ± 0.77	34.86 ± 0.92	92.28 ± 1.55	8.80x
	64	4.04 ± 0.39	550.77 ± 5.62	16.85 ± 0.20	6.39 ± 0.73	66.65 ± 1.97	94.09 ± 1.56	8.26x
	128	3.21 ± 0.32	1003.89 ± 14.45	16.88 ± 0.18	3.72 ± 0.41	130.43 ± 2.27	95.24 ± 1.09	7.70x
	256	2.64 ± 0.41	1763.47 ± 16.18	18.20 ± 0.19	2.63 ± 0.29	260.27 ± 2.85	96.23 ± 0.76	6.78x
fl3795	32	2.81 ± 0.70	603.05 ± 14.00	15.76 ± 0.12	4.24 ± 0.86	83.28 ± 1.99	93.72 ± 1.49	7.24x
	64	2.03 ± 0.88	1162.34 ± 19.72	15.09 ± 0.08	3.28 ± 0.97	161.89 ± 4.98	95.37 ± 1.23	7.18x
	128	1.60 ± 0.41	2180.17 ± 48.19	15.39 ± 0.14	1.73 ± 0.71	327.55 ± 8.07	96.52 ± 0.78	6.66x
	256	1.01 ± 0.53	3923.37 ± 98.16	16.64 ± 0.09	1.17 ± 0.31	646.34 ± 12.69	97.78 ± 0.47	6.07x

However, as with the results in Table 2, it should be noted that the synchronous parallel GA model achieves markedly better accuracy than the asynchronous methodology. This is especially noteworthy when using a small population size. Once again. the reason for this is *non-uniform* evolution leading to a generational gap occurring to a greater degree through the use of 2-opt local search. In the time taken to perform 2-opt on a single solution many generations of evolution of other population members can occur.

5 A PARTIALLY ASYNCHRONOUS GA MODEL

Evidence from the results shown in the previous section demonstrated that whilst an asynchronous parallel GA model is 10-12x faster than a synchronous approach the accuracy deteriorates. It is hypothesized that this is due to unbalanced evolution occurring such that a generational gap between slave threads occurs. Indeed, one thread can be hundreds or thousands of evolutionary generations ahead of another thread. Consequently, their solutions can be much more highly evolved and hence dominate the population reducing the diversity and impacting overall solution quality.

Algorithm 5 Partially-Asynchronous Parallel GA

1: Set synchronous generations as $\frac{Total\ Generations}{Asynchronous\ Generations}$
2: **for** each synchronous generation **do**
3: **for** each parallel thread t **do**
4: Create parallel slave thread with ID t
5: Wait for all threads to complete asynchronous evolution of assigned population members
6: **end for**
7: **end for**

Algorithm 6 Partially-Asynchronous Parallel GA Slave Thread

1: **for** each asynchronous generation **do**
2: **for** each population member defined by thread ID t **do**
3: Select two parents from population
4: Generate two new solutions using latest solutions from parents working memory
5: Apply mutation with random probability
6: Evaluate new solutions
7: Store in solution working memories and update counters
8: **end for**
9: **end for**

Given the speed advantage of the asynchronous methodology via better exploitation of processor resources this model could still be the preferred option. Moreover, if the generational gap can be mitigated it could lead to an improved accuracy. A method to achieve this could be to combine both models, in effect a *partially* asynchronous parallel GA. With this approach the main components of a purely asynchronous model are retained with slave threads executing for multiple generations on their given population members. Population members would still require a working memory to avoid memory conflicts when performing crossover from parents selected from the global population. However, these slave threads will only evolve asynchronously for a set number of generations designated *asynchronous generations*. In this way the generational gap can be no greater than the *asynchronous generations*. The master thread waits until each thread has completed its given *asynchronous generations* before creating new parallel slave threads to begin the next set of *asynchronous generations*. This process continues until the full number of standard generations is achieved. This is similar in some respect to the implementation by Bozikovic et.al. [2] although in their case only crossover is performed for multiple iterations by slave threads and mutation by the master thread.

Table 4: Average error and runtimes for the partially asynchronous parallel GA for a population size of 32, 1% probability of 2-opt and a range of asynchronous generations.

TSP Instance	Asynchronous Generations	Relative Error (%)	Runtime (secs)	Speedup
pr1002	5000	5.10 ± 0.64	9.53 ± 0.20	11.45x
	1000	4.71 ± 0.75	10.58 ± 0.25	10.31x
	500	4.69 ± 0.41	11.59 ± 0.21	9.41x
	250	4.04 ± 0.56	13.34 ± 0.26	8.18x
	100	3.26 ± 0.42	17.48 ± 0.25	6.24x
	50	3.03 ± 0.44	23.36 ± 0.37	4.67x
pcb1173	5000	6.68 ± 0.89	11.93 ± 0.32	10.71x
	1000	6.18 ± 0.69	13.45 ± 0.26	9.50x
	500	5.95 ± 0.78	14.82 ± 0.28	8.62x
	250	5.10 ± 0.61	17.17 ± 0.28	7.44x
	100	4.34 ± 0.59	22.71 ± 0.31	5.62x
	50	4.09 ± 0.54	30.43 ± 0.67	4.20x
fl1400	5000	2.81 ± 0.47	14.57 ± 0.35	9.83x
	1000	2.74 ± 0.48	16.50 ± 0.39	8.69x
	500	2.85 ± 0.50	18.61 ± 0.30	7.70x
	250	2.40 ± 0.43	21.66 ± 0.48	6.62x
	100	2.28 ± 0.50	28.67 ± 0.72	5.00x
	50	2.14 ± 0.33	38.37 ± 0.94	3.74x
u2193	5000	9.74 ± 0.73	31.45 ± 0.66	7.94x
	1000	9.59 ± 0.81	36.16 ± 0.78	6.90x
	500	8.87 ± 0.88	40.42 ± 0.84	6.18x
	250	7.91 ± 0.82	47.93 ± 0.82	5.21x
	100	7.15 ± 0.76	64.69 ± 1.32	3.86x
	50	6.62 ± 0.72	86.14 ± 2.01	2.90x
pr2392	5000	8.62 ± 0.73	36.90 ± 0.85	8.32x
	1000	8.05 ± 0.77	42.32 ± 0.98	7.25x
	500	7.38 ± 0.79	47.60 ± 0.86	6.45x
	250	7.28 ± 0.70	55.84 ± 0.76	5.50x
	100	6.41 ± 0.45	75.25 ± 1.44	4.08x
	50	5.76 ± 0.56	99.40 ± 2.06	3.09x
fl3795	5000	4.71 ± 0.79	88.44 ± 2.09	6.82x
	1000	4.49 ± 0.84	102.35 ± 2.18	5.89x
	500	3.66 ± 0.72	114.76 ± 2.13	5.25x
	250	3.49 ± 0.71	133.80 ± 3.65	4.51x
	100	3.07 ± 0.52	179.97 ± 5.78	3.35x
	50	2.95 ± 0.80	237.52 ± 6.41	2.54x

Algorithm 5 provides a high level overview of the master thread. Note that as with the fully synchronous approach there are two nested `for` loops but in this case the outer loop operates for less generations defined as the total generations divided by the number of allowable asynchronous generations. This reduces the degree of time spent waiting for threads to finish their allotted asynchronous generations. Algorithm 6 provides a high level overview of the partially asynchronous slave thread which is almost the same as the fully asynchronous slave aside from the outer loop only iterating over the number of allowable asynchronous generations.

This approach will allow slower threads to *catch up* enabling more uniform evolution to occur and reduce the generational gap. A disadvantage will be that there will be a loss of speed as a result of increased periodic waiting for slower threads to complete thereby reducing processor occupancy. To test the effectiveness of a *partially* asynchronous parallel GA the experiments from previously with 2-opt local search will be repeated for the smallest population size of 32 individuals which had the greatest degree of non-uniform evolution. A range of asynchronous generation sizes will be evaluated. Results are shown in Table 4 whereby it can be observed that as the size of permissible asynchronous generations reduces the runtime increases reducing achievable speedup. Also,

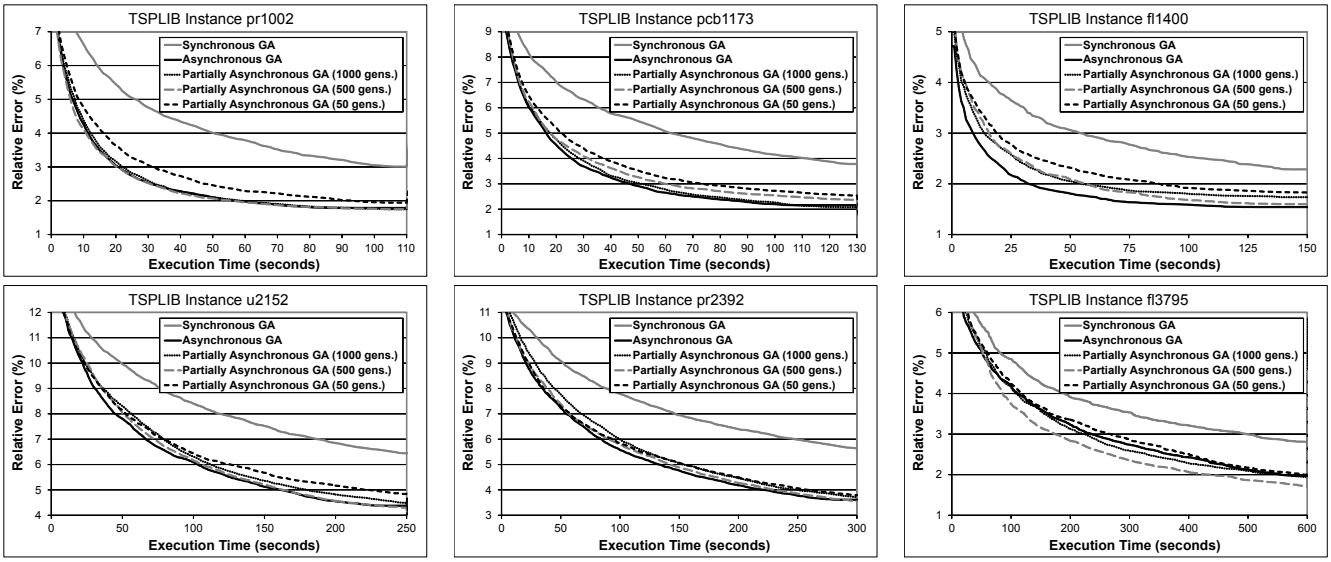

Figure 2: Convergence rates measured over time for synchronous, asynchronous and partially asynchronous parallel GA models applied to range of TSP instances using population size of 32 members.

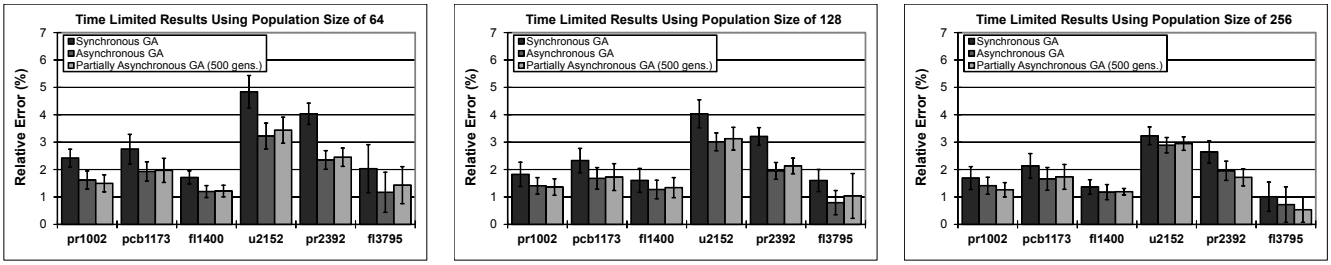

Figure 3: Relative errors for synchronous, asynchronous and partially asynchronous parallel GA models applied to range of TSP instances using population sizes of 64, 128 and 256 individuals.

as the size of permissible asynchronous generations reduces the relative error or accuracy improves. This occurs as parallel threads that get behind on evolutionary generations, a generational gap, get more opportunity to *catch up* leading to more uniform evolution. This reinforces the hypothesis that non-uniform evolution in fully asynchronous evolution reduces solution quality. However, contrasting to the synchronous parallel GA results in Table 3 even using the smallest number of asynchronous generations relative error remains slightly worse.

Given that the partially asynchronous parallel GA remains slightly less accurate even with a just a small number of allowable asynchronous generations it could be considered that speed is a more important component. If using a high number of asynchronous generations results in a runtime advantage of five fold over a synchronous model then in effect five times the overall evolutionary generations can be achieved within the same timescales. This additional evolution will improve the accuracy of the partially asynchronous GA, potentially to a greater level than the synchronous GA. The results in Table 4 show that when the asynchronous generations fall below 250 accuracy only slightly improves but the speed advantage reduces considerably. Therefore, the fully asynchronous and partially asynchronous approaches will be allowed to evolve solutions for the same timescales as the synchronous approach. As with Table 4 the smallest population size of 32 will be tested with the average convergence rates over time shown in Figure 2. These convergence rates show that whilst the fully asynchronous model is less accurate over fixed generations its greater speed enables better results then the fully synchronous approach through greater evolution over fixed timescales. Moreover, the better partially asynchronous approach is to use 500 asynchronous generations which achieves the best results for several of the problems especially the largest instance. When using just 50 asynchronous generations the reduction in speed reduces the attainable evolutionary steps reducing accuracy. The reduction in speed outweighs the improved accuracy achieved over a fixed set of generations.

Figure 3 compares the relative errors of the synchronous, asynchronous and partially asynchronous models when executed for the same degree of time using the larger population sizes. These results demonstrate that regardless of population size the asynchronous and partially asynchronous models achieve equally better performance than the synchronous model due to the extra evolutionary steps obtainable. The asynchronous model is slightly better for population sizes of 64 and 128 but the partially asynchronous approach improves with a population size of 256 individuals.

6 CONCLUSIONS

With parallel microprocessors now commonplace there is a requirement to fully harness the technology with computationally intensive techniques such as evolutionary algorithms. Genetic Algorithms are naturally parallel and hence ideal to exploit parallel processors. However, a parallel synchronous approach can spend too much time waiting for all threads to complete a generation whilst an asynchronous approach can result in memory conflicts when reading and writing chromosomes especially when longer. This paper demonstrated that by giving population members a *working memory* of their latest solutions an asynchronous global GA can avoid memory conflicts. Experiments using a permutation type problem revealed processor occupancy approaches 100% utilisation with speedups of as much as 12 fold over a synchronous model.

However, asynchronous evolution results in *non-uniform* evolution with differing threads many generations of evolution apart, a generational gap. This effect reduces overall solution quality especially when solution generation is unbalanced. To counter this runaway evolution population members need to *catch up*. A mixed synchronous and asynchronous model, a *partially* asynchronous model, can mitigate this generational gap by only allowing a fixed number of asynchronous generations before threads have to wait for others to *catch up*. Over a fixed number of evolutionary generations accuracy is improved but with a speed penalty.

Indeed, experiments over a fixed time frame rather than generations demonstrated that fully asynchronous or coarse grained partially asynchronous models can achieve the best overall solution quality through simply being able to simulate greater evolution by being faster. Consequently, although a generational gap can occur causing non-uniform evolution, solution quality degradation can be overcome given the ability to evolve longer.

It should be noted that the *working* memory incurs a memory cost which can be relatively large. Consider a TSP instance with 200k cities, each population chromosome is 200k in size thus with a population size of 256 and a working memory of just 50 requires 10GB of memory. However, an island model using 32 threads and sub-populations of 20 individuals would require a similar amount of memory. An asynchronous global parallel GA though can operate with a small population in contrast to an island GA model. Therefore, as with any evolutionary algorithm, careful consideration needs to be made as to the choice of population size relative to the number of processor cores, the evolution time, the working memory size and the degree of partial asynchronous generations to reduce the generational gap.

Further work with the partially asynchronous global parallel GA model will consider methods to reduce the working memory size such as a steady state implementation. This would reduce the frequency that population members update their latest chromosome. In addition, greater analysis is required regarding the generational gap from asynchronous evolution in terms of its impact vs. speed and methods to reduce its effect. Furthermore, a direct comparison between an island model and the partially asynchronous GA is required to ascertain which approach is the more advantageous including scalability to fine grained parallel architectures.

REFERENCES

[1] Enrique Alba and José M Troya. 2001. Analyzing synchronous and asynchronous parallel distributed genetic algorithms. *Future Generation Computer Systems* 17, 4 (2001), 451–465.

[2] Marko Bozikovic, Marin Golub, and Leo Budin. 2003. Solving n-Queen problem using global parallel genetic algorithm. In *The IEEE Region 8 EUROCON 2003. Computer as a Tool.*, Vol. 2. IEEE, 104–107.

[3] Su Chen, Spencer Davis, Hai Jiang, and Andy Novobilski. 2011. CUDA-based genetic algorithm on traveling salesman problem. In *Computer and Information Science 2011*. Springer, 241–252.

[4] Darren M Chitty. 2017. Applying ACO to Large Scale TSP Instances. In *UK Workshop on Computational Intelligence*. Springer, 104–118.

[5] James P Cohoon, Shailesh U Hegde, Worthy N Martin, and D Richards. 1987. Punctuated equilibria: a parallel genetic algorithm. In *Genetic algorithms and their applications: proceedings of the second International Conference on Genetic Algorithms: July 28-31, 1987 at the Massachusetts Institute of Technology, Cambridge, MA*. Hillsdale, NJ: L. Erlhaum Associates, 1987.

[6] Georges A Croes. 1958. A method for solving traveling-salesman problems. *Operations research* 6, 6 (1958), 791–812.

[7] Matjaž Depolli, Roman Trobec, and Bogdan Filipič. 2013. Asynchronous master-slave parallelization of differential evolution for multi-objective optimization. *Evolutionary computation* 21, 2 (2013), 261–291.

[8] Marco Dorigo and Luca Maria Gambardella. 1997. Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Transactions on evolutionary computation* 1, 1 (1997), 53–66.

[9] Russell Eberhart and James Kennedy. 1995. Particle swarm optimization. In *Proceedings of the IEEE international conference on neural networks*, Vol. 4. Citeseer, 1942–1948.

[10] Marin Golub and Leo Budin. 2000. An asynchronous model of global parallel genetic algorithms. In *Second ICSC Symposium on Engineering of Intelligent Systems EIS2000, University of Paisley, Scotland, UK*. Citeseer, 353–359.

[11] John J Grefenstette. 1981. Parallel adaptive algorithms for function optimization. *Vanderbilt University, Nashville, TN, Tech. Rep. CS-81-19* (1981).

[12] Michael Guntsch and Martin Middendorf. 2002. A population based approach for ACO. In *Workshops on Applications of Evolutionary Computation*. Springer, 72–81.

[13] Tomohiro Harada and Keiki Takadama. 2020. Analysis of semi-asynchronous multi-objective evolutionary algorithm with different asynchronies. *Soft Computing* 24, 4 (2020), 2917–2939.

[14] John H Holland. 1975. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press.

[15] Gabriel Luque, Enrique Alba, and Bernabé Dorronsoro. 2009. An asynchronous parallel implementation of a cellular genetic algorithm for combinatorial optimization. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*. 1395–1402.

[16] Bernard Manderick and Piet Spiessens. 1989. Fine-grained parallel genetic algorithms. In *Proceedings of the third international conference on Genetic algorithms*. 428–433.

[17] Heinz Mühlenbein. 1989. Parallel genetic algorithms, population genetics and combinatorial optimization. In *Workshop on Parallel Processing: Logic, Organization, and Technology*. Springer, 398–406.

[18] Heinz Mühlenbein. 1991. Evolution in time and space–the parallel genetic algorithm. In *Foundations of genetic algorithms*. Vol. 1. Elsevier, 316–337.

[19] Frédéric Pinel, Bernabé Dorronsoro, and Pascal Bouvry. 2010. A new parallel asynchronous cellular genetic algorithm for scheduling in grids. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*. IEEE, 1–8.

[20] Petr Pospichal, Jiri Jaros, and Josef Schwarz. 2010. Parallel genetic algorithm on the cuda architecture. In *European conference on the applications of evolutionary computation*. Springer, 442–451.

[21] Khaled Rasheed and Brian D Davison. 1999. Effect of global parallelism on the behavior of a steady state genetic algorithm for design optimization. In *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*, Vol. 1. IEEE, 534–541.

[22] Reiko Tanese. 1987. Parallel genetic algorithm for a hypercube. In *Genetic algorithms and their applications: proceedings of the second International Conference on Genetic Algorithms: July 28-31, 1987 at the Massachusetts Institute of Technology, Cambridge, MA*. Hillsdale, NJ: L. Erlhaum Associates, 1987.

[23] Pablo Vidal and Enrique Alba. 2010. Cellular genetic algorithm on graphic processing units. In *Nature Inspired Cooperative Strategies for Optimization (NICSO 2010)*. Springer, 223–232.