6th World Congresses of Structural and Multidisciplinary Optimization
Rio de Janeiro, 30 May - 03 June 2005, Brazil

A Parallel Particle Swarm Optimization Algorithm Accelerated by Asynchronous Evaluations

Gerhard Venter (gventer@vrand.com),
Jaroslaw Sobieszczanski-Sobieski (j.sobieski@larc.nasa.gov)

Vanderplaats Research and Development, Colorado Springs, USA
NASA Langley Research Center, Hampton, USA

1. Abstract
A parallel Particle Swarm Optimization (PSO) algorithm is presented. Particle swarm optimization is a fairly recent addition to the family of non-gradient based, probabilistic search algorithms that is based on a simplified social model and is closely tied to swarming theory. Although PSO algorithms present several attractive properties to the designer, they are plagued by high computational cost as measured by elapsed time. One approach to reduce the elapsed time is to make use of coarse-grained parallelization to evaluate the design points. Previous parallel PSO algorithms were mostly implemented in a synchronous manner, where all design points within a design iteration are evaluated before the next iteration is started. This approach leads to poor parallel speedup in cases where a heterogeneous parallel environment is used and/or where the analysis time depends on the design point being analyzed. This paper introduces an asynchronous parallel PSO algorithm that greatly improves the parallel efficiency. The asynchronous algorithm is benchmarked on a cluster assembled of Apple Macintosh G5 desktop computers, using the multi-disciplinary optimization of a typical transport aircraft wing as an example.

2. Keywords: Particle Swarm Optimization, PSO, asynchronous parallel computing

3. Introduction
Particle Swarm Optimization (PSO) is a fairly recent, but rapidly growing, addition to an expanding collection of non-gradient based, probabilistic search algorithms. Some widely used algorithms that fall into this category are genetic algorithms [1] that model Darwin's principle of survival of the fittest and simulated annealing algorithms [2] that model the equilibrium of large numbers of atoms during an annealing process. This class of optimization algorithms provides the designer with several attractive characteristics. For example, these algorithms are generally easy to implement, can efficiently make use of large numbers of parallel processors, do not require continuity in response functions and are better suited for finding global or near global solutions. Although these non-gradient based algorithms provide the designer with several advantages, they should be applied with care. Due to their high computational cost, these algorithms should only be used when a gradient-based algorithm is not a viable alternative, such as integer/discrete and discontinuous problems.

Many non-gradient based search algorithms are based on some natural phenomena, and PSO is no exception. Particle swarm optimization is based on a simplified social model that is closely tied to swarming theory and was first introduced by Kennedy and Eberhart [3, 4]. A physical analogy might be a school of fish that is adapting to its environment. In this analogy, each fish adapts to its environment by making use of its own memory as well as knowledge gained by the school as a whole.

Although PSO is a fairly recent algorithm, it is quickly gaining momentum in the engineering research community. For example Fourie and Groenwold applied the algorithm to structural shape and sizing [5] and topology optimization [6] problems. The authors applied the algorithm to a cantilevered beam [7] and multi-disciplinary design optimization of a transport aircraft wing [8]. The University of Florida has applied the algorithm to biomechanical system identification problems e.g., [9, 10].

Some authors have worked on parallel PSO algorithms in an attempt to alleviate the associated high computational cost. Most parallel implementations of the PSO algorithm presented to date are based on a synchronous implementation e.g., [9, 10] where all design points within a design iteration are evaluated, before the next design iteration is started. This implementation can easily lead to poor parallel performance in many cases. The present work presents a new asynchronous parallel implementation to increase the parallel performance of the PSO algorithm. The challenge is to keep all processors working as the algorithm moves from one design iteration to the next, without loss of numerical accuracy.

In the present work, both synchronous and asynchronous parallel versions of the PSO algorithm are presented and compared in terms of numerical accuracy and parallel efficiency. The multi-disciplinary design optimization of a typical transport aircraft wing is used as a test problem. This example makes use of a bi-level approach to perform the system level optimization of the wing for maximum range, accounting for the trade-off between the aerodynamic drag and the structural weight and is similar to that presented in [8]. In this formulation, the aerodynamic optimization is performed at the system level, and the structural optimization is considered as a sub-problem at the discipline level. The motivation for using PSO in the present design problem is the presence of discrete design variables and severe numerical noise, which makes the use of a gradient-based optimizer impractical.

4. Particle Swarm Optimization Algorithm

Particle swarm optimization is based on the social behavior that a population of individuals adapts to its environment by returning to promising regions that were previously discovered [11]. This adaptation to the environment is a stochastic process that depends on both the memory of each individual as well as the knowledge gained by the population as a whole. In the numerical implementation of this simplified social model, the population is referred to as a swarm and each individual as a particle.

The numerical implementation starts with an initial swarm of random distributed particles, each with a random initial velocity vector. The algorithm then repeatedly updates the position of each particle over a time period to simulate the adaptation of the swarm to its environment. The position of each particle is updated using the current position, a velocity vector and a time increment. The new position of each particle at iteration $k+1$ is calculated from Eq. (1)

$$\boldsymbol{x}^i_{k+1} = \boldsymbol{x}^i_k + \boldsymbol{v}^i_{k+1}\Delta t \tag{1}$$

where: $\boldsymbol{x}^i_{k+1}$ is the position of particle i at iteration $k+1$; $\boldsymbol{v}^i_{k+1}$ is the corresponding velocity vector; and Δt is the time step value. A unit time step is used throughout the present work.

The velocity vector of each particle can be obtained from many different formulations. A formulation that is widely used in the literature was introduced by Shi and Eberhart[12] and is shown in Eq. (2).

$$\boldsymbol{v}^i_{k+1} = w\boldsymbol{v}^i_k + c_1 r_1 \frac{\left(\boldsymbol{p}^i - \boldsymbol{x}^i_k\right)}{\Delta t} + c_2 r_2 \frac{\left(\boldsymbol{p}^g_k - \boldsymbol{x}^i_k\right)}{\Delta t} \tag{2}$$

In Eq. (2), r_1 and r_2 are random numbers between 0 and 1, $\boldsymbol{p}^i$ is the best position found by particle i so far, and $\boldsymbol{p}^g_k$ is the best position in the swarm during iteration k. There are three problem dependent parameters, the inertia of the particle (w), and two "trust" parameters c_1 and c_2. The inertia controls the exploration properties of the algorithm, with larger values facilitating a more global behavior and smaller values facilitating a more local behavior. The trust parameters indicate how much confidence the particle has in itself (c_1) and how much confidence it has in the swarm (c_2).

A slightly different formulation of the velocity vector [5] is to replace the best position of the current iteration $\boldsymbol{p}^g_k$ with the best position found so far $\boldsymbol{p}^g$. Both formulations will be considered here.

Enhancements to the basic algorithm include dealing with constrained optimization problems, a craziness operator, and dynamically changing the inertia value. These enhancements are discussed in more detail in [7, 8]. Constrained optimization problems are typically handled using a penalty function approach, but this is not used in the present work where an unconstrained optimization problem is considered. The craziness operator adds additional randomness to the swarm to avoid premature convergence and is similar to the mutation operator of a genetic algorithm. A small subset of particles are selected at each design iteration, and the position and/or velocity vectors are randomly modified. As mentioned, the inertia parameter controls the exploration properties of the algorithm. The goal is to start with a larger inertia value (a more global search) that is dynamically reduced towards the end of the optimization (a more local search).

5. Parallel Particle Swarm Optimization Algorithm

The PSO algorithm is ideally suited for a coarse-grained parallel implimentation on a parallel or distributed computing network. For each time step (design iteration), all particles (design points) are independent of each other and can be easily analyzed in parallel.

5.1. Synchronous Parallel Particle Swarm Optimization Algorithm

The most obvious PSO parallel implementation is to simply evaluate the design points within a design iteration in parallel, without changing the overall logic of the algorithm itself. In this implementation, all particles within a design iteration are sent to the parallel computing environment, and the algorithm waits for all the analyses to complete before moving to the next design iteration. This implementation is referred to as a synchronous implementation.

Although the synchronous implementation provides an easy extension to a parallel environment, it is not an ideal solution and generally results in poor parallel efficiency [10]. The problem with this implementation is that it is nearly impossible to keep all processors working towards the end of each design iteration, before the next iteration is started. There are three common situations that would result in some of or most of the processors being idle towards the end of each design iteration, as follows:

1. Having a swarm size that is not an integer multiple of the number of processors
2. A heterogenous distributed compute environment where processors with varying computational speed are combined into a parallel compute environment (e.g., [13])
3. Using a numerical simulation to evaluate each design point, where the required simulation time depends on the design point being analyzed

The influence of having idle processors towards the end of each design iteration on the parallel efficiency is amplified when more processors are used. As a result, one can expect a reduction in the parallel efficiency of a synchronous PSO algorithm when more processors are used.

5.2. Asynchronous Parallel Particle Swarm Optimization Algorithm

The poor parallel efficiency associated with the synchronous parallel PSO algorithm can be overcome by considering an asynchronous algorithm where design points in the next design iteration are analyzed before the current design iteration is completed. The goal is to have no idle processors as one moves from one design iteration to the next.

The key to implementing an asynchronous parallel PSO algorithm is to separate the update actions associated with each point and those associated with the swarm as a whole. These update actions include updating the inertia value, the craziness operator, and the swarm and point histories. For the synchronous algorithm, all the update actions are performed at the end of each design iteration. For the asynchronous algorithm, we want to perform the point update actions after each point is analyzed and the swarm update actions at the end of each design iteration. The parts of the algorithm that need to be considered when looking at the update actions are the velocity vector, the craziness operator and the dynamic reduction of the inertia value.

The velocity vector is the centerpoint of any PSO algorithm. For each design point, the velocity vector is updated using the following dynamic properties for that point: the previous velocity vector; the current position vector; and the best position found so far. In addition, the updated inertia value and the best position for the swarm as a whole are also required. To do the velocity update in an asynchronous fashion, we need to update the position vector and the best position found so far for each design point directly after evaluating that point. For the best position in the swarm, we have two choices: use the best position in the current iteration (p_k^g), or use the best position found so far (p^g). To keep the best position for the swarm current when moving to the next design iteration, before the current iteration is completed, it is necessary to use the best position found so far rather than the best position in the current iteration. This setup allows the algorithm to update all required dynamic properties of the velocity vector directly after evaluating each design point, except for the inertia value. The inertia value is the only iteration level update required to compute the velocity vector and is updated at the end of each design iteration. The craziness operator is the only other iteration level update and is also performed at the end of each design iteration.

The asynchronous algorithm is thus very similar to the synchronous algorithm, except that we update as much information as possible after each design point is analyzed. The inertia and craziness operators are only applied when a design iteration is completed. Of course, this could result in some points of the next design iteration being analyzed before the inertia and craziness operators are applied for that design iteration. However, the influence on the overall performance of the algorithm seems to be negligible.

5.3. Parallel Implementation

In the present work, both synchronous and asynchronous parallel PSO algorithms are implemented and compared. The parallel scheme used here is based on the Message Passing Interface (MPI) to provide a master-slave implementation. In this implementation, one processor is always used as the master processor, and all remaining processors are used as slaves processors. The master processor does not do any computing and is only used to control the communication to and from the slave processors. The master-slave scheme implemented here provides dynamic load balancing between the processors.

The parallel implementation starts with the master processor sending out a design point to each slave processor. Each slave processor receives the request and generates a local temporary working directory. Within this working directory, the analysis is setup and executed for the current design point. Once the analysis is completed, the objective function and constraint values are sent back to the master processor. The master processor checks if any more analyses should be performed and if so, sends design points to the slave processors. This process is repeated until all analyses are completed. The only difference between the synchronous and asynchronous implementations are:

1. The synchronous algorithm performs all design point and swarm updates at the end of each design iteration, while the asynchronous algorithm updates all the design point information as soon as the point is available
2. The synchronous algorithm waits at the end of each design iteration until all points are analyzed before starting the next design iteration, while the asynchronous algorithm directly starts the next design iteration without waiting for all points to be analyzed

5.4. Parallel Hardware

The authors found that the parallel hardware used is a crucial aspect for a successfull parallel implementation of the PSO algorithm. In the present work, each analysis includes a finite element analysis that heavily depends on disk input/output (IO) through scratch files. In this case, it is easy to generate a severe disk IO bottleneck when sharing a common disk among the parallel processors, similar to that reported in [13]. To achieve the best parallel efficiency, it is vital to have a parallel setup where it is possible to access local disk storage for each processor or small subgroup of processors.

In the present work, the Virginia Tech System X computer [14] was used as the parallel compute environment. System X is a collection of 1100 Apple Xserver G5 cluster nodes, where each node consists of two 2.3 GHz PowerPC 970FX processors, 4GB ECC DDR400 (PC3200) random access memory (RAM) and a 80 GB S-ATA hard disk drive.

6. Example Problem

The two parallel PSO algorithms are applied to the multidisciplinary design of a typical long-range transport aircraft wing in the Boeing 767 class, similar to that presented in [8]. For this setup, a single system level analysis consists of a simplified aerodynamic analysis and a structural sub-optimization using the commercially available GENESIS [15] code. The forces from the aerodynamic analysis are transferred to the structural analysis and optimization.

Two independent load cases are considered, a 3.75 G maneuver and a -1.5 G maneuver. The wing is optimized relative to a reference wing with properties summarized in Table 1. Note that the wing area (S), the take off gross weight ($TOGW$), the root chord to tip chord ratio (c_t/c_r) and the sweep angle (p) are all assumed to be constant.

Table 1: Reference wing

Parameter	Value	Parameter	Value
Span (b_{ref})	120 ft	Take off Gross Weight ($TOGW$)	300000 lbs
Root Chord (c_{rref})	0.5 ft	Aspect Ratio (A_{ref})	6.8571
Drag (D_{ref})	4000 lbs	$(h/c)_{ref}$ Ratio	0.12
Range (R_{ref})	5000 $n.\ mi.$	c_t/c_r Ratio	0.4
Area (S)	2100 ft^2	Sweep (p)	25/120

6.1. System Level Optimization

The aerodynamic optimization is performed at the system level. The goal of the system level optimization is to maximize the range of the wing by changing the aspect ratio (A), the depth-to-chord ratio (h/c), the number of internal spars, the number of internal ribs, and the type of wing cover construction. The system level optimization problem is thus an unconstrained problem with five design variables, three of which are discrete variables.

The range is calculated using the simplified Breguet formula that does not account for the required fuel reserve, as shown in Eq. (3)

$$R = C_r \ \frac{L}{D} ln \left(\frac{TOGW}{W_c + W_p + W_{opt}} \right) \tag{3}$$

where: R is the range; L the lift; D the total drag; W_c the non-structural weight; W_p the payload weight; W_{opt} the structural weight and C_r is a constant. The take-off gross weight $TOGW$ includes all the weight components $TOGW = W_c + W_p + W_{opt} + W_f$, where W_f is the fuel weight. In the present work, a structural sub-optimization of the reference wing generates W_{opt} and the range of the reference wing (5000 *n. mi.*) is used to obtain a C_r value of 155.81. The following assumptions are maintained: $W_c = 0.48 \ TOGW$ and $W_p = 0.13 \ TOGW$. Additionally, the lift (L) must equal the $TOGW$, which is considered constant in the present work. With these assumptions, the range formula is simplified as shown in Eq. (4).

$$R = C_r \frac{TOGW}{D} ln \left(\frac{TOGW}{0.61 \ TOGW + W_{opt}} \right) \tag{4}$$

Equation (4) represents the system level objective function for the aerodynamic optimization and consists of two designable components: D and W_{opt}. W_{opt} is the weight from the structural sub-optimization problem factored by 1.3 to account for structural non-optimum weight, and D is the total drag obtained from the aerodynamic analysis. The structural sub-optimization problem and the aerodynamic analysis is discussed in more detail in the following sections.

The finite element model of the wing-box for the reference wing with three spars and nine ribs is shown in Fig. 1. For the present design problem, the front and back spars and the ribs at the root and tip of the wing are always present. However, the internal spar is optional (the number of internal spars is allowed to vary between zero and one) and the number of internal ribs is allowed to vary between zero and seven. The rib spacing (S_{rib}) is a constant and equal to $S_{rib} = b_{ref}/(N_{rib\_max} + 1)$, where $N_{rib\_max} = 7$ is the maximum number of allowable internal ribs. The internal ribs are always placed from the root out, observing the rib spacing S_{rib}, with the number of internal ribs obtained from the corresponding discrete system level design variable. Additionally, the construction of the upper and lower wing covers are selected from either a sandwich or a hat-stiffened construction.

6.2. Structural Sub-Optimization

For the structural sub-optimization problem, a simplified finite element model is used to model the wing-box. The wing-box model consists only of shell elements and, for simplicity, the spar and rib caps are not modeled. It is assumed that the wing-box is manufactured from aluminum with a density of 0.1 lb/in^3, an allowable tensile strength of 50 *ksi* and an allowable compression strength of 25 *ksi*. The reference finite element mesh is shown in Fig. 1 and consists of 72 nodes and 72 shell elements. During the system level optimization, the number of finite elements used to model the wing-box varies between 50 and 72 depending on the number of internal spars and ribs for the current configuration. The nodes shown on the leading and trailing edges of Fig. 1 are non-structural and are included only to transfer aerodynamic loads from the aerodynamic analysis to the structural analysis. The load transfer is done using rigid elements that do not add stiffness to the finite element model.

The goal of the structural sub-optimization problem is to minimize the weight of the wing-box by changing the thickness values of the shell elements, subject to allowable stress and local buckling constraints. The number of design variables in the structural sub-optimization problem depends on the number of internal spars and ribs and the upper and lower wing cover construction selected at the system level. The spars and ribs are modeled as aluminum panels with a single thickness design variable per panel. All three spars between the same two ribs share the same thickness design variable. Similarly, a

single design variable is used to design the thickness of each rib. There is thus a maximum of eight design variables for designing the spar thickness and eight design variables for designing the rib thickness.

The top and bottom wing cover panels are designed on a per panel basis, where each panel is constrained by two spars and two ribs. There is a maximum of 16 panels for the top wing cover and 16 panels for the bottom wing cover of the wing-box. A typical panel with positive force directions, orientation and dimensions is shown in Fig. 2. In the model, each cover panel is represented by a single quadrilateral element if the internal spar and all seven internal ribs are present. If the number of internal spars and/or ribs is reduced, some cover panels are represented by more than one quadrilateral element.

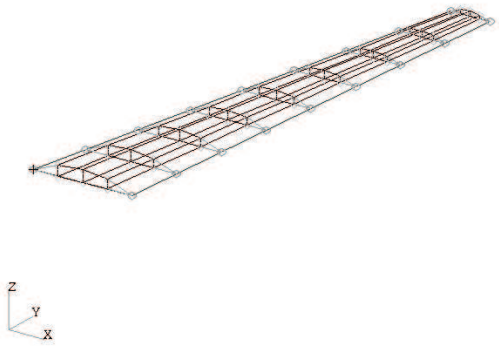

Figure 1: Wing-Box structural finite element model

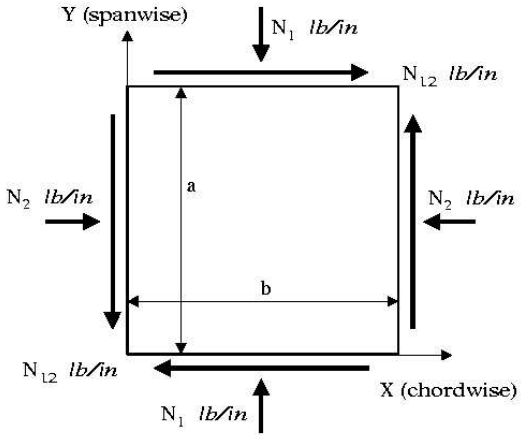

Figure 2: Typical wing cover panel

The number of design variables used to design each panel depends on the selected construction. For the sandwich construction, there are two design variables for each panel as shown in Fig. 3. The design variables are the thickness of the wing cover t and the thickness of the core t_c. There is thus a maximum of 80 structural sub-optimization design variables when considering the sandwich construction, which corresponds to 16 spar and rib thickness design variables and 64 top and bottom wing cover design variables. The core density is assumed to be 1.736 10^{-3} lb/in^3.

For the hat-stiffened construction, there are six design variables for each panel as illustrated in Fig. 4. The design variables are the number of hat stiffeners per panel n, the thickness of the wing cover t_1, the thickness of the stiffener t_2, and the dimensions of the stiffener d_2, d_3, and d_4. The number of stiffeners per panel is currently assumed to be a continuous variable that can be rounded to the next highest number in the final design. There is thus a maximum of 208 structural sub-optimization design variables when considering the hat-stiffened construction, which corresponds to the 16 spar and rib thickness design variables and 192 top and bottom wing cover design variables.

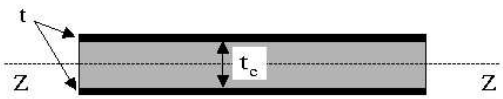

Figure 3: Sandwich construction

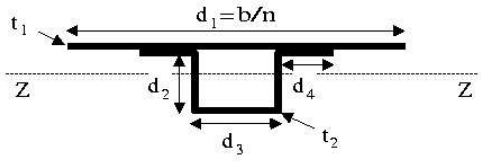

Figure 4: Hat-stiffened construction

The structural sub-optimization problem is subject to both stress and local buckling constraints. Each element is subject to maximum Von Mises stress constraints at both the upper and lower surfaces. The local buckling constraints are applied to each upper and lower wing cover panel, and the equations are summarized in [8].

The aerodynamic loads are applied as nodal forces to the bottom surface nodes only, including the non-structural leading and trailing edge nodes. The normalized span-wise and chord-wise distribution of the assumed forces are shown in Figs. 5 and 6.

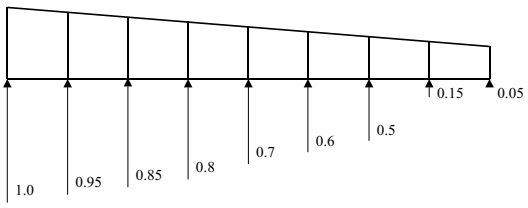

Figure 5: Normalized span-wise pressure distribution

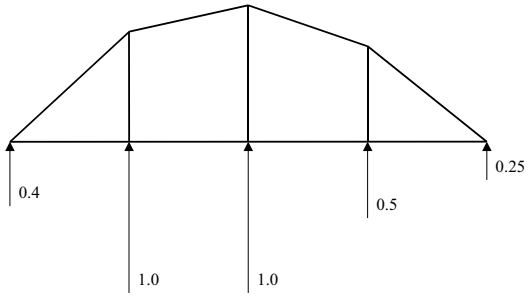

Figure 6: Normalized chord-wise pressure distribution

The aerodynamic pressure distribution is converted to concentrated nodal forces, accounting for the areas associated with each node, using Eq. (5)

$$F_{i,j} = C_f \; LP_j \; CP_i \; S_{i,j} \tag{5}$$

where i is the line number and j the chord number. Additionally, C_f is the wing loading, which is 143 *lb/sf* for the current wing, LP_j is the span-wise pressure distribution value based on the line number from Fig. 6, and CP_i is the chord-wise pressure distribution based on the chord number from Fig. 5. The area associated with each node is represented by $S_{i,j}$. The actual nodal forces are obtained by multiplying the forces obtained from Eq. (5) with a factor equal to the G value of the maneuver. For the 3.75 G maneuver, this factor is 3.75, and for the -1.5 G maneuver, the factor is -1.5. An important note regarding Eq. (5) should be made here. The aerodynamic nodal forces are influenced by the deformation of the wing. In the present work, this interaction is not accounted for. Instead, it is assumed that the wing will be built to a jig-shape that off-sets the deformation due to the aerodynamic loads.

6.3. Aerodynamic Analysis

The aerodynamic analysis is used to calculate the total drag required to calculate the range at the system level, using Eq. (4). A simplified drag calculation is used with the total drag (D) for the current wing calculated based on the total drag of the reference wing (D_{ref}). The calculations consists of the induced drag (D_I), the wave drag (D_w), and a constant fraction of the original drag, as shown in Eq. (6)

$$D = D_I + D_w + C_{dt} D_{ref} \tag{6}$$

where C_{dt} is a constant and is assumed to be 0.4 in the present work. The total drag of the reference wing is $D_{ref} = 40000$ *lb*. The induced drag depends on the aspect ratio of the current wing (A) relative to the aspect ratio of the reference wing (A_{ref}), as follows:

$$D_I = C_{di} D_{ref} \frac{A_{ref}}{A} \tag{7}$$

C_{di} is a constant and is assumed equal to 0.4 in the present work. The wave drag depends on the frontal area of the wing as projected on the streamline direction. The frontal area depends on the span, and the mean height of the wing as follows

$$S_h = b \frac{(h_r + h_t)}{2} \tag{8}$$

where S_h is the frontal area, b is the span and h_r and h_t are the height at the root and the tip of the wing, respectively. The wave drag is then obtained relative to the reference wing using Eq. (9)

$$D_w = C_{dw} D_{ref} \frac{S_h}{S_{href}} \tag{9}$$

where C_{dw} is a constant assumed equal to 0.2 in the present work, S_h is the frontal area of the current wing, and S_{href} is the frontal area of the reference wing.

7. Numerical Results

Both the synchronous and asynchronous PSO implementations were applied to the example problem. The goal was to study two effects: the numerical accuracy of the asynchronous implementation; and the parallel efficiency of both implementations. For each point shown in the following graphs, 10 independent repetitions were executed, and the results averaged over these 10 repetitions. For the numerical accuracy results, the objective functions were averaged for each design iteration. For the parallel efficiency, the total elapsed time was averaged for each number of processors used.

The parameters used for the PSO algorithm are summarized in Table 2. The swarm size of 31 particles was chosen based on the maximum number of processors used in this study. A maximum of 32 processors was used, one as the master processor that does the communication and the 31 remaining processors as slave processors that do all the work.

Table 2: PSO parameters

Parameter	Value	Parameter	Value
Number of particles	31	Trust parameter 1, c_1	1.50
Initial inertia weight, w	1.4	Trust parameter 2, c_2	2.50

7.1. Baseline accuracy study

The synchronous parallel implementation uses exactly the same logic as the serial implementation and as a result gives the same numerical accuracy, independent of the number of processors used. However, the asynchronous algorithm can lead to different numerical accuracy as a result of starting analyses in the next iteration before the current iteration is completed. This effect could be amplified as more processors are used. As a result, the accuracy of the asynchronous algorithm needs to be validated.

To perform this study, we first solved the optimization problem using the synchronous implementation with 16 processors. Four different cases were considered: using both formulations of the velocity vector (using the best point in the current iteration versus using the best point found to date) and applying or not applying the craziness operator. The results are presented in Fig. 7. From Fig. 7, the best synchronous case seems to be using the best point found to date and applying craziness.

For the asynchronous algorithm, we always evaluate the velocity vector using only the best point found so far. As a result, only two different cases were considered: applying or not applying the craziness operator. The results are summarized in Fig. 8. From Fig. 8, it appears that the best performing asynchronous algorithm is when we make use of the craziness operator.

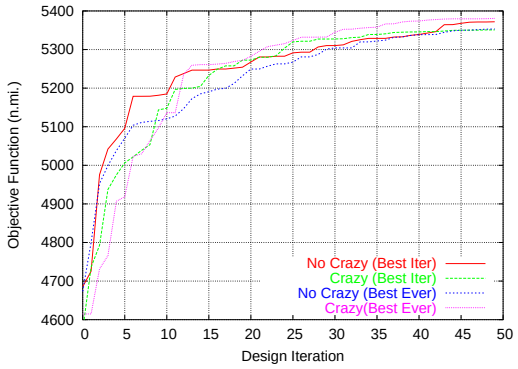

Figure 7: Synchronous baseline study

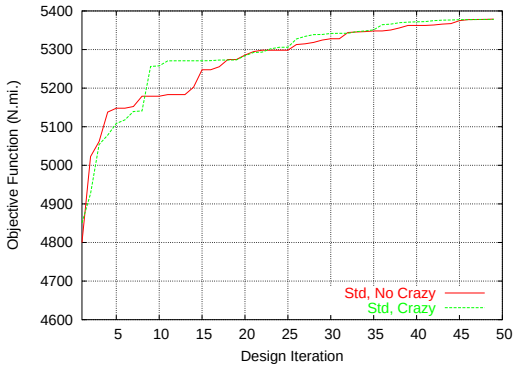

Figure 8: Asynchronous baseline study

From Figs. 7 and 8, it is clear that the numerical accuracy of the two algorithms are comparable. However, the influence of the number of processors on the numerical accuracy of the asynchronous

algorithm should also be studied. For example, for one processor the results should be the same as those of the synchronous algorithm. However, if the number of processors are equal to the number of particles in the swarm, the second iteration will be started after having analyzed only one data point in the first iteration. To check the numerical accuracy of the asynchronous algorithm as a function of the number of processors, the results for 4 (bottom curve), 8, 16 and 32 processors are compared as shown in Fig. 9. Figure 9 also includes the results for the best performing synchronous case from Fig. 7. From Figs. 7, 8 and 9, we can conclude that the numerical accuracy of the PSO algorithm is not compromised when using the asynchronous implementation, even if the number of processors used is equal to the number of particles in the swarm.

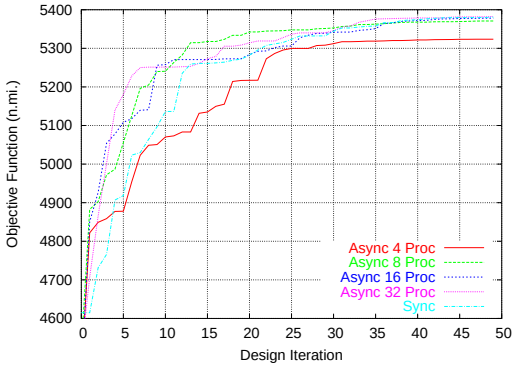

Figure 9: Asynchronous accuracy study

7.2. Parallel efficiency study

For the synchronous implementation, the parallel efficiency is highly problem dependent. For example, the analysis time for the current problem depends on the point being analyzed. However, the overall analysis time is fairly small so that the variation in analysis time does not result in having idle processors for a long time at the end of each design iteration. In contrast, if the analysis time was much larger with a big variation in analysis time, one could end up with a large number of processors being idle for a long time at the end of each design iteration.

To test the parallel efficiency and speedup of the two algorithms, the best performing synchronous and asynchronous algorithms were executed with 1, 4, 8, 16 and 32 processors. For each case, 10 repetitions were completed and the elapsed times averaged. The resulting parallel speedup and efficiency plots are shown in Figs. 10 and 11, respectively. In each case, the ideal speedup and efficiency are also shown.

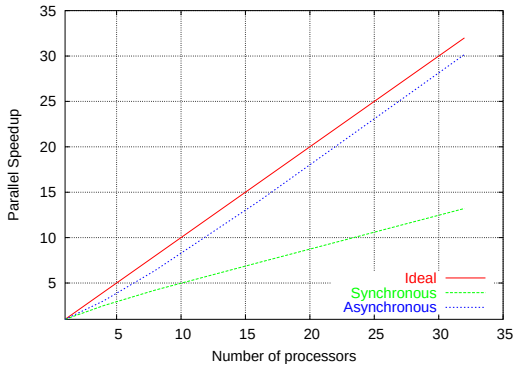

Figure 10: Parallel speedup comparison

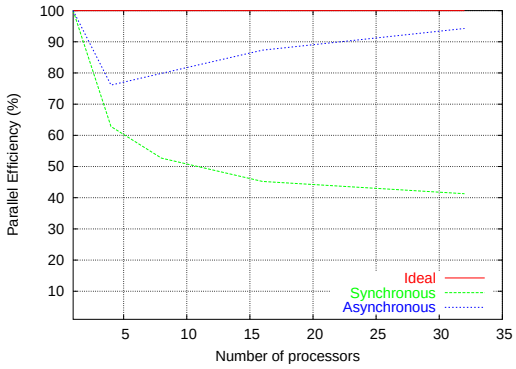

Figure 11: Parallel efficiency comparison

From Figs. 10 and 11, it is clear that the asynchronous algorithm significantly outperforms the synchronous algorithm. As expected, the synchronous algorithm becomes less efficient when more processors

are added. The asynchronous algorithm provides close to the ideal speedup and actually becomes more efficient as more processors are used. This increase in efficiency is explained by the master-slave implementation used. The master processor only communicates with the slave processors that do all the calculations. For small numbers of processors, the master processor is not fully utilized, and a larger fraction of the total number of processors are not working at full capacity. As the number of processors is increased, a larger fraction of the total number of processors is used to do calculations, and thus the increase in efficiency. Although the master processor is under utilized for smaller numbers of processors, it becomes very important to avoid a communications bottleneck for larger numbers of processors. For very large numbers of processors, one may even want to have multiple master processors to avoid possible communication bottlenecks between the master and slave processors.

8. Concluding Remarks

An asynchronous parallel PSO algorithm is presented and is compared to a synchronous parallel PSO algorithm. The numerical results presented indicate that the asynchronous algorithm significantly outperforms the synchronous algorithm in terms of parallel efficiency. In addition, the numerical accuracy of the asynchronous algorithm is comparable to that of the synchronous algorithm.

The present work indicates that PSO is a good candidate for efficient parallel computations when using the asynchronous algorithm. However, the parallel hardware should be appropriate for the problem being solved. For example in the problem presented here, it was crucial to have local disk storage available at the processor level to avoid a disk input/output (IO) bottleneck. The implementation presented here should scale very well to larger numbers of processors and will be the subject of future work.

9. References

[**1**] Michalewicz, Z. and Dasgupta, D., *Evolutionary Algorithms in Engineering Applications*, Springer Verlag, 1997.

[**2**] Nemhauser, G. L. and Wolsey, L. A., *Integer and Combinatorial Optimization, Chapter 3*, John Wiley & Sons, 1988.

[**3**] Kennedy, J. and Eberhart, R. C., Particle Swarm Optimization, *Proceedings of the 1995 IEEE International Conference on Neural Networks (held in Perth, Australia)*, pp. 1942–1948, 1995.

[**4**] Eberhart, R. C. and Kennedy, J., A New Optimizer Using Particles Swarm Theory, *Sixth International Symposium on Micro Machine and Human Science (held in Nagoya, Japan)*, pp. 39–43, 1995.

[**5**] Fourie, P. C. and Groenwold, A. A., The Particle Swarm Optimization Algorithm in Size and Shape Optimization, *Structural and Multidisciplinary Optimization*, 23, pp. 259–267, 2002.

[**6**] Fourie, P. C. and Groenwold, A. A., Particle Swarms in Topology Optimization, *In Proceedings of the Fourth World Congress of Structural and Multidisciplinary Optimization*, Dalian, China, 2001.

[**7**] Venter, G. and Sobieszczanski-Sobieski, J., Particle Swarm Optimization, *AIAA Journal*, 41(8), pp. 1583–1589, 2003.

[**8**] Venter, G. and Sobieszczanski-Sobieski, J., Multidisciplinary Optimization of a Transport Aircraft Wing using Particle Swarm Optimization, *Structural and Multidisciplinary Optimization*, 26(1-2), pp. 121–131, 2004.

[**9**] Schutte, J. F., Fregly, B. J., Haftka, R. T., and George, A., A Parallel Particle Swarm Algorithm. *In Proceedings of the Fifth World Congress of Structural and Multidisciplinary Optimization*, Venice, Italy, 2003.

[**10**] Schutte, J. F., Reinbolt, J. A., Fregly, B. J., Haftka, R. T., and George, A. D., Parallel Global Optimization with the Particle Swarm Algorithm, *International Journal of Numerical Methods in Engineering (accepted)*, 2003.

[**11**] Kennedy, J. and Spears, W. M., Matching Algorithms to Problems: An Experimental Test of the Particle Swarm and Some Genetic Algorithms on the Multimodal Problem Generator, *Proceedings of the 1998 IEEE International Conference on Evolutionary Computation (held in Anchorage, AK)*, 1998.

[**12**] Shi, Y.H and Eberhart, R. C., A Modified Particle Swarm Optimizer, *Proceedings of the 1998 IEEE International Conference on Evolutionary Computation (held in Anchorage, AK)*, 1998.

[**13**] Venter, G. and Watson, B., Efficient Optimization Algorithms for Parallel Applications, *Proceedings of the 8th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, AIAA-2000-4819, Long Beach, CA, 2000.

[**14**] *http://www.tcf.vt.edu/systemX.html*

[**15**] *GENESIS Version 7.0 Users Manual*, Vanderplaats Research and Development, Inc., 1767 S. 8th St., Colorado Springs, CO, 2001.