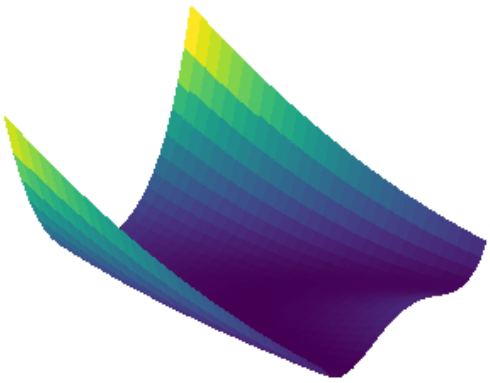
Parallel Particle Swarm Optimisation For Finding Efficient Paths: An Application



Authored by Sean Murray.
Supervised by Dr. Michael Peardon.
Presented for the degree of
M.Sc. in High Performance Computing.

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Department of Mathematics,
Faculty of Engineering, Mathematics, and Science.
Trinity College, The University of Dublin.
2021.

Contents

List of Tables

List of Figures

Declaration

I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at http://www.tcd.ie/calendar.

I have completed the Online Tutorial in avoiding plagiarism 'Ready, Steady, Write', located at http://tcd-ie.libguides.com/plagiarism/ready-steady-write

Sean Murray

Student number: 13319815

XX XX 2021

Acknowledgments

1 Abstract

This project used a Particle Swarm Optimisation (PSO) approach developed in C programming language to solve a path planning problem and comparing different approaches in parallel. A multiagent swarm that is initialised to find the most efficient pathway to a particular minimum or fitness point. This was first tested in serial using benchmark functions with many differently defined local minima and maxima, before being tested using a path planning application that sought to find a destination point for a hypothetical unmanned aerial vehicle. The algorithm itself was subsequently implemented in parallel using a mix of OpenMP and MPI. Each swarm was divided among multiple processors with the aim of solving the same problems posed by the different benchmark functions and the path planning problem. The results highlighted the benefits of using a parallel approach to this particular algorithm. All code, datasets, write-ups, source copies and associated materials relating to the project can be found at https://github.com/seancmry/msc_pro.

2 Introduction

Since its inception in the 1990s, Particle Swarm Optimisation (PSO) has gained credence as an effective solution to large-scale and computationally expensive problems across many different fields. Particle Swarm Optimisation was first described by Kennedy and Eberhart in 1995, who devised an algorithm that relates closely to other types of evolutionary algorithms [9]. Their model was the first to take social behaviour into account and model it based on the principle of "swarming", that is, a population of agents working to devise and actualise a preference for a preferred outcome. The uniqueness of preferences is key to the functioning of this algorithm as they allow the population to change their behaviour based on a number of quality criteria.

Efficient path planning is just one of the many problems that can be addressed through solutions offered by evolutionary algorithms in general and PSO algorithms in particular pathplan. One of the many ways it can be applied in parallel is within the context of a UAV, at a given altitude, searching for an optimal route to its destination.

It is precisely this application that this project looks at. It uses C code to implement a particle swarm approach to solve for a series of devised paths in order to find the global optima, by first generating a series of local optima which are then evaluated in accordance with a particular fitness.

The project looks at the efficient implementation of different parallel algorithms using a mix of OpenMP and MPI, specifically evoluationary-derived algorithms, and the potential speedups that could be observed. Many solutions to this problem have been devised, and the project will look at how the use of PSO in parallel contrast with other variations of the algorithm. It can be shown that the use of parallel strategies to implement these solutions can lead to better efficiency in terms of their iterations from different waypoints in terms of terrain navigation algorithms too. In particular, the algorithm examined will focus on path selection with the use of waypoints within a defined space. When the space in which vehicles have to navigate is large, and thus the number of waypoints increases and becomes large, algorithm efficiency breaks down, and this is known as the path-planning problem.

In terms of the methodology, the initial implementations that will be examined are standard PSO implementations. These implementations will be examined in parallel through a parallel PSO (PPSO) that will be developed in MPI. With the application of parallel algorithms using MPI architectures, the number of waypoints can be reduced with greater efficiency and the most economic route selected depending on different cost calculations. This computing process can be further parallelised as the number of constraints on navigation increases. These constraints will be

tested against controlled variables.

Other algorithms used for the purposes of navigation such as Dijkstra's algorithm could also provide greater efficiencies and speed-ups in light of the quantification of shorter paths to waypoints based on the assumption of greedy techniques, but this discussion is beyond the scope of this project [17]. For now this project aims to look at the application of PSO in terms of UAV path planning problem i.e. strategies used by UAVs. Aspects of the PSO algorithm itself that will be looked at will include the rate of convergence, the growth of elitism among the generated 'particles', and possible hybridization techniques which will form a part of an investigation into possible future research.

Section 1 of this report details how each individual particle is subject to different swarm rules that will dictate the possible approaches each can take. Section 2 details the specifics of the code itself. Section 3 looks at how the testing regime was established and conducted. Section 4 details the results of the investigation and looks at the direction of possible future research in this area.

2.1 Motivation and Research Questions

In terms of the practical applications of PSO algorithms, and due to the fact that it takes much of its dynamic inspiration from natural processes such as flocking or swarming among different animal populations such as within schools of fish, they are numerous. There have been a few prominent examples within the literature of the principles found within processes such as these as they apply to the use of TRN as demonstrated in different settings including by the NASA Perseverance Rover to avoid obstacles during its decent and landing on the Martian surface [4]. Likewise, the example used for this project utilises the concept of UAV navigation to avoid obstacles whilst navigating at a given altitude. This prompts the following research questions to be answered over the course of this investigation:

1. Can the parallelisation of the PSO algorithm used to find a given path produce sizeable and meaningful speedups?

2. Can the PSO algorithm be improved to produce more accurate estimates of fitness?

3. Do these fitness estimates converge quicker with parallelisation?

4. Does this parallelisation represent an improved topological setup?

Theory

3 Particle Swarm Optimization

3.1 General Specification

For a typical PSO implementation you will need:

- Search strategies
- Rules of "swarming" (e.g. move 10 km per day)
- Geography e.g. 2d/3d space plus valleys and mountains
- Treasure or end point that they all must retrieve
- Other variables i.e. random components which can vary travel distance, denoted by r_n

In order to calculate the search strategy you will need a number of important components including personal and team best solutions. The search strategy can be specified in the following manner:

$$\underbrace{\overrightarrow{V_i^{d+1}}}_{\text{Next velocity (tomorrow)}} = \underbrace{w}_{\text{inertia weight}} \underbrace{\overrightarrow{V_i}}_{\text{Current velocity (today)}}$$

$$\ldots + c_1 r_1 \left(\overbrace{\underbrace{\overrightarrow{P_i^d}}_{\text{Personal best solution}} - \overrightarrow{X_i^d}}^{\text{distance to personal best}} \right) + c_2 r_2 \left(\overbrace{\underbrace{\overrightarrow{G^d}}_{\text{Global best solution}} - \overrightarrow{X_i^d}}^{\text{distance to global best}} \right)$$

In addition to the velocity vector you will also have a position vector which is given as

$$\underbrace{X_i^{d+1}}_{\text{Position in day d+1}} = \underbrace{X_i^d}_{\text{Position in day d}} + \underbrace{V_i^{d+1}}_{\text{Velocity in day d+1}}$$

This variant of PSO is commonly referred to as, quite simply, standard PSO, and is given by the following equations for the updates of particle velocity and position. Respectively, X_i and V_i are the position and the velocity of a particular particle, i. After a certain number of iterations, X will denote the best possible position found by the particle, i. The rest of the swarm, the additional particles generated, will arrive at the coordinate of the best aggregate position and the superscript d is the counter of flights (iterations) of the algorithm. $c1$ and $c2$ are the cognitive and social parameters and w is the inertia weight, while $r1$ and $r2$ are two numbers drawn randomly within a given range.

3.1.1 Pseudo-code

Algorithm 1: Pseudocode of PSO algorithm

```
Initialize controlling parameters( N, c1, c2, Wmin, Wmax, Vmax,
and MaxIter);
Initialize the population of N particles;
while end of condition is not satisfied do
    for each particle do
        calculate the objective of the particle;
        update Pbest if required;
        update Gbest if required;
    end
    update the inertia weight;
    for each particle do
        update velocity (v);
        update position (x);
    end
end
return Pbest as the best estimate of the global optimum;
```

In the algorithm implemented, the specification for velocity and position are interlinked as shown. The default values of the user-defined cognition and social parameters c1 and c2 are 1.5 (see code).

3.1.2 Inertia Weight

The inertia weight was proposed to be updated in a decreasing manner according to

$$w = w_0 + (w_f - w_0)\frac{N_v}{N_{max} + N_v}$$

This is the same as in Moraes et al. (The general specification and some of the mathematical information come from Moraes et al [13], including the stochastic properties of the algorithm. Refer to Omkar, Venkatesh, and Mudigere for more information [15].) The code provides for a significant degree of user-directed functionality, and so the initial and final weighting parameters can be specified according to the desired specifications.

In order for the solution to converge to a local optima, all the particles in the swarm must reach to the global optima. However, due to the large computational cost involved, we cannot afford to wait that all particles reach it. Therefore, the basic idea behind the stop criterion is the definition of a suitable average position of the whole swarm. Due to the possibility of existence of two or more global optima, that is, optima with the same value for the best value of the optimal solution but at different x

positions, the swarm position must include a separate parameter to denote this coordinate, f. This is given as

$$\tilde{f} = \frac{f_i - min_j(f_j^0)}{max_j(f_j^0) - min_j(f_j^0)}$$

This specification normalises the minimum and maximum values of the objective function among all particles in the swarm, including all of the particles in the initial population of particles.

3.2 Potential drawbacks

The primary issues with PSO algorithms are (1) getting trapped in local optima (2) performance deterioration with increased problem size. One of the objectives of this project is to address the secondary problem head on, but to also highlight some of the issues brought on by the former problem by examining the performance variances as the problem size is increased to a scale that can only be efficiently computed in parallel. As with many different problems that necessitate the use of multi-core optimization, the problem dimension will inevitably affect the performance of the algorithm at larger and larger sizes [1]. A typical implementation would see the execution time increase in a certain proportion to the problem size, N^2. The testing done by Altinoz et al. demonstrated the solution quality-execution time trade-off when it came to how the performance was enhanced using a different PSO variation.

The problem is commonly referred to as premature convergence i.e. some points attaining fitness quicker than others. This arises from too strong a selective procedure towards a best solution. Too much exploitation of existing building blocks from current population (e.g. by recombining them, or mutating them only slightly in the case of genetic algorithms).

3.3 Evolutionary Context

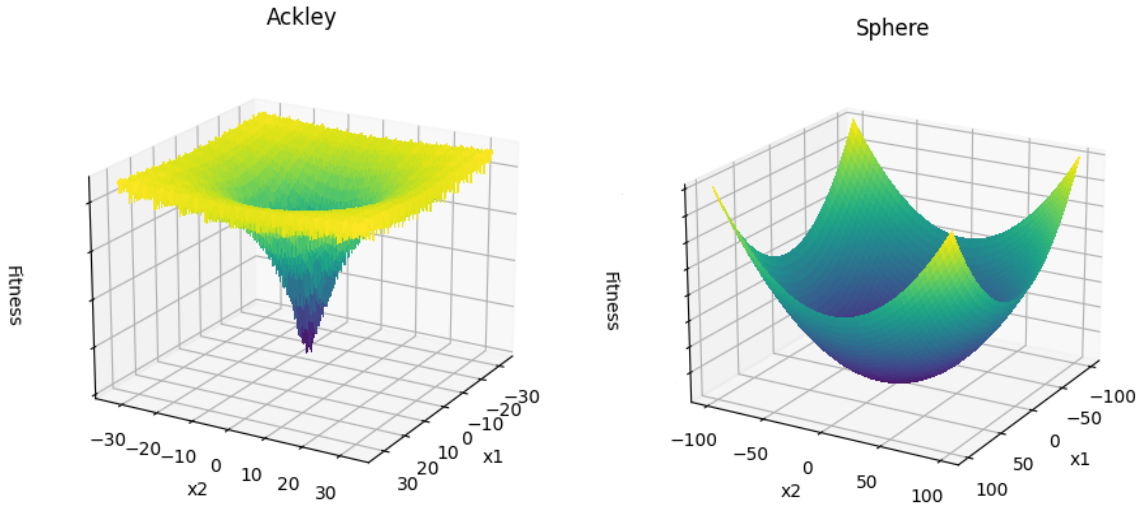
In terms of the topological make-up of the algorithm's application, if this were a master-slave model, the master node would maintain a population and applies genetic operators to the individuals of the population. The master node would then distribute the individuals of a population to slave nodes for fitness evaluations. Since this does away with the genetic approach to evolutionary algorithms, generating a bunch of candidate solutions and check them against an objective function is not required. Then generate subsequent points from the initial generation and see how they evolve.

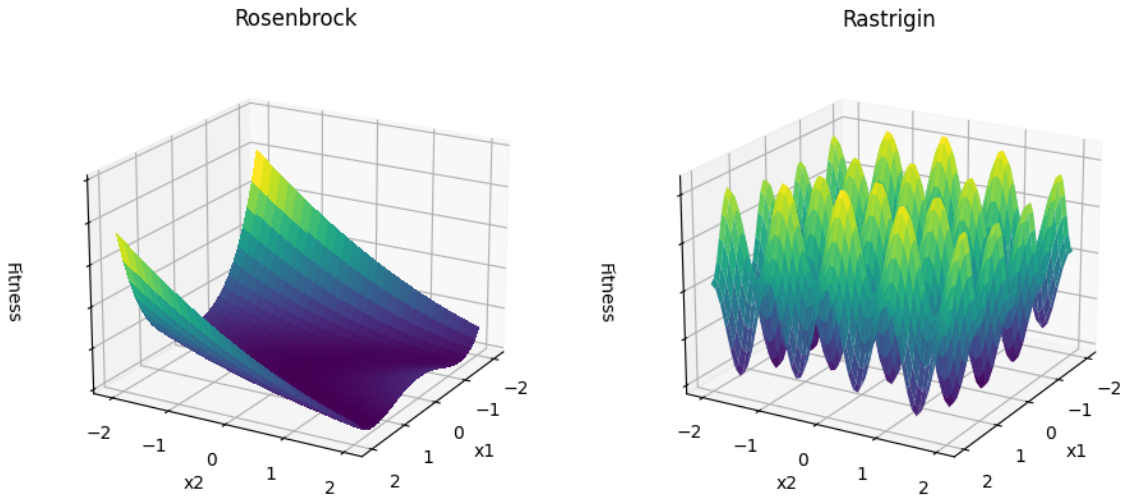
3.4 Premature Convergence

Problems of elitism and premature convergence still need to be discussed, though it will have to be looked at to see if they apply more so in the case of genetic algorithms. If these are not problems that can be picked up on in testing then pick something else to discuss - change the title in that case. Fitness test: It should be noted that individuals with a higher fitness level have a greater probability of reproducing in the case of genetic algorithms [8]. Fitness functions are similar to the ones in the TSP case. Might also be worth looking into what algorithms work with multiple populations. How are the individual components set up and what is required for them to pass certain tests?

3.5 Objective functions

Objective functions [20].





3.5.1 Ackley

$$-20exp\left(-0.2\sqrt{\left(\frac{1}{n}\sum_{i=1}^{n}x_i^2\right)}\right)$$
$$-exp\left(\frac{1}{n}\sum_{i=1}^{n}\cos 2\pi x_i\right)+20+e$$

3.5.2 Sphere

$$\sum_{i=1}^{n}x^2$$

3.5.3 Rosenbrock

$$\sum_{i=1}^{n}\left[100\left(x_{i+1}-x_i^2\right)^2+\left(x_i-1\right)^2\right]$$

3.5.4 Griewank

$$\frac{1}{4000}\sum_{i=1}^{n}x_i^2-\prod_{i=1}^{n}cos\left(\frac{x_i}{\sqrt{i}}\right)+1$$

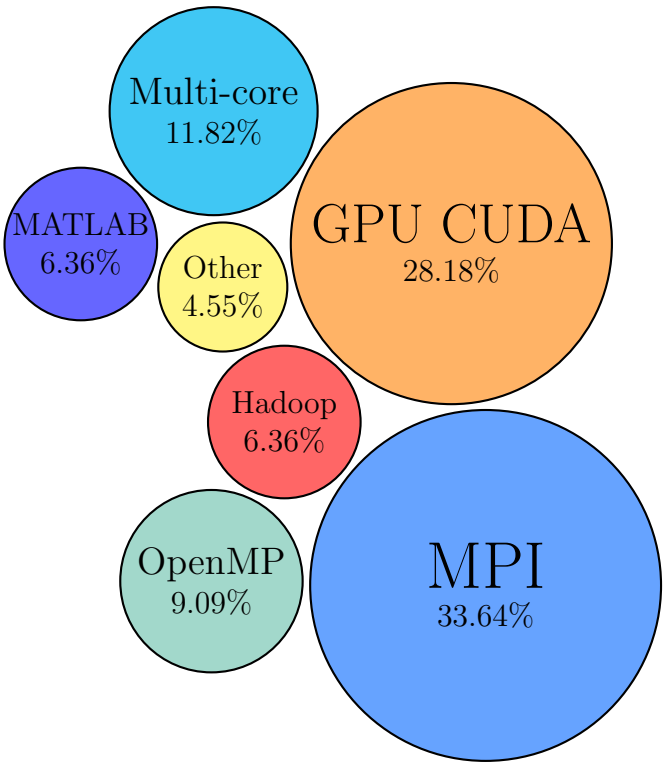


Figure 1: Parallelization strategy-based publication analysis on Parallel PSO [11]

4 Literature Review

MPI is the most popular method in use across a wide variety of applications that have sought to incorporate parallel methods [11]. There is a wealth of information that currently exists on the types of implementations that can be conducted in order to improve the performance of PSO algorithms and techniques. Will have to search through the articles again and compare several things, including the parallelisation strategies employed and problem definitions.

There is already a multitudinous amount of research and tests conducted using different variations of the PSO algorithm, each applied in different contexts such as robotics, geology, and nuclear physics to name but a few applications [11] [12].

4.1 MPI

The use of MPI along with GPU-based solutions has also been considered in some instances, such as the comparison drawn with greedy modular

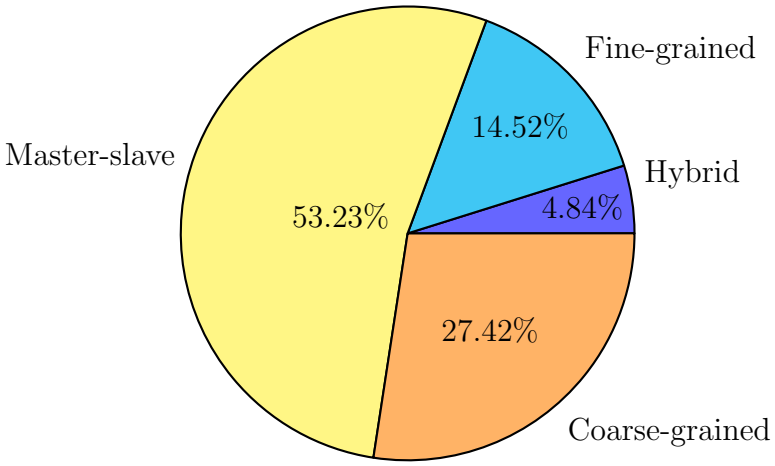


Figure 2: Communication model-based publication analysis on Parallel PSO [11]

eigenspaces, a method used in the selection of hyperspectral images [3]. Similarly, coarse-grained asynchronous solutions have also been looked at with regard to improving the time taken between analysis within the points looked at [23].

Nedjah et al. compare the implementation of PSO between MPI, OpenMP, and CUDA architectures for multi-core and many-core optimisation [14]. A predictable speedup was achieved, though the implementations were primitive in terms of their design, akin to a typical master-slave implementation.

Compared to most implementations, a hybrid design is not as common a feature as one many think [18]. Sengupta, Basak and Peters note that whilst some of these implementations have taken hold in recent years, crossovers such as a joint GA-PSO implementation many not offer as much in terms of simplicity; nonetheless, the literature in this area is growing and more research is likely needed.

3d discussion, see [19] and [22].

Code Design

5 Methodology

5.1 Demo overview

This version demonstrates the efficiency of the PSO algorithm in the context of the 4 different benchmark functions. The Sphere function is the default function. The program is adapted to iterate through each function for a given number of steps at a given ω - inertia weight. The original version of this program stops the algorithm once a desired error is reached, and the legacy code from this is given as a comment in the pso.c file.

The program will look at how long it takes to get from each step to the next with each inertia weight.

5.2 Code

5.2.1 PSO outline

5.2.2 Data structures - settings function

5.3 Inform Functions

The general inform function is a bit confusing at first appearances. It is a struct which connects to other functions. Declared in a general manner using void (*inform_fun)(), it connects with the other inform functions depending on the strategy used; either 'inform_global', 'inform_ring', or 'inform_random'.

5.4 Swarming elements

Bridging the gap between the different disciplines of robotics and computational analysis proves a challenge with terms such as path planning and efficient placement. In computational analysis, these terms do not exist – they are instead defined through abstract variations. In this instance, we take the path planning aspect to be somewhat codified through the travelling salesman problem. Discuss swarming with UAVs [5].

5.4.1 Objective functions

The other functions aside from the main function are benchmark functions, in which the PSO is employed as the preferred search strategy for optimization (sphere, Rosenbrock, Griewank). These functions are used to test the performance of the algorithm.

The main itself contains the basic functions of the algorithm. The parsing of arguments via the command line allows for the use of the benchmark functions, with preassigned values for each contained within the arguments

parsed. This could be altered so that these values can be seeded in a randomised fashion. You type the name of the benchmark function you want to use following './demopso'.

Contained within the settings struct are general PSO settings including the precision goal, the swarm size, the neighbourhood strategy, the neighbourhood size, and the weighting strategy – which assigns the weighting.

You then perform the running of the algorithms and free memory.

5.5 Note on Objective Functions

The demo code for the objective functions now runs up to the maximum number of specified iterations, without the need to stop at the desired error level. The tests for the for the individual functions were conducted on the basis of the original code, so the results are only presented for 25 runs of each function (do this during testing).

Pso.c/.h:

- RNGs are defined first and are used for the purpose of selecting topologies, informing particles as to their different positions. This topology is updated after every two iterations, and the number of informers for each particle is adjusted. *RNG_UNIFORM* is used to generate a random distribution for each particle in each dimension. It is also used in the calculation of the stochastic coefficients.

- In pso.h, the maximum size a swarm can be is defined along with the default value of the weighting, ω.

- The settings for the function are contained in the struct within pso.h.

- The inertia weight update strategies are next in .c, whereby the weight (ω) determining the balance between global and local search is updated in accordance with the boundaries set for the minimum and maximum weights. These are linearly decreasing.

- Matrix strategies: these update the global best, universally the most desirable number for each particle to gravitate towards, and copies it to the next best position.

- There is a general helper function beneath this which performs a similar task, but recurrently updates and informs the different particles as to the location of the global best.

- The different communication topologies are outlined next and include the ring topology, global topology and random topology. Each topology function also contains an informer function in order to inform each as to the directionality of the particles in the swarm.

- Next is the creation of the pso settings, which take the values and definitions given in the .h file. Might be good to list these and explain each in the context of the project report.
- Finally, the algorithm and the different case switches are created. This is the longest function; it is quite self-contained and requires some editing.

Read through current articles in the "Grade V" folder. In terms of defining the problem to be solved by parallelising the code – you can always just parallelise something for the sake of parallelising something – there needs to be a specific definition within the confines of path planning and the use of PSO in the context of path planning. Parallelising C code with MPI will be the solution, but to what problem remains to be defined and we're running out of time in that sense.

- In terms of setting up the geography of the problem, this work takes inspiration from the parameters outlined in Roberge, Tarbouchi, Labonté which compares a genetic algorithm and PSO algorithm in MATLAB – it does not explicitly use those parameters.
- Main results discussion should look at the issue of convergence, how and when the algorithm convergence occurs and the problems thrown up, if any i.e. differences in compilers, etc.
- For the MPI implementation a Cartesian 2d grid is likely the best way to go. Could also go with a graph type grid of some kind.
- Explain really nitty gritty details, such as "I added timing functions to the serial code using xyz".
- Can incorporate 3d elements using 2d set-up [19].
- Work on making obstacles dynamic in the path setup.
- Working on the altitude setup will obviously be tricky, and there is a risk of descending into metaheuristic approaches where Cartesian MPI setups are considered.
- The answer is to keep it in 2d format, but adding some form of tag to each of the different chunks/mesh to indicate the altitude.
- Look at contig1.c on chuck (in the derived folder).
- Side note: When it comes to explaining the reasons for hybrid programming, list all of them. Also, when checking on examples, see what the slides say about them.

5.6 Message Passing Interface(MPI)

Message Passing Interface (MPI)...explain

MPI_Gather: MPI_Gather does the opposite of MPI_Scatter. This routine takes data from different processes and gathers them to one single process, typically a root process. A program that requires a high degree of sorting and searching can benefit from this communicator.

MPI_Bcast: MPI_Bcast broadcasts a message from the root process to all other processes. If you look at the composition of the MPI_Bcast function, you'll note that there is a specification for the root process in the function definition, which means that it is possible to specify whomever you wish to do a send. All processes call the broadcast, including the receives, which don't post a receive.

5.7 General issues

Have a look at the BST implementation for organising function definitions and other operations. Use equations when explaining the different makeup of the topologies used. Some of the organisation of the .h files has involved a bit of work. There is currently a lot of crossover between the functions used by the path file and the pso file, all of which are called by the main. However, some of the functions called by one require variables that are defined in the other, which is leading to confusion in the programme. At first I thought that the Makefile might be the issue but that is not the case.

5.8 Bugs

In terms of designing the obstacles, a circular obstacle provides a greater deal of utility versus a square shaped obstacle, and leads to more interesting results when it comes to moves calculated thereafter. Will require external ints for the utils.h file for the parse arguments to work. Detailed list of bugs so far: Look at what has been declared as "extern" and what the implications are. Other definitions need to be looked at. Implicit declarations. "uav" pointers and initializations. "pso_params" v "pso_result". Request for members in something that aren't structs or unions. "printEnv". "pso_path". "pso_set_default_settings". "countObstructions". Invalid argument types. "pso_settings_t".

5.9 From "Outstanding Issues"

Outstanding issues with the implementation:

- Timing will need to be set up correctly, and work with both the serial and parallel versions.

- The primary difference between the code with the path addition to the algorithm and the basic algorithm set-up is in the use of the settings parameter, which is stored as a singular struct in the basic implementation, but branches out to include parse-friendly values via command arguments in the implementation with the path included. Bridging the gap and seeing which one would be worthy of this is key to the set-up.

- This also feeds into the problem of where to include the parse arguments function. I think it would be best to store this elsewhere.

- Outputting the map generated and the different random generation of obstacles and so on is important for the sake of the analysis.

- Each obstacle could be coded in a way that corresponds to the altitude it is at i.e. tagged so as to indicate the severity of its position. In this sense, if an obstacle is coded with a 1, the UAV would work to avoid it with less effort than an object coded as 4, which could be thought of as a more severe obstacle than one coded as 1 or 2, and thus take more severe evasive action as a result.

- The main function for the path implementation is included in the path.c file itself, and so it will have to be extracted and put somewhere else i.e. its own main.c

- Will have to add an extra 2 or 3 benchmark functions to the pso.c folder.

- These benchmark functions could likewise be implemented in parallel, and parallel communication strategies devised to see how they would be run for larger N^2 problems.

- New main now installed in main.c. Will have to delete old main and work from there to resolve other conflicts when compiling the code.

5.10 OpenMP

OpenMP discussion should be moved to parallelism section. Note that an OpenMP without MPI version should be made and discussed. Omp: explain why you went with the particular implementation you went with, and how else you could have sectioned up the code with OMP. Outline steps taken to decide on hybrid model. Give reasons for the hybrid programming approach as well as advantages of it.

5.11 Memory usage

Memory usage: see latest example in Examples folder

5.12 Benchmark functions

Table and discussion of benchmark functions.

5.13 Topologies

Discussion of topologies. Code segments could be very useful here.

5.14 Modules

Modules loaded included `module load cports` and `module load gsl/2.2.1-gnu` for the random number generation when testing on Lonsdale.

5.15 Path Overview

5.16 ω ranges

6 Parallelism

6.1 Hardware

Break the hardware specs down by author in a table; have a separate table then for the authors and their problem application.

6.1.1 Lonsdale and Kelvin systems

The tests for the both the demo and serial analysis were run on the TCHPC computer cluster systems 'Lonsdale' and 'Kelvin'. Following upgrades in March 2021 Lonsdale was taken out of service for a prolonged period, and so all testing and analysis on parallel code was conducted on Kelvin. The figures for each system's hardware can be viewed in Appendix C.

6.2 Theory

• For the sake of simplicity, the parallelisation of the code will primarily consist of the introduction of MPI and OpenMP components to the PSO algorithm only. The parallelisation of the path planning algorithm and associated functions is the beyond the scope of this project. Discussion about thread safety might be warranted.

Thread safety [7].
One-sided communication [21].

6.3 Code

As well as the timing structure and various other instrumentation aspects, a number of other factors were changed in the context of the solving code itself.

6.3.1 `pso_calc_swarm_size`

For the purposes of testing a large enough swarm across multiples processes, modifications were made to the sizing variable that altered how the swarm size is calculated. `settings->size` is assigned a value in `main.c` but it is overwritten by the `pso_calc_swarm_size` function in the settings. The primary function of this function is to generate a size that fits with the maximum possible size for the swarm giving the value for `settings->dim`. The integer for sizing was adjusted for the number of processes and is multiplied by N to give

```
1
2
3 int size = (100. 20. * sqrt(dim))* nproc;
```

Listing 1: Calculation for swarm size

6.3.2 Removal of `inform` function and `pos_nb`

The primary change that took place during the simple MPI and MPI-OpenMP hybrid runs was the removal of the matrix that informs particles at the local level. This was done for the scalability and for allowing the processes themselves to act as the containers for the particles and how they are informed in future. During this testing phase, the informer functions were not needed and will be replaced by the Cartesian topology and communicator so that communication between particles can take place.

6.3.3 Velocity update

As part of the removal of the `pos_nb` matrix that informed particles locally, an update was required for the calculation of velocity during the particle update loop, which consisted of replacing the next best matrix value with the `solution->gbest` value, which then meant that velocity was calculated as

```
1
2 vel[i][d] = w * vel[i][d] +   \
3       rho1 * (pos_b[i][d] - pos[i][d]) +    \
4       rho2 * (solution->gbest[i] - pos[i][d]);
```

Listing 2: New velocity uodate calculation

Given that the iteration only took place once, the solution result would be arrived at upon the termination of the loop, so this update would not have taken place and was merely for the purpose of executing a clean coded example.

6.4 Modules and evaluation criteria

The table below details the different modules loaded in the cluster system in order to perform the runs.*[0]

Table 1: Loaded modules for testing

Module	Version
`gcc`	`9.3.0`
`gsl`	`2.5`
`openmpi`	`3.1.6`

- **Success Rate (SR)**: This represents the total number of runs that successfully ran through each iteration and completed, expressed as % of the total.
- **Time**: Average time per run in seconds.
- `nproc`: Number of processes selected.

[0]The updated version of `gsl` that was used required an additional `CFLAGS` and `LDFLAGS` linkage in order to work. These can be viewed in the Makefile in the Github repository.

6.5 Preliminary analysis

The preliminary analysis was conducted on both the MPI and hybrid MPI-OpenMP implementations. The analysis showed several issues to be rectified. The primary issue was that suspension of the informer setup and associated functions meant that the program did not iterate or converge to a solution through examining the surround particle points and values, and thus this wasn't a proper particle swarm in the truest sense of the term. The analysis did however highlight the benefits and disadvantages of the hybrid implementation over the standard MPI one, including the more consistent success rate at a lower number of processes.

Table 2: Results for runs conducted through the use an MPI implementation for $N = 1000$, size = 2629 for `nproc = 4`; size = 5792 for `nproc = 9`.

	MPI					
`nproc`	4			9		
	Fitness	**SR**	**Time (sec)**		**SR**	**Time (sec)**
Ackley	2.08E+01	96%	11.056	2.07E+01	88%	12.344
Sphere	2.08E+06	72%	8.181	1.95E+06	84%	9.375
Rosenbrock	2.66E+05	92%	11.399	2.70E+05	48%	12.676
Griewank	1.90E+04	84%	15.016	1.94E+04	56%	16.459

Table 3: Results for runs conducted through the use of a hybrid MPI-OpenMP implementation for $N = 1000$, size = 2629 for `nproc = 4`; size = 5792 for `nproc = 9`.

	MPI-OpenMP					
`nproc`	4			9		
	Fitness	**SR**	**Time (sec)**		**SR**	**Time (sec)**
Ackley	2.07E+01	92%	11.089	2.07E+01	72%	12.332
Sphere	2.07E+06	96%	8.186	2.04E+06	68%	9.411
Rosenbrock	2.85E+05	96%	11.392	2.60E+05	84%	12.684
Griewank	1.89E+04	96%	15.026	1.90E+04	52%	16.441

6.6 Preliminary results

The resulting errors and timing after 1 iteration on the same non-improving particle swarm were broadly consistent across both implementations and displayed no remarkable differences from each other. The orders of magnitude of the errors achieved were the same.

6.7 SR

The key difference that this analysis produced was the different success rates and variation between which implementation performed better at the lower number of processes, `nproc = 4`. The influence of the OpenMP code is clearly shown by splitting the work between threads at the lower level. However, the same cannot be said for a higher number of processes, where the resulting SR is actually lower for the Ackley and Sphere functions, but considerably improved for the Rosenbrock function and marginally better for the Griewank function. This is puzzling when it is also considered that the Ackley function saw a decreased success rate at the lower level of processes for the hybrid implementation.

6.7.1 Computational Imbalances

The results are indicative of computational imbalances at the higher number of processes for the hybrid implementation. What this suggests is that the benefits of this implementation will produce diminishing marginal returns in terms of speedup achieved at the higher number of processes. In this case, the solution would be to change size of work packages for each process to even out work load across threads, and so the different OpenMP functions will require some work.

6.7.2 Serial Optimisation

In terms of the serial optimisation and what should be done next, the solution is clear: it is now necessary to examine a time intensive application and see of code modifications can be made accordingly.

6.8 Function test results

Table 4: Results for runs conducted through the use of an MPI implementation for $N = 100$, for `nproc = 4`.

	MPI			
`nproc`	4			
Function	**Ackley**	**Sphere**	**Rosenbrock**	**Griewank**
Best fitness	1.242E+01	0.000E+00*	8.548E+01	0.000E+00*
Mean fitness	1.565E+01	4.958E+03	6.039E+02	4.335E+01
Poorest fitness	2.124E+01	3.378E+05	4.624E+04	2.833E+03
Standard deviation	1.403E+00	5.663E+03	1.939E+02	3.270E+01
Mean time (MS)	13803.53	6395.39	8619.48	13562.12

Table 5: Results for runs conducted through the use of an MPI implementation for $N = 100$, for `nproc = 9`.

	MPI			
`nproc`	9			
Function	**Ackley**	**Sphere**	**Rosenbrock**	**Griewank**
Best fitness	1.893E+01	1.403E+04	2.152E+03	1.805E+02
Mean fitness	1.955E+01	5.393E+04	5.411E+03	5.072E+02
Poorest fitness	2.127E+01	3.282E+05	4.998E+04	3.049E+03
Standard deviation	2.701E-01	1.428E+04	1.768E+03	1.349E+02
Mean time (MS)	8019.66	5975.03	7361.27	8628.64

Table 6: Results for runs conducted through the use of an MPI implementation for $N = 500$, size = 56, for `nproc = 4`.

	MPI			
`nproc`	4			
Function	**Ackley**	**Sphere**	**Rosenbrock**	**Griewank**
Best fitness	1.833E+01	1.785E+01	1.182E+03	1.687E+00
Mean fitness	1.895E+01	3.300E+04	5.403E+03	3.212E+02
Poorest fitness	2.123E+01	1.615E+06	2.323E+05	1.446E+04
Standard deviation	2.462E-01	6.592E+03	1.018E+03	5.713E+01
Mean time (MS)	91401.02	41231.41	64238.89	75963.03

Table 7: Results for runs conducted through the use of an MPI implementation for $N = 500$, size = 56, for `nproc = 9`.

	MPI			
`nproc`	9			
Function	**Ackley**	**Sphere**	**Rosenbrock**	**Griewank**
Best fitness	1.957E+01	2.938E+05	2.378E+04	2.460E+03
Mean fitness	1.981E+01	4.060E+05	3.822E+04	3.529E+03
Poorest fitness	2.126E+01	1.658E+06	2.439E+05	1.505E+04
Standard deviation	1.319E-01	3.254E+04	5.300E+03	4.265E+02
Mean time (MS)	48848.89	32588.58	43355.32	53043.62

Table 8: Results for runs conducted through the use of an MPI implementation for $N = 1000$, size = 72, for `nproc = 4`.

	MPI			
`nproc`	4			
Function	**Ackley**	**Sphere**	**Rosenbrock**	**Griewank**
Best fitness	1.861E+01	2.046E+03	3.311E+03	2.052E+01
Mean fitness	1.919E+01	9.767E+04	1.413E+04	8.935E+02
Poorest fitness	2.124E+01	3.245E+06	4.720E+05	2.920E+04
Standard deviation	1.592E-01	1.476E+04	3.028E+03	2.050E+02
Mean time (MS)	234544.60	104869.08	162449.60	196292.12

Table 9: Results for runs conducted through the use of an MPI implementation for $N = 1000$, size = 72, for `nproc = 9`.

	MPI			
`nproc`	9			
Function	**Ackley**	**Sphere**	**Rosenbrock**	**Griewank**
Best fitness	1.948E+01	5.452E+05	4.698E+04	4.750E+03
Mean fitness	1.986E+01	7.808E+05	7.649E+04	7.089E+03
Poorest fitness	2.126E+01	3.312E+06	4.898E+05	2.973E+04
Standard deviation	2.065E-01	7.481E+04	9.504E+03	6.683E+02
Mean time (MS)	118191.71	76700.00	104025.10	129684.29

Table 10: Results for runs conducted through the use of an MPI-OpenMP implementation for $N = 100$, for `nproc = 4`.

	MPI-OpenMP			
`nproc`	4			
Function	**Ackley**	**Sphere**	**Rosenbrock**	**Griewank**
Best fitness	1.334E+01	0.000E+00*	7.126E+01	0.000E+00*
Mean fitness	1.587E+01	3.488E+03	5.478E+02	4.499E+01
Poorest fitness	2.122E+01	3.207E+05	4.675E+04	2.890E+03
Standard deviation	1.153E+00	2.981E+03	8.146E+01	4.455E+01
Mean time (MS)	14912.90	8472.07	12471.08	15208.16

Table 11: Results for runs conducted through the use of an MPI-OpenMP implementation for $N = 100$, for `nproc = 9`.

	MPI-OpenMP			
`nproc`	9			
Function	**Ackley**	**Sphere**	**Rosenbrock**	**Griewank**
Best fitness	1.870E+01	2.739E+04	1.446E+03	3.029E+02
Mean fitness	1.953E+01	5.857E+04	4.942E+03	5.176E+02
Poorest fitness	2.127E+01	3.578E+05	5.107E+04	3.244E+03
Standard deviation	2.454E-01	1.650E+04	1.718E+03	1.265E+02
Mean time (MS)	7937.64	7351.56	7361.14	8607.37

Table 12: Results for runs conducted through the use of an MPI-OpenMP implementation for $N = 500$, for `nproc = 4`.

	MPI-OpenMP			
`nproc`	4			
Function	**Ackley**	**Sphere**	**Rosenbrock**	**Griewank**
Best fitness	1.852E+01	7.564E+00	9.929E+02	8.307E-01
Mean fitness	1.900E+01	3.403E+04	5.436E+03	2.923E+02
Poorest fitness	2.124E+01	1.635E+06	2.333E+05	1.470E+04
Standard deviation	1.791E-01	8.238E+03	1.155E+03	6.585E+01
Mean time (MS)	89967.31	41105.31	64335.55	75975.94

Table 13: Results for runs conducted through the use of an MPI-OpenMP implementation for $N = 500$, for `nproc = 9`.

	MPI-OpenMP			
`nproc`	9			
Function	**Ackley**	**Sphere**	**Rosenbrock**	**Griewank**
Best fitness	1.956E+01	2.733E+05	2.128E+04	2.670E+03
Mean fitness	1.988E+01	4.142E+05	3.825E+04	3.639E+03
Poorest fitness	2.127E+01	1.697E+06	2.456E+05	1.478E+04
Standard deviation	1.505E-01	4.171E+04	6.002E+03	3.598E+02
Mean time (MS)	49021.54	32470.61	43278.35	52867.81

Table 14: Results for runs conducted through the use of an MPI-OpenMP implementation for $N = 1000$, for `nproc = 4`.

	MPI-OpenMP			
`nproc`	4			
Function	**Ackley**	**Sphere**	**Rosenbrock**	**Griewank**
Best fitness	1.862E+01	1.587E+03	3.116E+03	2.130E+01
Mean fitness	1.920E+01	9.454E+04	1.413E+04	8.854E+02
Poorest fitness	2.123E+01	3.247E+06	4.680E+05	2.938E+04
Standard deviation	2.056E-01	1.319E+04	1.723E+03	1.390E+02
Mean time (MS)	235159.84	104655.68	162806.80	196361.55

Table 15: Results for runs conducted through the use of an MPI-OpenMP implementation for $N = 1000$, for `nproc = 9`.

	MPI-OpenMP			
`nproc`	9			
Function	**Ackley**	**Sphere**	**Rosenbrock**	**Griewank**
Best fitness	1.950E+01	5.530E+05	4.639E+04	5.635E+03
Mean fitness	1.983E+01	7.642E+05	7.297E+04	7.400E+03
Poorest fitness	2.125E+01	3.348E+06	4.803E+05	2.999E+04
Standard deviation	1.247E-01	6.058E+04	8.656E+03	5.003E+02
Mean time (MS)	119049.22	76010.51	104320.48	131163.41

6.9 Scaled improvement

For the purposes of calculating the speedup for each of the different functions and later on for the application, we use the standard formula for calculating improvements in latency scaled for the number of nodes known as Gustafson's law:

$$Speedup \leq p + (1 - p)s$$

As with the related Amdahl's Law, p is the number of processes and s is the serial execution time. This being true, a programme that runs on 4 cores should perform a speedup factor of 4x of the same programme running on 1 core and so on. The numerical and graphical analysis point to this law being true for only one of the functions, namely the Sphere function, for the purely MPI version of the code for 4 processes. Though superscaling is always possible - where the speedup exceeds the number of processors in use - this does not appear to have occurred here. Speedup graphs show the improvements for 1000 dimensions, the largest size tested, for 4 and 9 processes respectively.

Implementation

7 Testing

7.1 Demo Analysis

Table 16: Parameter settings

Parameter	ω_{min}	ω_{max}	$c_1 = c_2$	steps	`clamp_pos`	`nhood_size`
Value	0.7	0.3	1.496	100,000	periodic	5

The present value for ω_{min} is set as `PSO_INERTIA` or 0.7. The strategy implemented is `PSO_W_LIN_DEC`, which decreases the inertia weight. The settings control the degree of descent. `nhood_size` denotes the number of informers for each particle, which in this case is 5.

The demo was primarily analysed to examine the degree of premature convergence present in the algorithm itself. The tests were informed by the work on previous examinations of these specific benchmark functions [2]. The program was executed for the purpose of testing using the `test.sh` bash file. Ten runs of each algorithm were performed for data collection. The `output.dat` files obtained were converted to `*.csv` for analysis.

7.1.1 Search ranges

Table 17: Search ranges for each function

Function	Range
Ackley	$-32.8, 32.8$
Sphere	$-100, 100$
Rosenbrock	$-2.048, 2.048$
Griewank	$-600, 600$

The Ackley function has a global minimum of $f = 0$ where $x = (0, 0, ..., 0)$ though it has many minor local minima.

7.1.2 Analysis

The first element will be to check the error achieved against the number of iterations of the algorithm required to achieve the desired margin of error. The second element of this analysis is to check the desired fitness level against the different levels actually achieved across different dimensions, `dim`. The mean fitness is the figure to look for. Note, mean time is quoted out of 25 runs. The standard deviation is the standard deviation between the mean values for each run, as explained below.

- **Best fitness:** This is the best fitness solution achieved by each benchmark function as represented by the minimum error level across all runs.
- **Mean fitness:** This is the average figure for the fitness achieved over all runs by each benchmark function.
- **Poorest fitness:** The least best fitness level achieved across all runs by each benchmark function.
- **Standard deviation:** As mentioned, this is the standard deviation between the mean levels of fitness achieved in each individual run, aggregated for all runs.
- **Mean time:** The average time taken for each run by each benchmark function.

Table 18: Results for benchmark functions with dimensions $N = 20$, size = 18.

Function	Ackley	Sphere	Rosenbrock	Griewank
Best fitness	**2.6375E+00**	**1.2000E+01**	**0.0000E+00**	**4.1120E-01**
Mean fitness	**5.3458E+00**	1.2620E+03	**7.0764E+02**	**1.0834E+01**
Poorest fitness	**1.9595E+01**	4.4634E+04	**5.3914E+03**	**4.3180E+02**
Standard deviation	**1.1635E-01**	**4.0254E+01**	1.5574E+02	**1.022E+00**
Mean time (ms)	120.40	71.20	108.40	130.80

Table 19: Results for benchmark functions with dimensions $N = 30$, size = 20.

Function	Ackley	Sphere	Rosenbrock	Griewank
Best fitness	**3.276E+00**	2.6000E+01	**0.0000E+00**	**6.573E-01**
Mean fitness	**5.8233E+00**	**1.6646E+03**	3.1518E+03	**1.4016E+01**
Poorest fitness	**1.9640E+01**	7.9735E+04	**1.0195E+04**	**7.1847E+02**
Standard deviation	**1.7158E-01**	**1.4613E+02**	3.2299E+02	**7.6188E-01**
Mean time (ms)	270.40	160.00	240.80	290.40

Table 20: Results for benchmark functions with dimensions $N = 50$, size = 24.

Function	Ackley	Sphere	Rosenbrock	Griewank
Best fitness	**4.0217E+00**	**7.8000E+01**	3.8887E+03	**8.9071E-01**
Mean fitness	**6.931E+00**	**2.6504E+03**	1.4254E+04	**1.8769E+01**
Poorest fitness	**1.9800E+01**	1.4625E+05	**1.8408E+04**	**1.245E+03**
Standard deviation	**2.9957E-01**	**1.4614E+02**	1.9429E+03	**1.1489E+00**
Mean time (ms)	741.20	434.40	696.40	785.60

Table 21: Results for benchmark functions with dimensions $N = 70$, size = 26.

Function	Ackley	Sphere	Rosenbrock	Griewank
Best fitness	**4.9272E+00**	**1.4500E+02**	1.9924E+04	**9.9511E-01**
Mean fitness	**7.9936E+00**	**3.7159E+03**	2.4221E+04	**2.5144E+01**
Poorest fitness	**1.9873E+01**	2.0908E+05	**2.7458E+04**	**1.8833E+03**
Standard deviation	**2.8402E-01**	**2.4331E+02**	1.9516E+03	**1.2648E+00**
Mean time (ms)	1445.20	851.60	1394.80	1538.80

Table 22: Results for benchmark functions with dimensions $N = 100$, size = 30.

Function	Ackley	Sphere	Rosenbrock	Griewank
Best fitness	**6.0542E+00**	**4.2000E+02**	**2.9383E+04**	**1.1134E+00**
Mean fitness	**9.4993E+00**	6.0538E+03	3.5019E+04	**3.2981E+01**
Poorest fitness	**1.9894E+01**	**3.0353E+05**	**4.5821E+04**	**2.7054E+03**
Standard deviation	**4.1190E-01**	4.3564E+02	3.6994E+03	**1.3156E+00**
Mean time (ms)	2908.00	1819.60	2831.20	3194.00

Table 23: Results for benchmark functions with dimensions $N = 500$, size = 54.

Function	Ackley	Sphere	Rosenbrock	Griewank
Best fitness	**1.4433E+01**	5.7015E+04	1.9829E+05	**4.8773E+01**
Mean fitness	**1.6645E+01**	1.3874E+05	2.1792E+05	**3.0480E+02**
Poorest fitness	**1.9972E+01**	1.5935E+06	2.2998E+05	1.4501E+04
Standard deviation	**6.1797E-01**	2.7717E+04	6.9748E+03	**7.1154E+01**
Mean time (ms)	147818.00	95558.40	74689.60	134804.80

Table 24: Results for benchmark functions with dimensions $N = 1000$, size = 73.

Function	Ackley	Sphere	Rosenbrock	Griewank
Best fitness	**1.7250E+01**	3.2513E+05	4.3102E+05	**7.3685E+02**
Mean fitness	**1.8593E+01**	5.7175E+05	4.5025E+05	1.6362E+03
Poorest fitness	**1.9993E+01**	3.2470E+06	4.6860E+05	2.9311E+04
Standard deviation	**3.2277E-01**	1.1247E+05	1.0289E+04	**3.2838E+02**
Mean time (ms)	560022.80	419196.80	299853.20	590098.00

7.1.3 Timing Graphs

7.1.4 Convergence Graphs

7.2 Discussion of observations

7.2.1 Ackley

In terms of timing, Ackley is fine with 20 and 30 dimensions, but then begins to decay, coming second only to Griewank. That being said, the mean fitness levels achieved are consistently the best out of the benchmark set and even the poorer fitness levels cannot compare with even the better Rosenbrock or Sphere ones in the higher dimensions. The standard deviation between the mean best fitness levels achieved per run (see Figure 1) are also comparatively the lowest.

7.2.2 Sphere

The Sphere function is dependable in lower dimensions and doesn't decay too much in terms of time. This function does however achieve the poorest fitness levels once N is greater than 30. Whilst the means achieved do not deviate as much as those achieved by the Rosenbrock function, it is nonetheless prone to break in terms of deviations earlier.

7.2.3 Rosenbrock

Rosenbrock is not well behaved in the higher dimensions in spite of its better performance measures at the lower dimensions. A quick median fitness analysis for Rosenbrock at $N = 30$ is 0.77712, so it converges very quickly in lower dimensions. Time decay takes hold when $N = 70$, and so it does not achieve the correct fitness levels afterwards.

7.2.4 Griewank

Griewank decayed the most in terms of timing. Less deviation between the average time of each run for each function due to higher computational load and the sort mechanism in memory is the case here, since the function, whilst able to achieve a consistent fitness level, can also encounter much poorer fitness levels when in the process of evaluating.

Remark. *The next step will be to run these same tests in parallel to see what the performance improvement is by adjusting the number of processors. With the path planning problem, the serial evaluation will look at how increasing the number of particles affects performance, and thus with the parallel evaluation of that application, the same assessment will take place, but with the spread of N particles across multiple processors. The appendix will be further developed to accommodate more graphical analysis that cannot be included in the main body of the report.*

7.3 Increment and Problem Dimensionality

```
1 //Calculate swarm size based on dimensionality
2 int pso_calc_swarm_size(int dim) {
3   int size = 10. + 2. * sqrt(dim);
4   return (size > PSO_MAX_SIZE ? PSO_MAX_SIZE : size);
5 }
```

Listing 3: Function for calculating appropriate swarm size

All of the demo runs were performed with swarm sizes that were less than 100, with the highest, $N = 1000$ containing a swarm size of 73. An automatic calculation of the swarm size was performed by the function shown above, which was used in each case. The classical thinking on the population size is that the best performance typically occurs in populations with smaller sizes, hence the restricted allowance for incrementation with the problem dimensions. However, it has been argued recently that the smaller sizes selected to evaluated different optimisation problems may indeed be too small [16].

The authors of this paper have argued that the best results are typically obtained in the 70-500 size range. This is not a problem for the results obtained above as the range of sizes is incremented sufficiently, and evaluation of the impact of higher dimensions versus the increased population size is beyond the scope of this analysis. Nonetheless, it is evident that the impact of higher dimensions likely outweighs any influence a population size beyond 100 would have, as can be seen by the breakdown in performance of the Rosenbrock benchmark.

7.4 Debugging

Debugging can be complicated with the use of multiple `printf` statements, so for the purposes of this project `gdb` was used.

7.4.1 Gdb

`gdb` is a debugger that can be easily added to a program that is compiled with the `gcc` command. It is added to the `Makefile` in the following manner:

```
gcc myprog.c -o myprog -g
```

The `-g` aspect was removed in the instance where runs were used for the accurate measurement of times, as leaving the command in the code would slow down the program. Some faults that were uncovered in the final stages of the coding were common segmentation faults and double free or corruption faults, which were dealt with by adding `set environment MALLOC_CHECK_ 2` and `MALLOC_CHECK_ 1` to the debugger, to make invalid `free` commands visible in the backtrace.

7.5 Profiling

Profile was conducted using a mixture of Vampir, Score-P, and `gprof`, the latter of which was used for both the demo code and serial application. An example of how the profiling was run is given as

```
gprof -b ./prog ackley > analysis.txt
```

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 80.75      0.92     0.92        1     0.92     1.14  pso_solve
 12.29      1.06     0.14   200040     0.00     0.00  pso_ackley
  7.02      1.14     0.08    10001     0.00     0.00  inform
  0.00      1.14     0.00    10001     0.00     0.00  calc_inertia_lin_dec
  0.00      1.14     0.00    10001     0.00     0.00  inform_ring
  0.00      1.14     0.00        4     0.00     0.00  pso_matrix_new
  0.00      1.14     0.00        1     0.00     0.00  elapsed_time
  0.00      1.14     0.00        1     0.00     0.00  end_timer
  0.00      1.14     0.00        1     0.00     0.00  init_comm_ring
  0.00      1.14     0.00        1     0.00     0.00  parse_arguments
  0.00      1.14     0.00        1     0.00     0.00  print_elapsed_time
  0.00      1.14     0.00        1     0.00     0.00  pso_calc_swarm_size
  0.00      1.14     0.00        1     0.00     1.14  pso_demo
  0.00      1.14     0.00        1     0.00     0.00  pso_settings_free
  0.00      1.14     0.00        1     0.00     0.00  pso_settings_new
  0.00      1.14     0.00        1     0.00     0.00  start_timer
```

As evidenced by the `gprof` profiling of the demo code, the key sections that the parallel implementation will seek to improve will include the `pso_solve` algorithm elements, the topological functions and related inform functions (`inform_ring`, `inform_random`, and `inform_global`), and the elements that deal with the allocation of matrices.

As well as these functions, the evidence from the profiling suggests that the calculation of the inertia weight should be sped up in light of the call count on that function (`calc_inertia_lin_dec`).

7.6 Parallel Profiling

7.7 L2 Cache count

7.8 Serial Path analysis

```
1
2  /* Path options */
3  int inUavID = 0;
4  double inStartX = 70.0;
5  double inStartY = 70.0;
6  double inEndX = 136.0;
7  double inEndY = 127.0;
8  double inStepSize = 1;
9  double inVelocity = 2;
10 double inOriginX = 0;
11 double inOriginY = 0;
12 double inHorizonX = 200;
13 double inHorizonY = 200;  // 70
14 int waypoints = 5;
15
16 /* PSO parameters */
17 double pso_c1 = -1.0;
18 double pso_c2 = -1.0;
19 double pso_w_max = -1.0;
20 double pso_w_min = -1.0;
21 int pso_w_strategy_select = -1;
22 int pso_nhood_size = -1;
23 int pso_nhood_topology_select = -1;
24
25 int pso_w_strategy = -1;
26 int pso_nhood_topology = -1;
```

Listing 4: Path parameter settings and additional PSO settings

In addition to the parameter settings for the demo analysis, the serial parameter settings are shown above. The previous settings that were used for the highest range during the demo analysis were left unchanged and are exactly as shown in that table.

Note: increasing the number of particles through the `PopSize` variable will break the `gsl` generator and lead to a memory leak, so this will be removed. The correct way in this instance will be to adjust the dimensions via the multiplier in the code, as the size of the swarm is conditional on the number of dimensions. This gives us the best of both worlds in the sense that we can test for the effects of an increased number of dimensions as well as an increase in the size of the swarm.

7.8.1 Number of obstacles per run

This merits a discussion as the informed observer would think that a random generation of obstacles would give different results no matter how many dimensions each run was conducted in. This is not the case as tests were carried out with a map containing pre-existing obstacles and there were no measurable effects on timing found.

7.8.2 Testing

Testing was conducted on the provided map `sample_map_Doorways.txt` to check for the influence on obstacles on the timing of the initial runs and it was found that they had no real measurable effect. The same can be said for the effect on the error. This is likely due to the low number of obstacles included with the map; this number would have to be increased if there are to be any noticeable effects, but this is beyond the scope of this report.

NOTE: below should be changed to a stacked bar graph upon completion of the parallel tests.

Remark. *Will upgrade to include parallel timing.*

Figure 3: Swarm initialisation

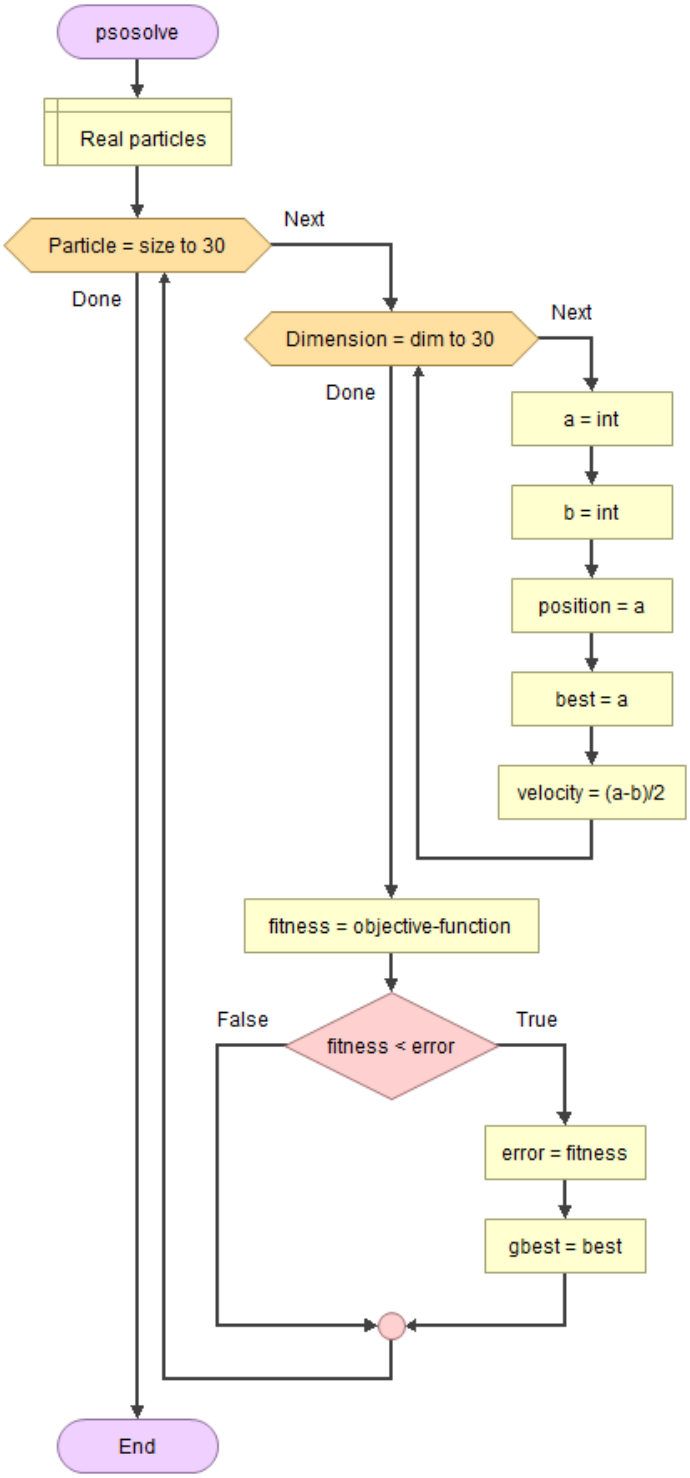


Figure 4: PSO algorithm

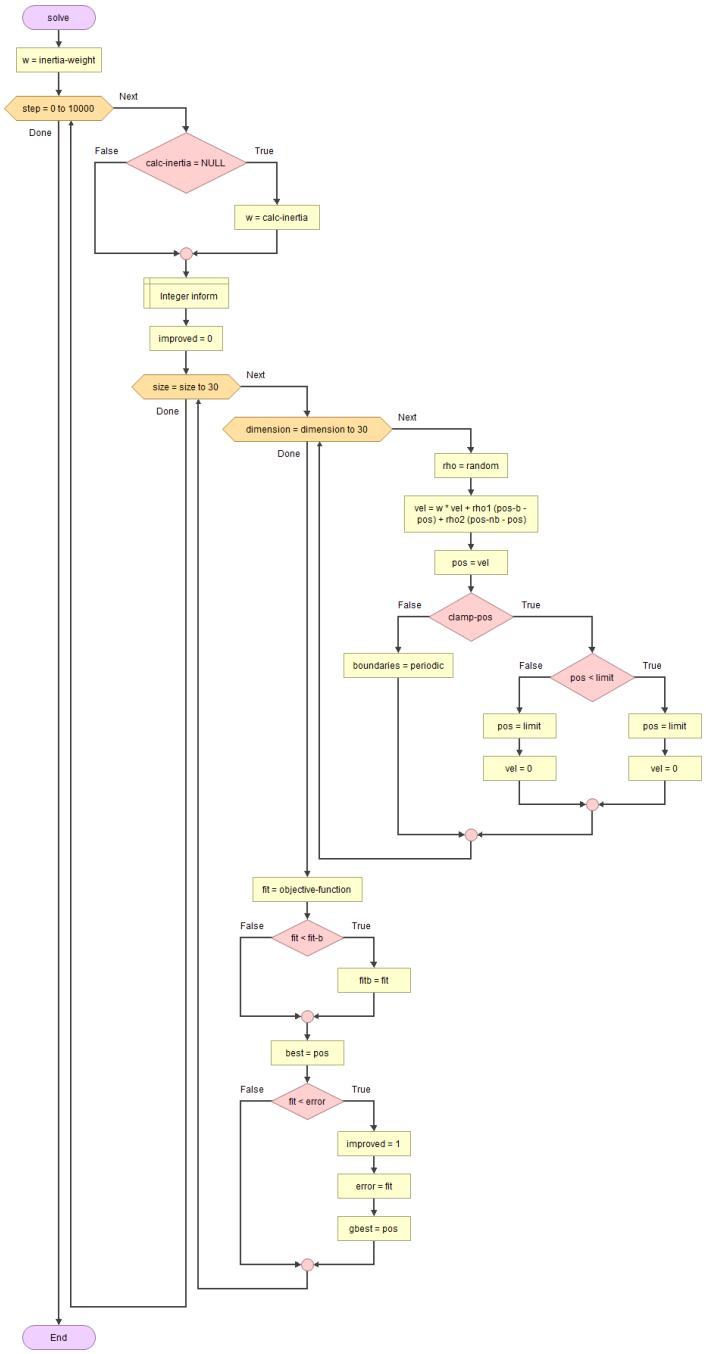
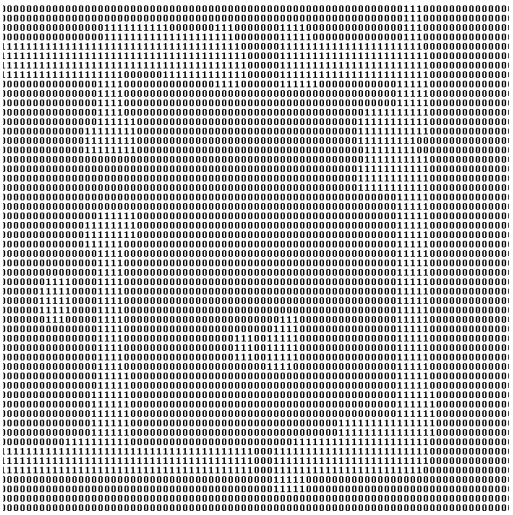


Figure 5: Cropped image of ASCII map illustrating obstacles



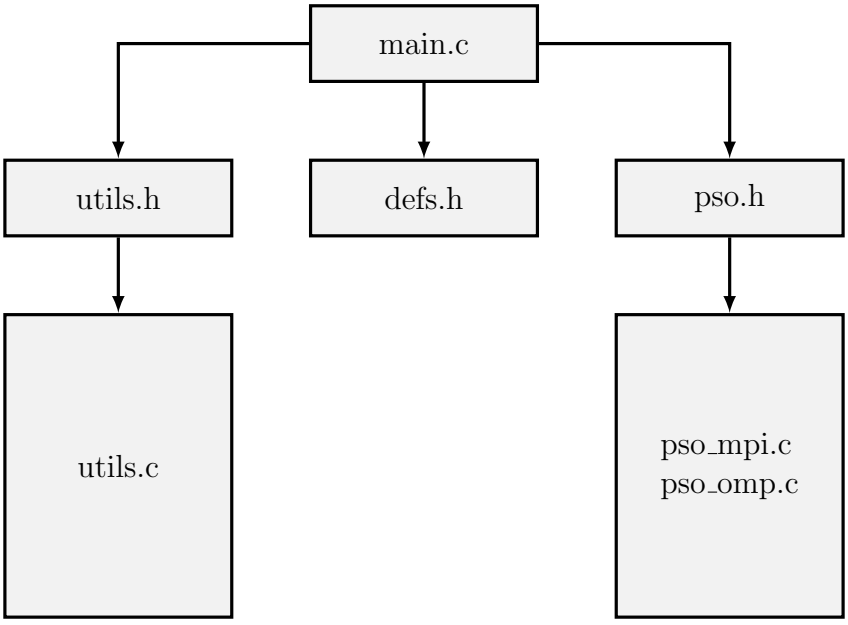


Figure 6: Tree diagram of parallel code for demo implementation (path files excluded.)

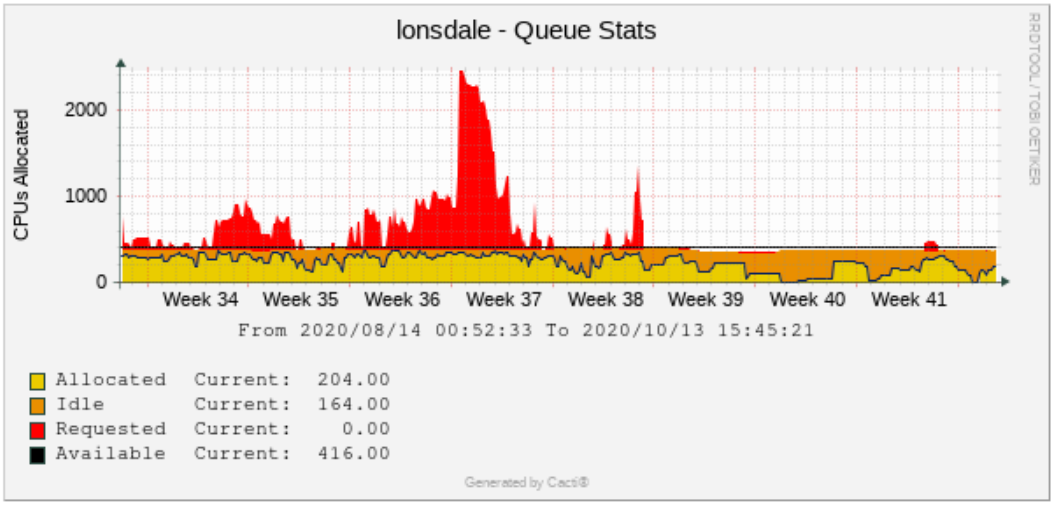
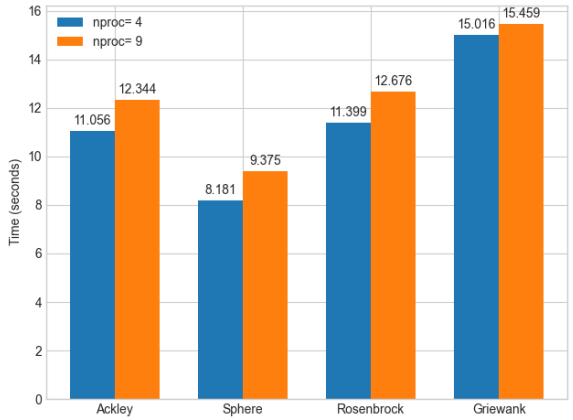
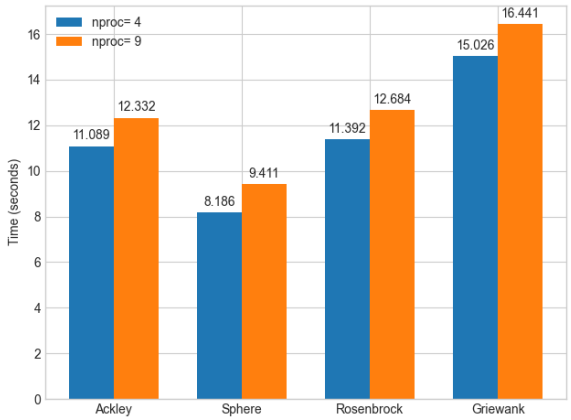


Figure 7: The graph shows the number of Requested, Allocated and Idle CPUs in the Lonsdale cluster over a monthly basis.



(a) Timings for MPI setup for `nproc` of 4 and 9.



(b) Timings for hybrid MPI-OpenMP setup for `nproc` of 4 and 9.

Figure 8: Bar plot of timing for each function in both code versions.

Figure 9: Standard deviation of the average fitness achieved across 25 runs for the Ackley function.

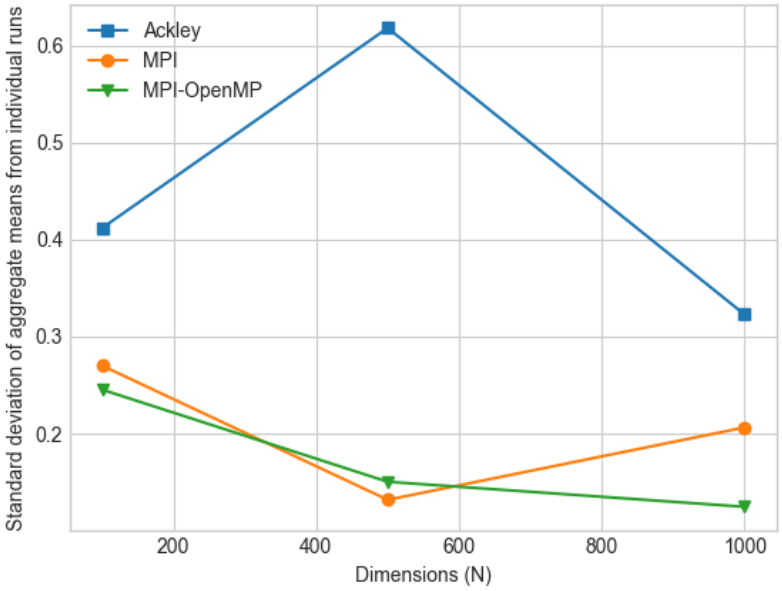


Figure 10: Standard deviation of the average fitness achieved across 25 runs for the Sphere function.

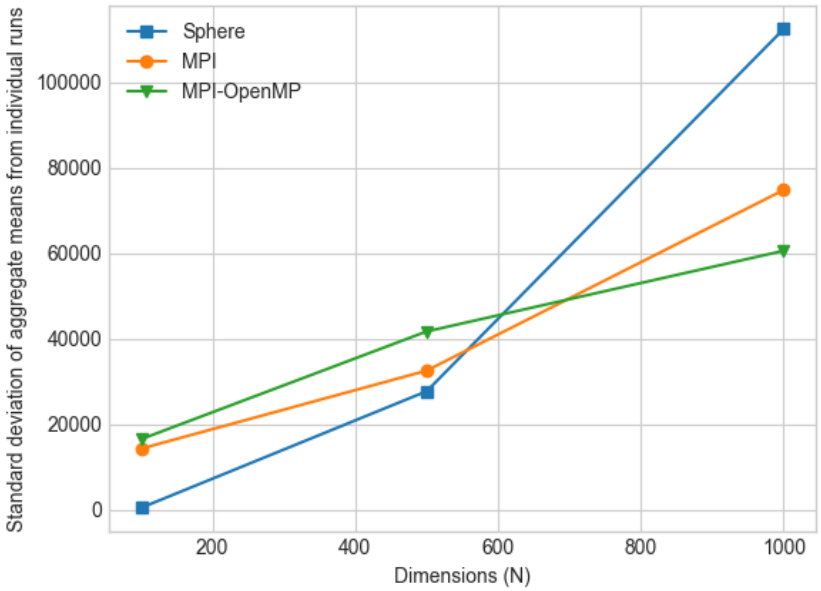


Figure 11: Standard deviation of the average fitness achieved across 25 runs for the Rosenbrock function.

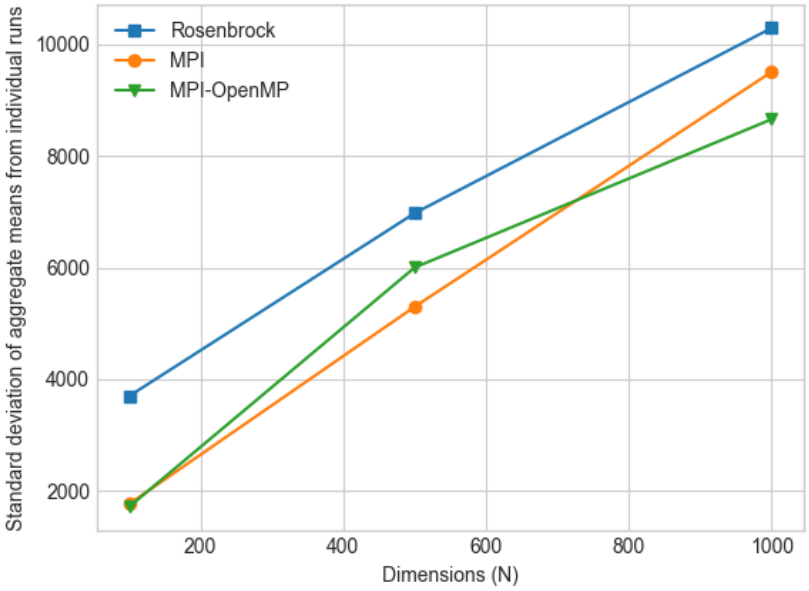


Figure 12: Standard deviation of the average fitness achieved across 25 runs for the Griewank function.

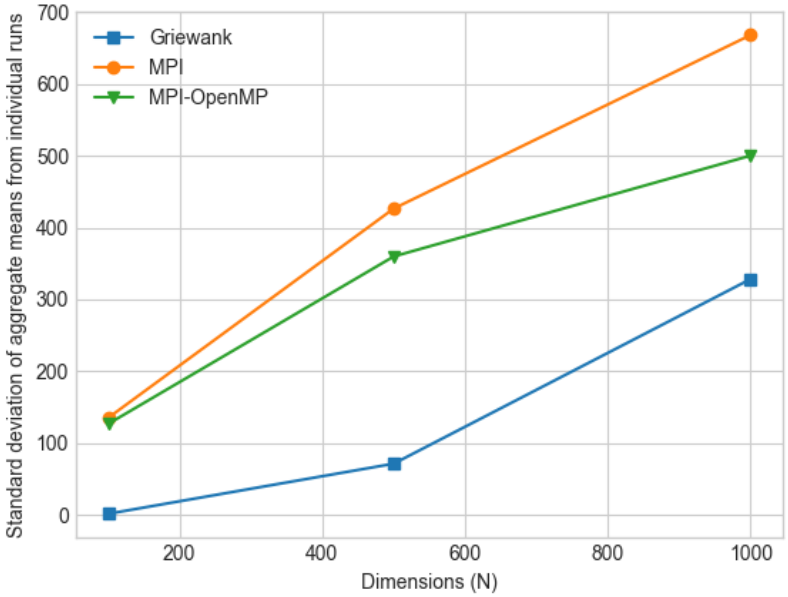


Figure 13: Speedup for each function.

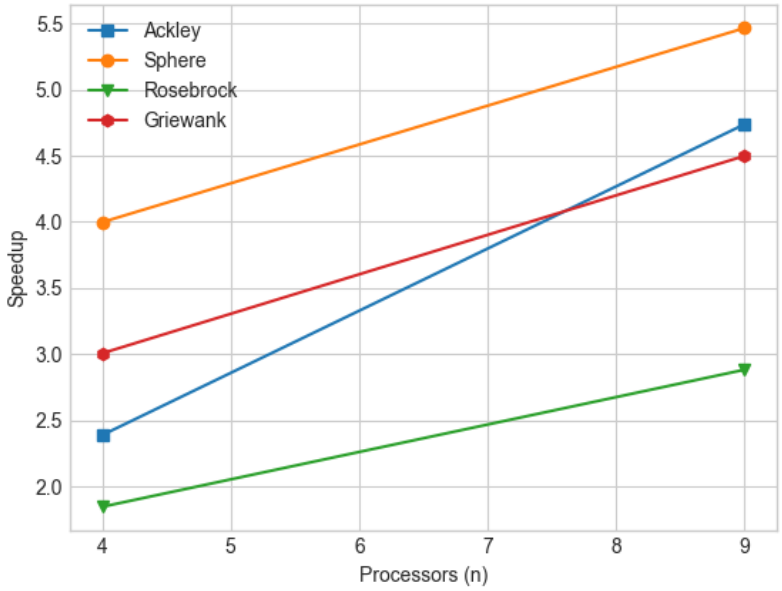


Figure 14: Timing for both serial and parallel functions (Ackley).

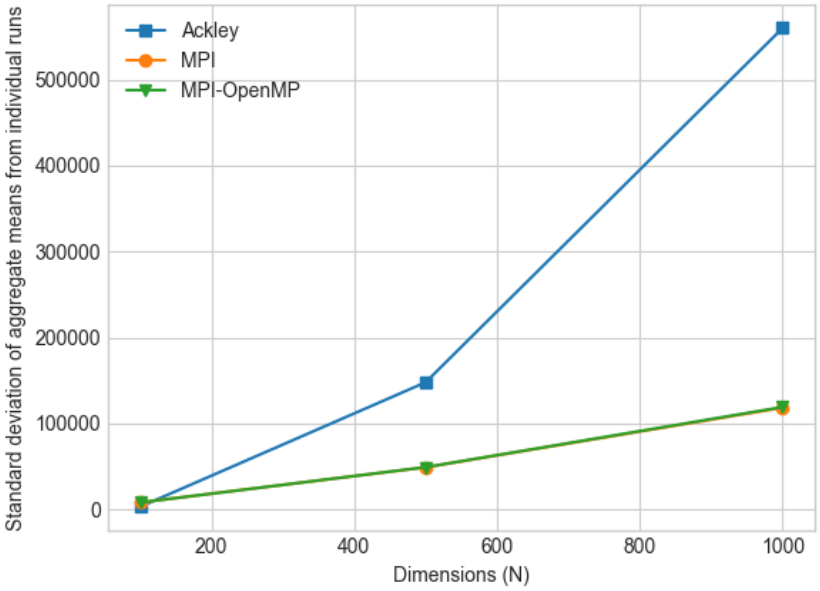


Figure 15: Timing for both serial and parallel functions (Sphere).

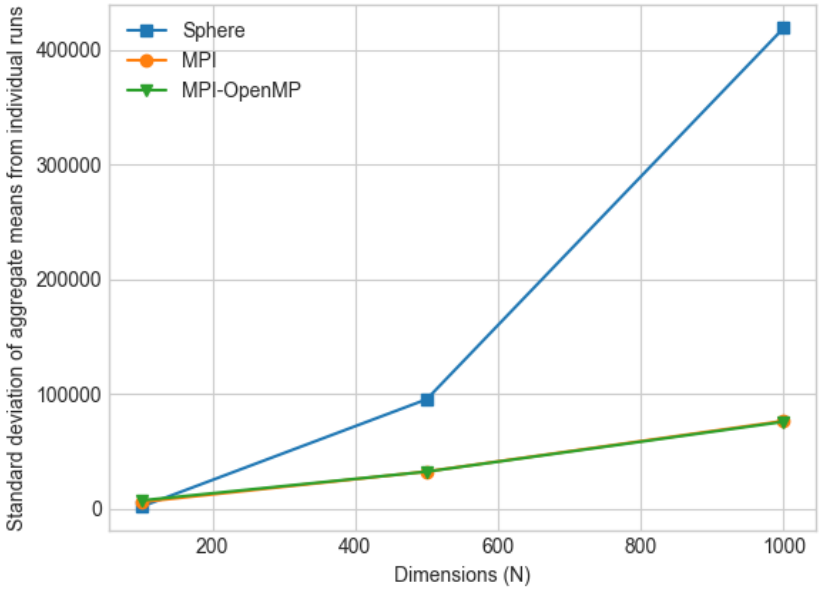


Figure 16: Timing for both serial and parallel functions (Rosenbrock).

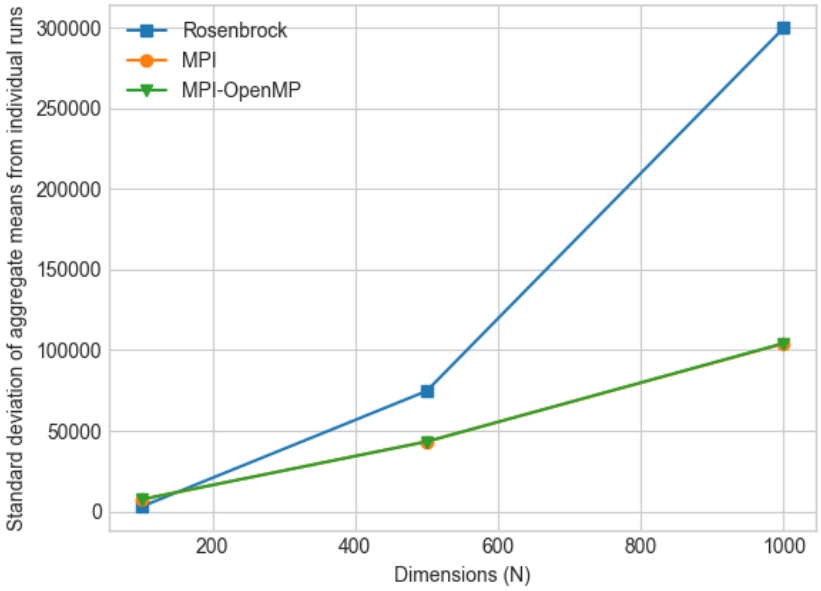


Figure 17: Timing for both serial and parallel functions (Griewank).

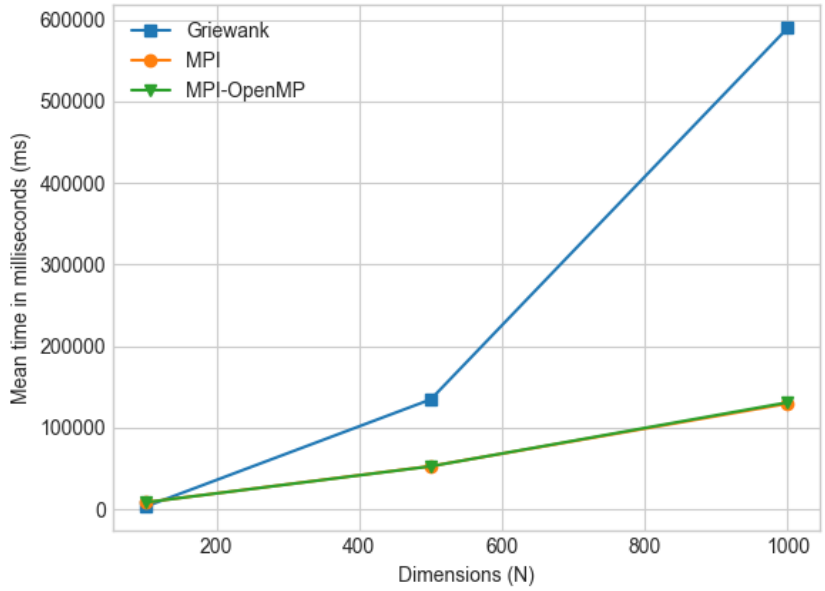


Figure 18: Standard deviations of the means illustrate the break in consistency with the Rosenbrock benchmark.

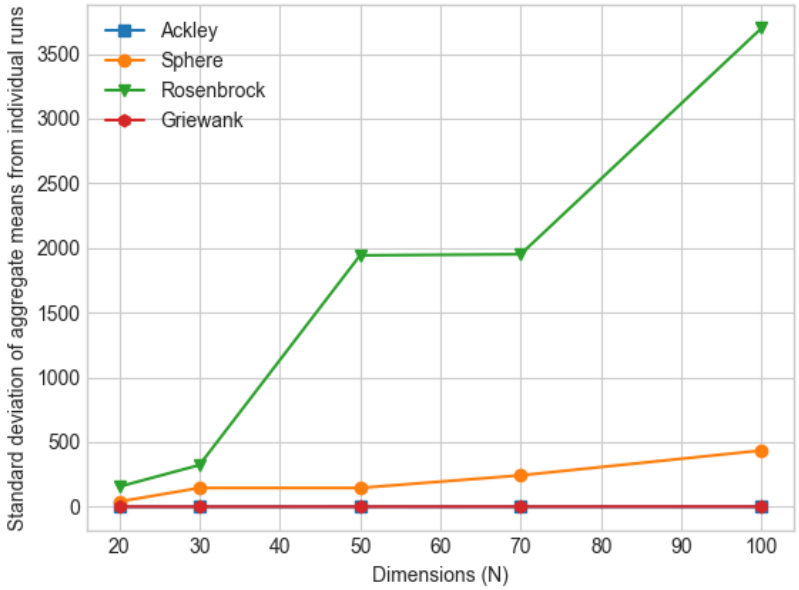


Figure 19: Standard deviations of the means up to $N = 1000$.

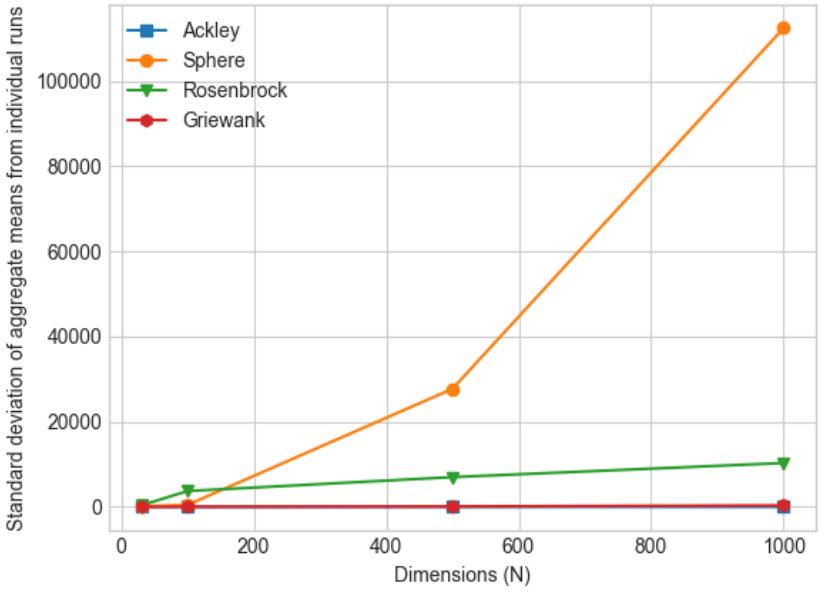
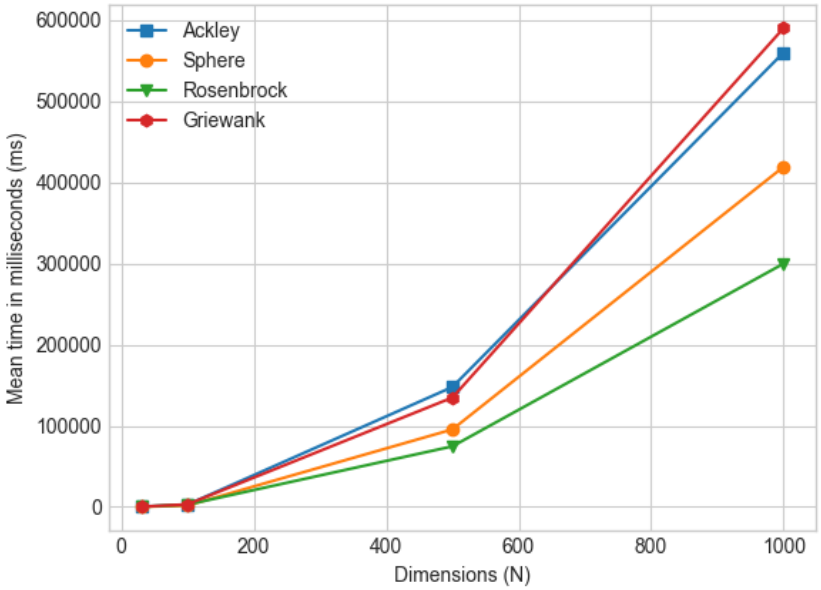
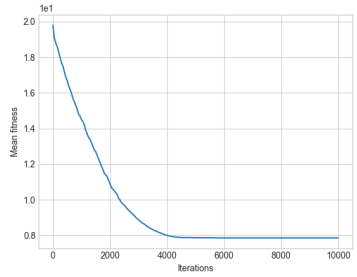
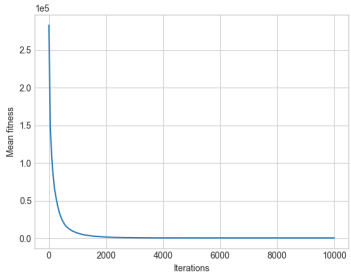


Figure 20: Average time per run for each function.

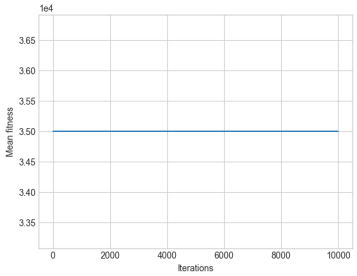




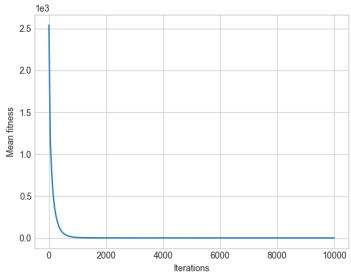
(a) Ackley



(b) Sphere



(c) Rosenbrock



(d) Griewank

Figure 21: Convergence graphs for each function at $N = 100$ dimensions.

Figure 22: Function Summary.

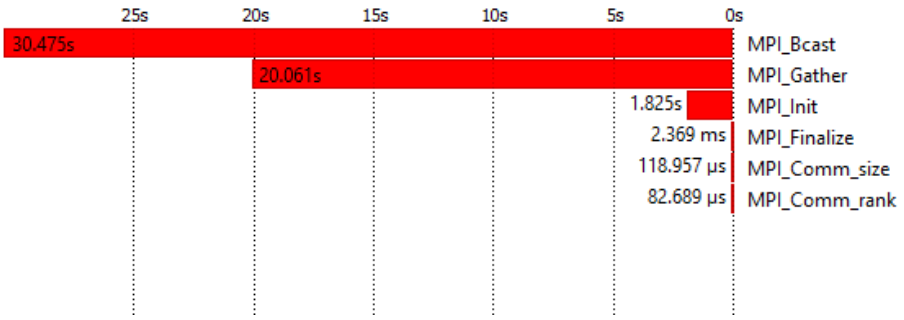


Figure 23: L2 Cache misses over time.

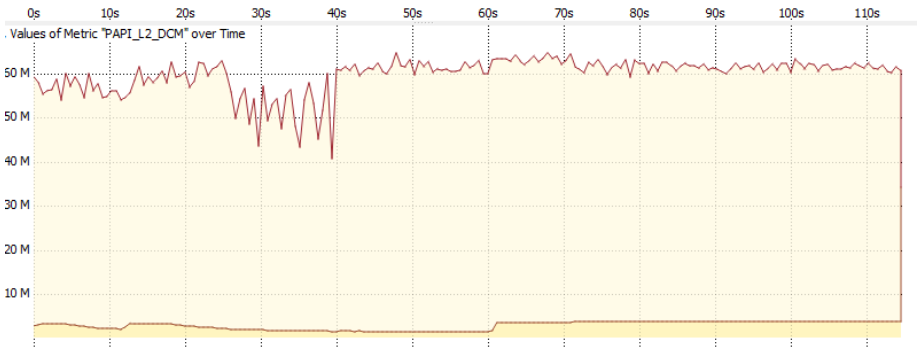


Figure 24: Average solution distance.

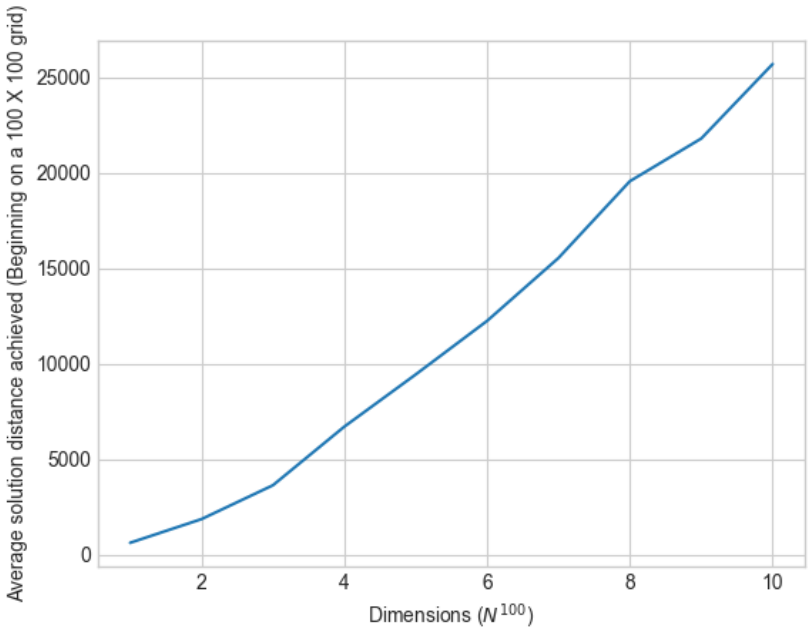


Figure 25: Timing across runs.

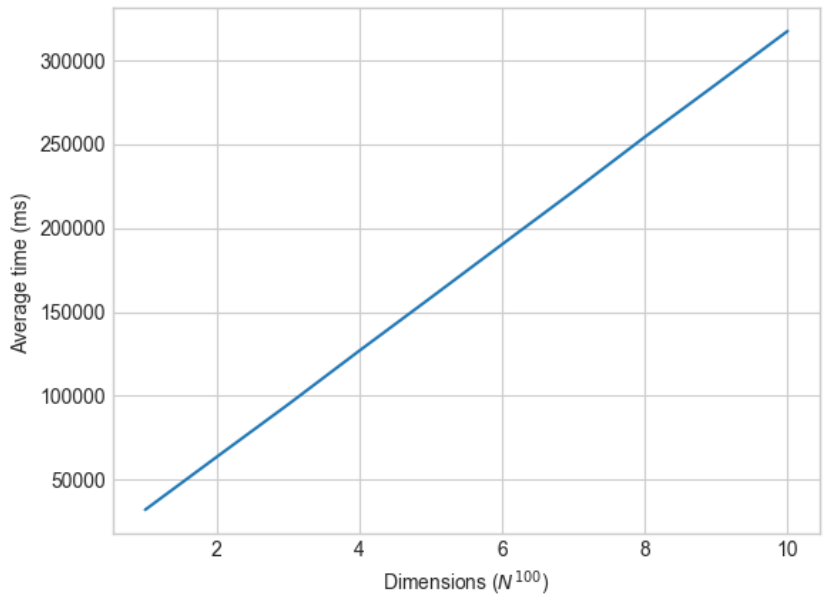


Figure 26: Convergence graph.

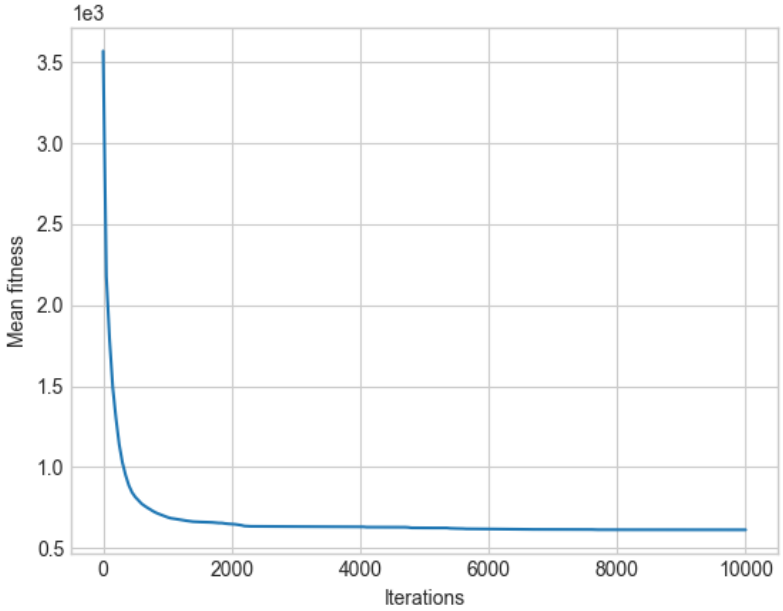


Table 25: Results across different $N^1$00.

N	1	2	3	4	5	6	7	8	9	10
Obstacles (avg)	0	19	99.6	264.8	548	804.4	1284	2058	2034.8	2812.4
Solution distance (avg)	611.78	1854.10	3628.08	6699.95	9435.24	12243.68	15541.26	19564.87	21802.40	25707.28
Avg error	6.6852E+02	2.0838E+03	3.9828E+03	7.1107E+03	9.8900E+03	1.2818E+04	1.6163E+04	2.0074E+04	2.2412E+04	2.6416E+04
Best	4.3074E+02	1.3755E+03	2.1111E+03	5.0095E+03	7.2606E+03	8.1969E+03	1.2458E+04	1.6955E+04	1.9115E+04	2.1504E+04
Worst	4.2178E+03	8.866E+03	1.4031E+04	1.9283E+04	2.3186E+04	2.9204E+04	3.3699E+04	3.8305E+04	4.3142E+04	4.7813E+04
Standard deviation	1.1201E+02	3.3027E+02	6.8822E+02	8.2608E+02	1.0447E+03	1.2427E+03	1.5363E+03	1.1984E+03	1.4494E+03	1.5479E+03
Time (average,ms)	31956	63414.8	94690	126811.2	158425.2	190279.6	221928	254361.2	285816.4	317517.2

7.9 PSO Topologies

Each of the following topologies is used to determine the matrix strategy used to update the neighbouring particles upon running the algorithm. It should be noted that the testing phases for the serial and demo versions specified the global topology as the default for each test. Upon completion of the parallel version, the each topology will be tested with $N = 100$ and compared with the resulting parallel implementation for different numbers of processes.

Both the random and ring topologies are governed by a general inform function, which copies `pos_b` of each particle to `pos_nb` of the matrix itself. This takes the principle of the best informer and applies it to the general function in `pso_solve`. The global inform function is unconnected to this function and is the default function for this purpose.

The idea will be to both initialise and execute the inform communication within a singular function for both global, ring, and random topologies to avoid the communication overhead associated with the initialising and execution of each function separately. This is where `MPI_Cart_create` will be employed to create three separate topologies for such a purpose.

7.9.1 Global

In this case, all particles are drawn to the best position `pos_b` at that particular iteration and copy the contents of `pos_b` to the next best position, `pos_nb`.

7.9.2 Ring

This is a fixed topology. The array is reset and informer particles are chosen, with the diagonal being set to 1. The adjacent particles inform each other in a ring fashion.

7.9.3 Random

An average number of particles is chosen to act as informers, with a random integer generated to denote the topology to be chosen. Each particle informs itself in this case at the beginning, with particle i informing particle j thereafter.

7.10 Parallel Implementation

7.11 MPI functions

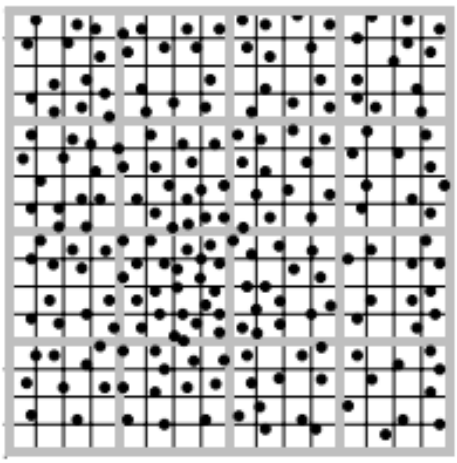
7.11.1 `MPI_Gather`

7.11.2 MPI topologies

The general inform function will be cut out due to the ability of `MPI_Cart_shift` and related functions to enable efficient communication between processes, which will be mapped with the individual values for `pos_b` and `pos_nb`. These two variables will feature prominently in the new topological set-ups given their link to delivering a solution.

7.11.3 Distributed particles

Figure 27: Illustration of distributed particles



The particles are distributed to processes by assigning them different processes mapped out onto a Cartesian grid. The node number, location within the process mesh, and the corresponding details for the neighbouring processes will all be found and each process will be updated with this information accordingly. Importantly, where periodic boundaries are enforced, particles within the different cells located in these regions will need to know the about particles in the neighbouring cells and update their best positions. The number of neighbours is 8; within a 5-point stencil grid, the diagonal can be excluded, but this is not the case here.

7.11.4 Particle migration

This becomes an issue when different particles migrate between different cells where communication takes place between them. This is important for the evaluation of the algorithm and for eventually reaching a global optimum. By allowing particles lying along the boundaries communicate with each other, the global best position can be reached. The receiving process will determine how many particles have migrated. It is not possible to tell before this takes place i.e. how many particles will be communicated. The number of bytes received on the receive buffer indicates how many have migrated. The number of particles on each processes will change over time, and the need for communication between processes is paramount for this reason. The distribution will be evened out over time, and the study of how evenly they are distributed is something to be looked at in future reporting on the subject.

7.12 Algorithm

In terms of the parallelisation that takes place on the Cartesian, the above explanation should suffice in describing how it takes place. The workings of the Cartesian can be conceptualised in terms of a linked list set-up that transfers the data given by the pointer to another linked list. The linked lists in question are then updated in a bookkeeping. In summary, the process requires a data structure that is allowed to change proportions. When the data is sent, the pointers are allowed to change to make the list consistent. It should be noted that the `MPI_Sendrecv` step in this process does not update the pointers - this is done using regular C functions.

7.12.1 Pseudocode

Algorithm 2: Parallel PSO algorithm pseudocode

```
for each step do
    Communicate particle data for boundary cells;
    update current step;
    update inertia weight if required;
    //Check optimization goal;
    update pos_nb matrix;
    reset improved value;
    //Update all;
    for each particle do
        find the location (i,j) within cell the current particle is in;
        for each dimension do
            calculate ρ1 and ρ2;
            update velocity;
            for each cell i, j do
                update position;
            end
        end
    end
    for each particle do
        for each dimension do
            update particle fitness;
            update personal best position;
            copy pos i to pos_b i (migration step)
        end
    end
    //update gbest?;
    for each particle do
        for each dimension do
            copy to gbest if required;
        end
    end
end
```

7.12.2 New topologies

Torus Cylinder Grid

7.12.3 MPI_Cart_*

Although the Cartesian decentralises the distribution of data, `rank 0` is the nominal receiver of data for the algorithm when it comes to output. The global output is given in array form, so it is necessary to account for this.

7.12.4 The replacement of `memmove()`

It is certainly the case that, at least in serial code, the C standard version of `memmove` is perfectly efficient at copying from source to destination. It is possible to implement a version of this function using bare code in serial, but it creates severe overheads where the source and destination buffer overlap. This will lead to undefined behaviour. There is a way to counter this overlap and this will be implemented here where MPI is concerned.

In summary, the goal is to change a standard way of data movement through a value assignment to something that is parallelisable and scalable. The latter part, that of scalability, is the most challenging aspect of this project, but there are already MPI functions that perform a similar role. For instance, `MPI_Reduce` calculates the values of the data to be transmitted, with the MPI libraries then assigning the destination buffer thereafter.

7.12.5 Scatter-Gather

Questions to answer: What are you trying to do with scatter-gather? And how does the example that you've done out relate to it?

Get schema for MPI_scatter and relate it to how bcast is working right now
Detail new way to update velocity(new function for rho and what pos_nb was doing before)
Do out diagram for code map

7.13 OpenMP

For the OpenMP implementations: discuss the type of schedules (static, dynamic, guided, etc.) that have been implemented. Explain why you went with the particular implementation you went with, and how else you could have sectioned up the code with OMP.

OpenMP can generally be used where particles can be updated independently (see [24]). The methodology employed here to update particles

in the implementation implies that there is no communication between particles except the step of updating the best model for this particular implementation.

7.14 Instrumentation

7.14.1 Timing

There are a couple of changes that needed to be made to the algorithmic setup prior to testing. A new timer was implemented using the `sys/time.h` header library, and placed within the solving function itself so that an accurate performance for the MPI process could be gauged.

7.14.2 Score-P

Score-P is a joint performance measurement run-time infrastructure set-up that can be used in conjunction with other profilers such as Vampir (see next subsection). It is an instrumentation tool that contains several run-time libraries and helper tools, and uses common output formats (`OTF2, CUBE4`). It is useful for the generation of call-path profiles and event traces, for using direct instrument and sampling, flexible measurement without need for re-compilation, recording time, visits, communication data, and hardware counters. In order to use it, simply prefix all compile/link commands with `scorep` and affix the advanced instrumentation (detailed in the next subsection).

7.15 Profiling

Vampir version 9.9 was selected for the purposes of profiling the code, with Score-P enabled as the primary code instrumentation and run-time measurement framework for Vampir. This has a wide use base including instrumentation at both source level and at compile/link time.

7.15.1 Vampir

The VampirTrace library for tracing parallel programmes and visualisation tool to allows the user to examine program execution results at two distinct levels: it allows the user to monitor and collect data relating to the program's performance and execution, and it allows the user to look at charts and browse through different metrics that have been cataloged. It was developed by TU Dresden and is open source software. The key benefit of using this program is that is shows dynamic run-time behaviour graphically at any level of detail, and provides statistics and performance metrics, timeline charts (application activities and communication along

time axis) and summary charts (quantitative results for currently selected time intervals). Had to run as administrator to get the demo version to work.

7.15.2 Use Case

The following commands were used to instrument for Score-P in Vampir, with exports for advanced instrumentation metrics including the number of L2 cache misses:

```
$ scorep mpicc -Wall -L '/home/support/apps/cports/rhel-6.
x86\_64/gnu/papi/5.6.0/lib'-lmpi -o a.out example.c
$ export SCOREP_ENABLE_TRACING=true
$ export SCOREP_EXPERIMENT_DIRECTORY=path/to/directory
$ export SCOREP_METRIC_PAPI=PAPI_L2_DCM
```

The program is then executed with the `.otf2` files being used for the trace analysis and profiling.

Results

8 Discussion of Results and Future Research

8.1 Revisions

List of possible MPI revisions that could be made.

8.2 Initial Observations

The converged fitness is typically achieved quicker with the hybrid version. This is to be expected given the quicker performance achieved in the lower dimensions and the efficient multithreading architecture at this performance level. However, the standard deviation is variable when it comes to the parallel function applications, and this is evident if we look at the Griewank and Ackley functions in particular. The standard deviation of the means is worse in the Griewank function and better with the Ackley function evaluated in parallel. This is due to the more numerous minima present in the Griewank function, which leads to less efficient speedups.

8.3 Cartesian topology

8.4 CUDA

Compute Unified Device Architecture (CUDA) is another popular way of implementing a parallel program, utilising the GPU power of a machine. As mentioned previously in the literature review, MPI is still the most effective way of incoroporating parallelism into PSO applications on multiple cores [11]. That being said, CUDA implementations are equally as useful where GPU processing power and the limitations of CPUs are considered, particularly in the case of PSO [25]. The research by Zou et al. demonstrates a unique approach to the problem of developing intelligent optimization algorithms (IOA) with the use of a OpenMP and CUDA utilization with GA-PSO hybrid implementations. Their results demonstrated particularly usefulness for industrial applications, and a sample of the PSO code can be seen below.

The implementation of PSO using CUDA above figure shows the process of allocating memory for the CPU and the GPU, which are used to store each of the variables. The following box figure shows the kernel function to realize the updating of the velocity/speed, locations, and individual optimal solution. Finally, the last part demonstrates the process of coping kernel function results from GPU memory to CPU memory, and then releasing previous memory spaces.

8.5 Premature Convergence

It is often said that due to the simplistic nature of the PSO algorithm, objective functions and their complexities do sometimes lead the algorithm down the wrong path, or sometimes instill the notion that convergence to local optima occurs too early [18].

Appendix A

Acronyms

ASCII American Standard Code for Information Interchange

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

GA Genetic Algorithm

GDB GNU Debugger

GPU Graphical Processing Unit

HSI Hyper-Spectral Imagery

IOA Intelligent Optimization Algorithms

MPI Message Processing Interface

OMP OpenMP

PSO Particle Swarm Optimisation

TRN Terrain-Referenced Navigation

TSP Travelling Salesman Problem

UAV Unmanned Aerial Vehicle

VCS Version Control System

Citations

Citations were generated using: https://www.citationmachine.net/apa/cite-a-website.

Code

The implementations used for this project are available at: https://github.com/seancmry/msc_pro. If for any reason the code is not available at the URL, email: murras11@tcd.ie.

Flowcharts

Flowcharts were created using Flowgorithm: http://www.flowgorithm.org/index.htm. These files were first saved as `.fprg` files and are available in the Github repository as `.png` files.

Descriptive statistics

All descriptive statistics were filtered and calculated through Microsoft Excel. The datasets used contains any formulae used in these calculations and are available in the Github repository.

8.6 Version Control

Explain what Git stuff you did. For the purposes of running version control on Lonsdale, a remote set-up was established in the manner outlined below.

8.6.1 SSH keys

- An SSH key was first created using `ssh-keygen -t rsa`
- A folder called `ids` with the new keys were generated in the `.ssh` folder with an executable permission added using `chmod +x ~/.ssh/`, followed by `chmod 400 ~/.ssh/*`, adjusting permissions of the files of the public keys and private keys, which should only be readable by the user and not anything by any other user.
- Finally, the key was copied to Lonsdale using `ssh-copy-id -i ~/.ssh/ids murras11@lonsdale.tchpc.tcd.ie`.
- The key was then tested using `ssh -i ~/.ssh/ids.pub murras11@lonsdale.tchpc.tcd.ie`.

We then have set up a server which can be used to push/pull to and from Lonsdale. Using `git clone` then allowed for the existing repository, which is available on both my own machine and at https://github.com/seancmry/msc_pro, to be visible on Lonsdale.

Appendix B

	Lonsdale	Kelvin
Vendor:	ClusterVision	Dell
Available to:	TCD Researchers	Irish researchers
Processor Type:	Opteron	Intel
Architecture:	64-bit	64-bit
Number of Nodes:	154	100
RAM per node:	16	24
RAM:	3.2TB (128x16GB, 24x32GB, 2x64GB)	2.4TB
Clock Speed:	2.30GHz	2.66GHz
Interconnect:	Infiniband DDR	Qlogic Infiniband QDR
Theoretical Peak Performance:	11.33TF	12.76TF
Total number of cores:	1232	1200
Number of sockets per Node:	2	2
Number of Cores per Socket:	4	6
Linpack Score:	8.9TF	10.8TF

Table 26: TCHPC Cluster System Specifications

References

[1] Altinoz, O.T., Yilmaz, A.E., and Ciuprina, G. (2013) 'Impact of problem dimension on the execution time of parallel particle swarm optimization implementation'. 2013 8th International Symposium on Advanced Topics in Electrical Engineering (ATEE), Bucharest, pp. 1-6, doi: 10.1109/ATEE.2013.6563482.

[2] Arasomwan, M.A. and Adewumi, A.O. (2014) 'Improved particle swarm optimization with a collective local unimodal search for continuous optimization problems'. The Scientific World Journal, 2014, pp.1-23, doi: https://doi.org/10.1155/2014/798129.

[3] Chang, Y., Fang, J., Benediktsson, J., Chang, L., Ren, H., and Chen, K. (2009) 'Band Selection for Hyperspectral Images based on Parallel Particle Swarm Optimization Schemes'. 2009 IEEE International Geoscience and Remote Sensing Symposium, 5, pp. 84-87, doi: 10.1109/IGARSS.2009.5417728.

[4] Chen, A., (2018) 'The Mars 2020 Entry, Descent, and Landing System', Jet Propulsion Laboratory, National Aeronautics and Space Administration (NASA), Pasadena, CA, doi: https://trs.jpl.nasa.gov/bitstream/handle/2014/50084/CL%2318-3099.pdf?sequence=1&isAllowed=y.

[5] Corner, J.J., and Lamont, G.B., (2004) 'Parallel simulation of UAV swarm scenarios'. Proceedings of the 2004 Winter Simulation Conference, vol. 1.

[6] Dang, W., Xu, K., Yin, Q., and Zhang, Q. (2014) 'A Path Planning Algorithm Based on Parallel Particle Swarm Optimization'. In: Huang DS., Bevilacqua V., Premaratne P. (eds) Intelligent Computing Theory. ICIC 2014. Lecture Notes in Computer Science, 8588. Springer, Cham., doi: https://doi.org/10.1007/978-3-319-09333-8_10.

[7] Gropp, W., and Thakur, R. (2007) 'Thread-safety in an MPI implementation: Requirements and analysis'. Parallel Computing, 33(9), pp. 595-604.

[8] Ioannis, Z., Brehm, J., and Akselrod, M. (2013) 'Function Based Benchmarks to Abstract Parallel Hardware and Predict Efficient Code Partitioning'. 26th International Conference on Architecture of Computing Systems 2013, pp. 1-13.

[9] Kennedy, J. and Eberhart, R. (1995) 'Particle Swarm Optimization'. Proceedings of the IEEE International Conference on Neural Networks, 4, pp. 1942-1948.

[10] Koçer, B. (2015) 'The Analysis of GR202 and Berlin 52 Datasets by Ant Colony Algorithm'. 2015 4th International Conference on Advanced Computer Science Applications and Technologies (ACSAT), pp. 103-108.

[11] Lalwani, S., Sharma, H., Chandra Satapathy, S., Deep, K., and Chand Bansal, J. (2019) 'A Survey on Parallel Particle Swarm Optimization Algorithms'. Arabian Journal for Science and Engineering, 44, pp. 2899–2923.

[12] Li, Y., Cao, Y., Liu, Z., Liu, Y., and Jiang, Q. (2009) 'Dynamic optimal reactive power dispatch based on parallel particle swarm optimization algorithm'. Computers Mathematics with Applications 57, no. 11-12, pp. 1835-1842.

[13] Moraes, A. O., Mitre, J. F., Lage, P. L., Secchi, A. R. (2015) 'A robust parallel algorithm of the particle swarm optimization method for large dimensional engineering problems'. Applied Mathematical Modelling, 39(14), 4223-4241.

[14] Nedjah, N., de M. Calazan, R., de Macedo Mourelle, L., and Wang, C. (2016) 'Parallel implementations of the cooperative particle swarm optimization on many-core and multi-core architectures'. International Journal of Parallel Programming 44, no. 6, pp. 1173-1199.

[15] Omkar, S.N., Venkatesh, A., and Mudigere, M. (2012) 'MPI-based parallel synchronous vector evaluated particle swarm optimization for multi-objective design optimization of composite structures'. Engineering Applications of Artificial Intelligence, 25, pp. 1611–1627, doi: 10.1016/j.engappai.2012.05.019.

[16] Piotrowski, A.P., Napiorkowski, J.J., and Piotrowska, A.E. (2020) 'Population Size in Particle Swarm Optimization'. Swarm and Evolutionary Computation, 58, doi: https://doi.org/10.1016/j.swevo.2020.100718.

[17] Roberge, V., Tarbouchi, M., and Labonté, G. (2012) 'Comparison of parallel genetic algorithm and particle swarm optimization for real-time UAV path planning'. IEEE Transactions on industrial informatics, 9, 1, pp. 132-141.

[18] Sengupta, S., Basak, S., and Peters, R.A. (2019) 'Particle Swarm Optimization: A survey of historical and recent developments with hybridization perspectives'. Machine Learning and Knowledge Extraction, 1(1), pp. 157-191.

[19] Shakhatreh, H., Khreishah, A., Alsarhan, A., Khalil, I., Sawalmeh, A., and Shamsiah Othman, N. (2017) 'Efficient 3D placement of a UAV using particle swarm optimization'. 2017 8th International Conference on Information and Communication Systems (ICICS), pp. 258-263.

[20] Singhal, G., Jain, A. and Patnaik, A. (2009) 'Parallelization of particle swarm optimization using message passing interfaces (MPIs)'. 2009 World Congress on Nature & Biologically Inspired Computing (NaBIC), Coimbatore, pp. 67-71, doi: 10.1109/NABIC.2009.5393602.

[21] Thakur, R., Gropp, W., and Toonen, B. (2005) 'Optimizing the synchronization operations in message passing interface one-sided communication'. The International Journal of High Performance Computing Applications, 19(2), pp. 119-128.

[22] Utkarsh, G., Varshney, S., Jain, A., Maheshwari, S., and Shukla, A. (2018) 'Three dimensional path planning for UAVs in dynamic environment using glow-worm swarm optimization'. Procedia computer science, 133, pp. 230-239.

[23] Venter, G., and Sobieszczanski-Sobieski, J. (2006) 'Parallel Particle Swarm Optimization Algorithm Accelerated by Asynchronous Evaluations'. Journal of Aerospace Computing Information and Communication, 3, pp. 123-137, doi: 10.2514/1.17873.

[24] Yang, M., Chen, R., Chung, I., and Wang, W. (2016) 'Particle Swarm Stepwise Algorithm (PaSS) on Multicore Hybrid CPU-GPU Clusters'. 2016 IEEE International Conference on Computer and Information Technology (CIT), Nadi, pp. 265-272, doi: 10.1109/CIT.2016.101.

[25] Zou, X., Wang, L., Tang, Y., Liu, Y., Zhan, S., and Tao, F. (2018) 'Parallel design of intelligent optimization algorithm based on FPGA'. International Journal of Advanced Manufacturing Technology, 94, pp. 3399–3412, doi: https://doi.org/10.1007/s00170-017-1447-y.