|epcc|

Performance Investigation and Parallelisation of the CosmoNest code

Ignat Tolstov

August 20, 2013

MSc in High Performance Computing

The University of Edinburgh

Year of Presentation: 2013

Abstract

High Performance Computing have been widely used in various aspects of scientific studies. Using the computing power of modern machines, it allows researchers to conduct various experiments that would be impossible to carry out in the real world, hence, one might obtain the valuable experimental data.

Commonly, it is necessary to make some scientific application, which was initially sequential, to be executable by a number of cores, i.e. to make it parallel. In this case we aim to parallelise "CosmoNest" program that is applied in the field of Cosmology. The given application tackles the issues of locating the region in the cosmological parameter space, which corresponds to the maximum value of the likelihood function. In fact, this program was an implementation of the sequential algorithm called Nested Sampling. The aim of the project was to investigate the various possible methods of parallelisation of the "CosmoNest".

We introduced several parallel strategies that were developed using the original Nested Sampling algorithm as a basis. The given strategies were implemented using OpenMP and, therefore, we conducted series of performance tests. Then, we compare the obtained performance results in order to define the best strategy.

As a result, two out of five developed parallel strategies have shown good results since we have achieved the speed-up values of 41.6 and 47.5 on 64 threads, respectively. At the same time, one strategy proved to be completely useless as its best performance result corresponded to the value of speed-up 1.1 using 8 threads. Meanwhile, the obtained results of the remaining strategies have been quite modest in comparison with the best received ones. That is, the corresponding values of speed-up were only 5.2 and 21.5 on 64 threads.

Contents

List of Tables

List of Figures

Acknowledgements

First of all, I would like to thank my main supervisor Dr. Mark Bull. Without his support and guidance, I would never have managed to finish this project. His help in in all aspects of the project was invaluable.

I would also like to thank my external supervisor Dr. Andrew Liddle for providing the code for the "CosmoNest" application and his help throughout the project. Thanks to his explanations and provided information, I was able to understand the theoretical side of the project.

Moreover, I would like to thank Dr. David Henty, who proposed the given project and introduced me and Mark to Andrew.

Finally, I would like to thank my friends and family, especially my grandfather, who does not understand the High-Performance Computing, but he motivated and inspired me at the time when I needed it.

Chapter 1

Introduction

High Performance Computing has become an essential and effective tool for many areas in modern science and industry. The wide variety of experiments, which either are impossible to carry out in the real-world conditions or too expensive to conduct, can be easily simulated on high-performance systems and supercomputers. However, such simulations require huge amounts of computation and, therefore, they impose highly demanding requirements on the performance that is provided by the machine. Hence, in order to efficiently exploit the machine resources and obtain reasonable application run times, one might require taking into account various issues while developing the application. Factors such as the amount of available parallelism that is exploited by the program and how balanced is the workload between the processors on which it runs can significantly affect the resulting performance of the application. One of most common situations when developing such applications for research purposes is when the developer's aim is to parallelise some sequential application that already has been developed in the past, e.g., developer tries to redesign provided legacy code, which is can be run on the single processor only, to make the application being runnable on several processors simultaneously.

This project is aimed at solving problem of this kind. The program that it aims to parallelise is "CosmoNest". This application calculates the likelihood ("evidence") of a given model in terms of a Bayesian framework and is intented to be used in the field of Cosmology. In fact, "CosmoNest" is actually an add-on for the "CosmoMC" application, which uses Markov-Chain Monte-Carlo (MCMC) methods for evaluation of the likelihood of the models in the cosmological parameter space. On the contrary, "CosmoNest" is based on the so-called Nested Sampling algorithm. Though this application uses completely different algorithm, it still heavily relies on the functionality of "CosmoMC" in many ways. The code was initially developed by David Parkinson, Pia Mukherjee, and Dr. Andrew R. Liddle and was courteously provided by Dr. Liddle, who participated in the project as an external supervisor. The original "CosmoNest" program was developed as a completely sequential application and takes a considerable amount of time to finish all the required calculations for the input data of interest for the research area.

Several modifications for the initial algorithm were developed during the project, each modification is aiming to exploit some amount of parallelism that is offered by the original Nested Sampling algorithm and, therefore, use this modified algorithm to implement application that will be capable to run using several units of execution (UEs).

On the preparation phase of the project it was concluded that strategies will be implemented using OpenMP parallelisation methods as the given problem area is not likely to be able to exploit the high degree of parallelism, i.e. it does not require considerable amount of processors. Hence, it can be solved using machine with Shared Memory architecture.

Furthermore, the initial investigation of the original sequential application showed that it is impossible to use developed parallel strategies with some functions of "CosmoNest". In fact, these functions were the main ones that represented the real cosmological model (i.e. the given subroutines were in charge of evaluation of the likelihood values for the points). Therefore, it became apparent that we will not be able to exploit the introduced strategies with real cosmological model. However, we have overcome the given issue by using the artificially created test cases. The parameters of these tests were configured in such a manner as to be the most similar to those used in the real model. Therefore, the performances of the developed strategies working with these "synthetic" tests are likely to be similar to their performances in the "real" conditions. Although, the given approach still does not allow us to work with real cosmological model, it allows us to vary the parameters of the used test cases, i.e. it makes them more flexible. Hence, we are capable to study the performances of the developed strategies in the different conditions.

Thus, in order to investigate which modification (parallel strategy) appears the most beneficial one for the given problem, several performance tests were conducted. As a result, the required data for the performance analysis of the each parallel strategy was obtained.

The relevant background information is given in Chapter 2. Next, we introduce the new parallel strategies in Chapter 3. Chapter 4 contains a description of the issues that were related to the implementation process. The obtained performance results for the corresponding parallel strategies are presented in the next Chapter 5. Finally, Chapter 6 contains the conclusions in which we summarise the project's results. In addition, we introduce a brief description of the problems that we faced during the project and future work, which might be carried out in order to continue the given studies.

Chapter 2

Background

2.1 Model selection in Cosmology

The object of study and research in the science of Cosmology is the universe itself, its properties and its internal processes. Consequently, the scale of the problems and challenges faced by this science requires a special approach in some aspects of research, especially in the analysis of obtained experimental data. Data analysis should be based on a complex theory that takes into account many factors in the whole universe. Moreover, one might expect the theory's complexity will rise and in order to obtain more precise and reliable results from data analysis we might need more refined inference methods [1]. Furthermore, the experimental data will have some statistical inaccuracy as, eventually, it is not possible to collect this data across all of the universe. The gathering of cosmological data requires a lot of resources and can take a long time. For instance, European Space Agency (ESA) Planck space telescope project [2], which intended to gather cosmic microwave background (CMB) information, took almost four years with the approximate cost of 515 million pounds. Hence, these factors increase the value of obtained cosmological data due to the limitation of resources. Therefore, already obtained data should be carefully evaluated using more efficient methods. In addition, one might want to concentrate on the analysis of the most plausible aspects in order to maximise the scientific feedback from the research. In fact, we have to deal with the uncertainty in theory as well as with the uncertainty in the collected data (as we cannot be sure that proposed hypothesis is perfectly correct, i.e., it either may have excessive factors, or may overlook some important ones, which are not known yet).

Thus, all these factors raise a number of issues that should be considered in the cosmological data analysis. One of the widely used techniques in Cosmology, which allows us to overcome these challenges is Bayesian inference [3]. Next, we consider the method in more detail.

2.1.1 Bayesian inference

First of all, we should specify what definition of probability is used in Bayesian inference. Probability is defined as degree of belief, i.e. *"probability is a measure of the degree of belief about a proposition"*. This definition differs from the "frequentist" one that considers probability as the frequency of occurrence of the event during the infinite number of attempts, each of which is equiprobable. In the Bayesian case, the probability is some value that reflects how strongly we believe that this this proposition is true. The Bayesian definition is better suited to the goals that are pursued in the study of Cosmology as we can not have an infinite set of data. Furthermore, unlike the "frequentist" one, this determination can be applied to a wider range of events, such as "What is the probability that the universe is made of jelly?". We denote this proposition as H and its probability as $P(H)$. The given probability expresses our subjective belief in this proposition, before we observe any new data that could either confirm or refute this hypothesis, i.e. this is "prior probability". Next, we denote a new obtained data as D, therefore, the joint probability of H and D will be $P(HD)$. According to the product rule we can express this probability as follows:

$$P(HD) = P(H|D) * P(D) = P(D|H) * P(H) \tag{2.1}$$

Hence, we can convert (2.1) into Bayes' Rule:

$$P(H|D) = \frac{P(D|H) * P(H)}{P(D)}, \tag{2.2}$$

where $P(H)$ is prior probability of hypothesis H, $P(D|H)$ is likelihood (commonly referred to as $\mathcal{L}(H)$), i.e. the possibility that new obtained data D arises from our hypothesis H, $P(D)$ is probability of data based on all possible outcomes of proposition H (H is true or H is false); it can be expressed as $P(D) = \sum_{H} P(D|H) * P(H)$ and usually called "Bayesian evidence". Finally, $P(H|D)$ is the new probability of hypothesis H considering the new gathered data D, i.e. posterior probability of the proposition. One might note that Bayes' Rule expresses how our degree of belief changes according to the light of the new observed data. In fact, hypotheses can act as different models, which will attempt to predict the data values before it is gathered. Therefore, we can mathematically evaluate how well a particular model corresponds to the data obtained by calculating its posterior probability. Thus, (2.2) can be expressed by the equation as follows (where model M serves as a hypothesis):

$$P(M|D) = \frac{P(D|M) * P(M)}{P(D)}. \tag{2.3}$$

Moreover, each model is not only trying to define how the values of data will change, but they also define a set of parameters that affect this data. The set of parameters θ

can be represented as N-dimensional vector. Hence, assuming the model M one can express (2.2) as

$$P(\theta|D, M) = \frac{P(D|\theta, M) * P(\theta|M)}{P(D|M)}. \tag{2.4}$$

In order to evaluate the model's posterior probability $P(M|D)$, we have to obtain the prior model's probability $P(M)$ and the model's likelihood $P(D|M)$ (usually referred as evidence). It is common to set the prior probabilities of different models to be equal to each other. In turn, the former value might be computed using (2.4). According to [4] evidence can be computed as following integral:

$$P(D|M) = \int P(D|\theta, M) * P(\theta|M) d\theta. \tag{2.5}$$

The higher the value of evidence $P(D|M)$ will be, the more precise and accurate the model will be. Consequently, to maximise evidence (or model's likelihood) $P(D|M)$, one might aim to maximise the parameters' likelihood $P(D|\theta, M)$ (usually denoted as $\mathcal{L}(\theta)$). Therefore, in order to create accurate model, we pursue two distinct goals: we want to define the set of parameters, which is implied by the given model, and determine the region in the parameter space, where their likelihood $\mathcal{L}(\theta)$ will be of its maximum value. Furthermore, in order to determine which of several proposed models makes a more accurate predictions, we have to calculate the integral (2.5) and compare the obtained values of evidence for the given models.

Thus, using Bayesian inference for Cosmology, we have an excellent method that can be applied to the construction of cosmological models, which are describing the underlying physical processes in the Universe. Furthermore, one might use it for model comparison and selection, and hence it will be possible to create more accurate and precise models.

In the aspect of Cosmology, we have a set of experimental data which were obtained during a series of cosmological observations, including projects such as the previously mentioned Planck space telescope [2]. Moreover, we have proposed theoretical cosmological models that describe the physical processes taking place in the universe. Each such model implies a set of parameters with each parameter having some prior distribution. We might expect that "good" predictive model will have fairly high average value of likelihood across all the parameter space. In order to verify how well the data predicted by the model are correlated with the actual data obtained during the experimental observations, we aim to study the behaviour of the model's likelihood function. In particular, we want to find the regions of parameter space, where the model's likelihood takes the maximum value. Studying the behaviour of the likelihood function on the boundaries of such regions allows further refining of the model to make it more predictive and accurate.

Hence, our challenge is based on finding the maximum of likelihood function $\mathcal{L}(\theta)$. Although the task seems quite trivial, in practise it becomes more complicated by several

issues. Foremost, situations that are of interest for studies usually have dimensions of the parameter space in the range 6-10 [3]. Evidently, increasing the number of measurements leads to the fact that the problem can not be solved analytically. Therefore, all necessary calculations are carried out via numerical methods and techniques using machine computing power. However, even the computing speed of modern machines may not be enough as the likelihood function may have several local maximas (i.e. the amount of required calculations to define the "true" global maximum may drastically rise). Furthermore, calculations can be hindered due to the fact that the function's values will vary only slightly in one or more dimensions of parameter space. In addition, evaluation of the likelihood function values itself can take quite a long time since modern cosmological models are based on the complex theory and take into account many factors [3].

Thus, all these issues make a simple "brute-force" approach impractical (as the time required to compute will go beyond reasonable limits) and challenge us to use more efficient methods. Fortunately, techniques known as Monte-Carlo methods have been proved to be suitable for solving the problems of this kind and have been widely used to find solutions. For instance, one can take Monte-Carlo Markov Chain (MCMC) methods. In general, the principle of these methods might be described as follows: we randomly sample new points from the parameter space and compute a likelihood value for each new point; based on the fact whether the new point's likelihood value is greater than the value of the previous point, we replace the "old" point that was obtained in the previous step for the new one. Moreover, we perform sampling of new points and points replacements according to the particular algorithm. Therefore, one step of this algorithm includes picking a new sampling point and replacing some "old" point with it. Thus, the sequence of such consecutive steps will be a chain. Hence, starting from some random place in the parameter space and using this approach we might expect to obtain a set of points with the maximum values of likelihood (i.e. to discover required region, where $\mathcal{L}(\theta)$ has its maximum). A number of methods and algorithms of this category have been developed over the years, for example the Metropolis-Hastings algorithm [5] [6].

However, defining such regions is not enough to solve model selection problem since we want to be able to compare developed models to determine the best (more predictive and accurate) one. Therefore, we have to calculate the evidence value for each model, i.e. we have to calculate the integral (2.5). Calculation of this integral is not trivial as we are facing the very same issues that were mentioned above (parameter space has high number of dimensions, evaluating of the likelihood function is itself computationally demanding, etc.). One of the algorithms that is intended to solve this problem is the method called Nested Sampling algorithm. This clear and elegant method was initially developed by John Skilling in 2006 [7]. We shall discuss it in the next section.

2.2 Nested Sampling algorithm

The main idea of this algorithm was presented by the author in [7]. Below we describe the key ideas of this method. We denote the model's evidence as Z, likelihood function as $\mathcal{L}(\theta)$ and the parameter prior $P(\theta|M)$ as $\pi(\theta)$, therefore, the expression (2.5) takes the following form:

$$Z = \int \mathcal{L}(\theta)\pi(\theta)d\theta. \tag{2.6}$$

Then, Skilling showed that rather than calculate a multidimensional integral, we can switch to calculating a single-dimensional one. This transformation can be achieved if we redefine $\pi(\theta)d\theta$ as dX, where we denote dX as an element of prior mass. Next, we sort the given elements in descending order of corresponding likelihood values. Therefore, we can define prior mass $X(\lambda)$ as the sum of prior mass elements, whose likelihood values exceed a certain value λ, with $X(0) = 1$ (all prior mass) and $X(\infty) = 0$. In addition, we express the function $\mathcal{L}(\theta)$ in terms of X. Hence, the integral (2.6) will be represented as:

$$Z = \int_0^1 \mathcal{L}(X)dX. \tag{2.7}$$

Consequently, this one-dimensional integral can be calculated by numerical techniques. We introduce a set of points $X_1, X_2, ..., X_n$ lying within the range (0,1) and corresponding likelihood $L_1, L_2, ..., L_n$ values. Therefore, one might represent integral (2.7) as weighted sum $Z = \sum_{i=1}^{m} L_i w_i$, where $w_i = \Delta X = X_i - X_{i+1}$. Hence, we can evaluate the integral by sampling a new point X from the given range and adding its corresponding value to the current sum. However, Skilling states that in order to cover the whole sampling range, we need to sample new points uniformly in the range $(X - X_{i-1})$ [7]. In fact, such sampling will be similar to sampling new points in the initial multidimensional parameter space, if we pick new points with likelihood value being greater than the corresponding likelihood value L_{i-1} for the X_{i-1}. Eventually, we avoid direct sampling in the "X-space", i.e. we avoid a redundant sorting step that was required for switching to the one-dimensional problem.

A comprehensive and detailed analysis of algorithm is introduced in [7]. The actual algorithm can be described as the following series of steps:

1. We start with N points $\theta_1, \theta_2, ..., \theta_N$ that were picked randomly across the parameter space with each point having corresponding likelihood value $L(\theta_i)$.
2. Set the starting values, i.e. set the evidence Z=0 and the prior mass X_0=1.
3. Next, we define the point θ_k with the lowest value of likelihood function $L(\theta_k)$ and record this value as L_i (where i is the number of current iteration).

4. Set $X_i = exp(-i/N)$ and $w_i = X_i - X_{i+1}$.
5. Increment Z by corresponding value $L_i w_i$.
6. We drop the point θ_k and sample a new random point θ' from the parameter space.
    - If the new obtained point has the value of likelihood function such that $L(\theta') > L(\theta_k)$, we will replace the "old" point with this one.
    - Otherwise, we will continue sampling until we get a new point with the requested likelihood.
7. We will repeat steps 3-6 until the growth in Z becomes less than a certain amount (e.g. some tolerance).

A diagram of the algorithm is presented below in Figure 2.1.

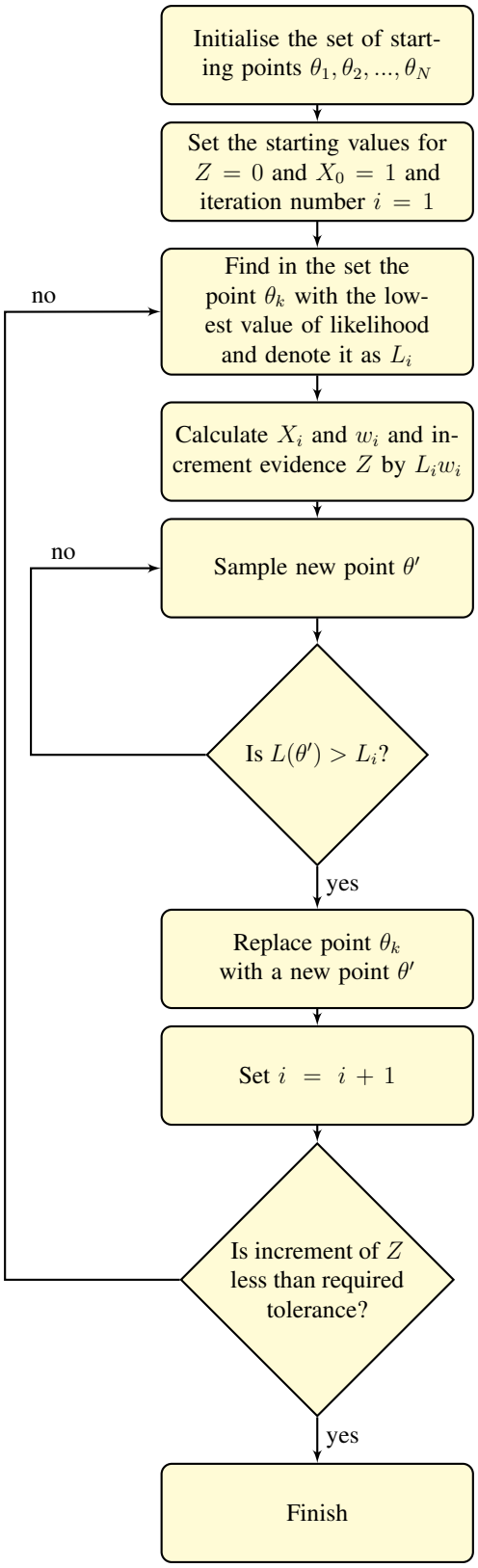

Figure 2.1: Diagram of the Nested Sampling algorithm.

Several steps of the algorithm on a two-dimensional parameter space are introduced below in Figure 2.2.

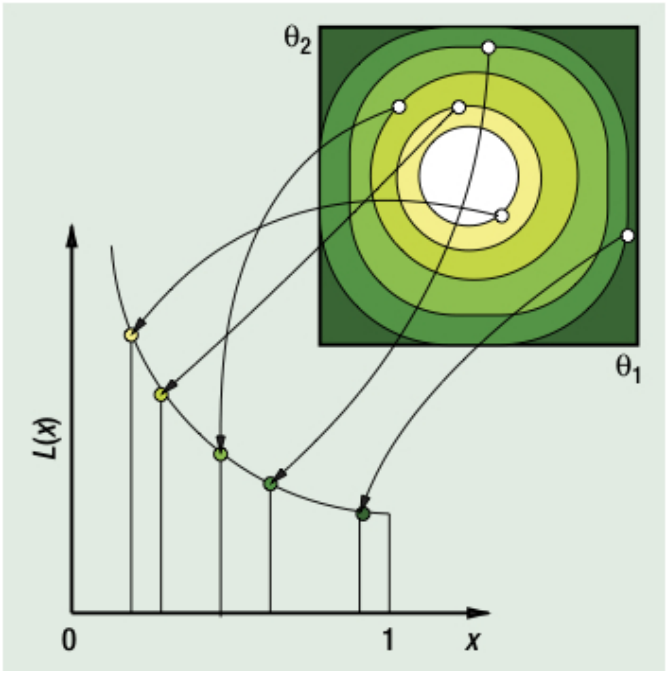

Figure 2.2: Nested Sampling algorithm on two-dimensional parameter space [8].

The prior mass "shrinks" as points in the set become localised in the region with maximum likelihood values. Moreover, we avoid direct calculation of the multidimensional integral in order to obtain model's evidence (it is computed as a one-dimensional integral using numeric methods instead).

Thus, this algorithm is capable of tackling the problem of model selection in Cosmology. Along with other algorithms that operate on this problem, the given method was implemented in actual cosmological application "CosmoNest". This project aims to study ways to parallelise this program.

Moreover, we should mention "Multinest" as an implementation of the given algorithm for use on multiple processors [9]. It is based on the modification of the original Nested Sampling algorithm. We shall consider its idea in more detail later.

2.3 "CosmoNest" and "CosmoMC" applications

The "CosmoNest" [10] application was developed by Pia Mukherjee, David Parkinson and Andrew R. Liddle and is designed to compute the evidence value for cosmological models. In fact, "CosmoNest" is actually an add-on for the "CosmoMC" application. Therefore, before we look at the "CosmoNest" program, it is necessary to consider the "CosmoMC" application.

"CosmoMC" [11] was implemented by Antony Lewis and Sarah Bridle in Fortran 2003 as a Monte-Carlo Markov Chains sampling engine. The program and its output results were described in [12]. The program is capable of using either the Metropolis algorithm or a fast-slow sampling method. In addition, one might use the given program as generic sampling engine (i.e. independently from functionality implied by cosmological theory). The resulting chains of points which the application produces are written to a text file and available to the user. The initial parameters such as various cosmological factors and options, as well as the boundaries of the parameter space are defined by the user in a special input file. In addition, the user is free to choose the method that will be used for the calculation of the evidence value (e.g. Metropolis algorithm). At the time of the work on this project this application worked with a number of libraries, including the WMAP library that contained the cosmological data gathered during Wilkinson Microwave Anisotropy Probe Project (WMAP) [13]. At present, the application is intended to use data from more recent Planck project [2]. However, since this new data came at a time when a substantial part of the work on the project has already been done, it was decided to continue to work with the old data. Furthermore, "CosmoNest" application can not work properly with this new data at this point.

"CosmoNest" adds Nested Sampling algorithm to the set of available algorithms of "CosmoMC" program. "CosmoNest" consists of a single source file. In order to integrate it into "CosmoMC", one can simply perform minor changes of the initial Makefile and "driver.F90" source file.

Although, "CosmoNest" is an implementation of the Nested Sampling algorithm, it contains some differences from the original algorithm, which have been made by its creators. These changes have been introduced due to the following issues. During the sampling points in the parameter space it becomes more and more difficult to obtain the "good" points, i.e. points whose likelihood values are greater than some current lower limit. Hence, in order to efficiently sample new "good" points the authors of the application introduced the following idea. We will sample new points based on the current amount of the prior mass, i.e. we will bound the current set of points in a certain region. This region will be represented in the form of an n-dimensional ellipse expanded by some constant enlargement factor [8]. The given ellipse will be constructed based on the point with the lowest value of likelihood, i.e. we aim to construct an n-dimensional ellipse that will have this point on its border. Consequently, new points will be sampled from this ellipse. To prevent possibly getting "stuck" due to the impossibility of finding a new "good" point in the current ellipse, we will rotate the ellipse after some number of unsuccessful attempts.

Thus, at the heart of its operation, this application uses Nested Sampling algorithm. However, sampling new points is using the new "ellipsoid" approach.

2.4 Initial run time results for the "CosmoNest"

The original "CosmoNest" was developed as a sequential program with all the corresponding work being performed by single processor. Therefore, in order to reduce the run time of the application this work should be done by several processors working simultaneously, i.e. application becomes parallelised. One might represented some sequential program as a series of procedure calls, each of which is responsible for some of the functionality and is preforming its part of the whole work. Moreover, we should consider that some of them perform major part of the work. This fact should be taken into account, since the efficiency of the the speed-up, which was obtained as a result of parallelisation, will depend on what functions have been parallelised. The greater fraction of the program's work will be divided, the more efficient will be the acceleration. Hence, it is necessary to speed up (parallelise) the part of the application, which is the most time consuming, i.e. functions of the program that require the most amount of time for their execution.

Thus, one might use a profiler program to receive required information. In this case we used "gprof" profiler with Portland Group Fortran compiler "pgf90". The initial source code for both "CosmoNest" and "CosmoMC" programs was compiled using profiling flag "-pg". The code was run using six parameters and 300 points in the initial set. The program performed 500 iterations of the algorithm. The part of the application's "call graph" that was produced by profiler is given on Figure 2.3.

First of all, it is worth noting that profiler-program ignores calls of procedures of standard libraries (e.g. built-in I/O functions like reading from file). Due to that fact the percentage of the total time, which corresponds to the basic "main" function, corresponds to nearly 50% (instead of 100%). Next, this function calls "Nestedsample" procedure that is responsible for the execution of the Nested Sampling algorithm. This function performs calls of a number of other procedures, each of which is responsible for a particular aspect of the functionality of the algorithm. The given graph clearly shows that the most time-consuming function is "GetLogLike". This function is responsible for calculating the likelihood values for the sampling points and the total time that is spent on its execution is 48,4% of 48,5%. This is part of the "CosmoMC" program and it performs a series of calls to other functions, the execution time of which makes this such a demanding procedure. Therefore, one might attempt to get "deeper" into the call graph in order to parallelise this internal functions.

```
index % time    self  children    called     name
                                             <spontaneous>
[1]     48.9    0.00   58.22                 main [1]
                0.00   58.22       1/1           MAIN_ [2]
-----------------------------------------------
                0.00   58.22       1/1           main [1]
[2]     48.9    0.00   58.22       1         MAIN_ [2]
                0.00   57.75       1/1           nested_nestsample_ [3]
                0.00    0.21       1/1           io_io_ini_load_ [104]
                0.00    0.06      21/23          inifile_ini_read_logical_ [162]
                0.00    0.06       1/1           inifile_ini_close_ [172]
                0.00    0.05       1/1           paramdef_initialize_ [178]
                0.00    0.03      11/11          inifile_ini_read_int_ [196]
                0.00    0.02       6/6           inifile_ini_read_real_ [224]
                0.00    0.01       5/5           settings_readinifilename_ [232]
                0.00    0.01       1/1           random_initrandom_ [273]
                0.00    0.01       3/3           inifile_ini_read_string_ [275]
                0.01    0.00       2/12          inifile_tnamevaluelist_init_ [185]
                0.00    0.00       5/5           inifile_ini_haskey_ [341]
                0.00    0.00       1/1           inifile_ini_read_double_ [352]
                0.00    0.00       3/3           amlutils_numcat_ [536]
                0.00    0.00       2/2           io_io_outputopenforwrite_ [543]
                0.00    0.00       2/3           io_io_close_ [537]
                0.00    0.00       2/2           amlutils_getparam_ [541]
                0.00    0.00       1/1           cmbdata_readdataset_ [553]
                0.00    0.00       1/1           cmbdata_readsztemplate_ [554]
                0.00    0.00       1/1           amlutils_setidlepriority_ [548]
                0.00    0.00       1/1           paramdef_dostop_ [556]
-----------------------------------------------
                0.00   57.75       1/1           MAIN_ [2]
[3]     48.5    0.00   57.75       1         nested_nestsample_ [3]
                0.01   47.05     501/501           nested_getnewp_box_ [6]
                0.02   10.61       1/1           nested_gen_initial_live_ [16]
                0.00    0.04    1/1632           calclike_getloglike_ [4]
                0.02    0.00     501/501         nested_calc_covmat_ [230]
                0.00    0.00     801/801         nested_plus_ [508]
                0.00    0.00     501/502         matrixutils_matrix_diagonalize_ [511]
-----------------------------------------------
                0.00    0.04    1/1632           nested_nestsample_ [3]
                0.00   10.60    300/1632         nested_gen_initial_live_ [16]
                0.00   47.03   1331/1632         nested_getnewp_box_ [6]
[4]     48.4    0.00   57.66    1632         calclike_getloglike_ [4]
                0.00   51.30   1632/1632         calclike_getloglikepost_ [5]
                0.00    6.36   1632/1632         paramstocmbparams_ [24]
-----------------------------------------------
```

Figure 2.3: Part of the original "CosmoNest" application's call graph.

However, it was decided to work with more abstract and general approach, i.e. we

aim to parallelise “Nestedsample” function itself, rather than its “child” functions. This approach will allow the possible resulting solution to be more independent from the current implementation of the “CosmoMC” program, therefore, our solution will be more portable. Furthermore, this method will not hinder the efficiency of solution’s speed-up as we are still working with the function that requires the greatest part of the overall program’s run time. In addition, in this case we will be able to study possible different parallel strategies that will exploit available parallelism, which is implied by Nested Sampling algorithm.

Thus, in the course of the project, all developed parallel strategies were based on this abstract approach. Hence, each of these strategies is a modification of the original algorithm. Each of them is introduced in more detail in the next section.

Chapter 3

Parallel Strategies

The main task when we aim to parallelise some sequential application is to find a source of available exploitable parallelism that is implied by the given problem. By this we mean that some steps of the algorithm, which is implemented by the program, can be performed using multiple performers. In order to avoid binding to a specific implementation of a parallel program, it is a common practise to consider such performers as some abstract workers (they are usually called "Units of Execution" or UEs). Moreover, as mentioned previously, it is also very important to parallelise the steps of the algorithms which correspond to the most time-consuming part of the application's functionality. In this case, the function which is responsible for the calculation of the likelihood values for new points represents such a part. This function corresponds to the step of the algorithm in which we sample a new point and the calculate likelihood value for it.

In Figure 3.1 we introduce the pseudo-code for the algorithm of the "CosmoNest" application.

```
generate initial set of N points
set initial algorithm parameters
while (evidence Z is not converged) do
    find the point with the smallest likelihood L_min
    construct ellipse
    repeat
        sample new point θ' from ellipse
        evaluate its likelihood value L(θ')
    until (L(θ') > L_min);
    increment the evidence Z by the corresponding value
    replace the point with the lowest likelihood L_min with a new point θ'
end
```

Figure 3.1: Pseudo code of the original "CosmoNest" application's algorithm.

As can be seen from the above pseudo-code, the number of calls to the "likelihood"

function will depend on how frequently we will sample a point with a higher likelihood than the current lower bound L_{min}. This is the frequency of obtaining "good" points, i.e. sampling efficiency (we denote it as s). The lower the value of s, the greater number of trials we have to perform in order to receive new "good" point. Therefore, the amount of work that is required to execute the algorithm increases as we decrease the value of sampling efficiency.

Thus, in this case a source of exploitable parallelism is the low value of sampling efficiency s. Since single UE will have to perform significant amount of work while executing the algorithm due to the small value of s, this work can be divided among several different UEs. In fact, new sampling points will be produced by a number of UEs working simultaneously on the each algorithm's iteration. We introduce several parallel strategies that were developed using this idea in the following sections.

3.1 Parallel Strategy 1

One might divide the required work, which is required to obtain a "good" point on some iteration of the algorithm, among several UEs by simply assigning a number of them to evaluate likelihood values for the several sampling points at once. Therefore, rather than of picking one sampling point at a time, we will pick a number of points instead. As a result, any of obtained points that satisfies the required condition ($L(\theta') > L_{min}$) may be used as a replacement for the "old" point.

The pseudo-code for the given parallel strategy is presented in Figure 3.2.

```
generate initial set of N points
set initial algorithm parameters
while (evidence Z is not converged) do
    find the point with the smallest likelihood L_min
    construct ellipse
    repeat
        sample P new points θ'_1, θ'_2, ..., θ'_P from ellipse
        for i = 1, 2, ..., P do in parallel
            evaluate likelihood value L(θ'_i)
        end
    until (max(L(θ'_i)) > L_min);
    increment the evidence Z by the corresponding value
    replace the point with L_min by a new point θ'_i with L(θ'_i) > L_min
end
```

Figure 3.2: Pseudo code for the first parallel strategy.

This parallel strategy was suggested in the context of "Multinest" implementation in [9]. One might define expected theoretical scalability of this parallel strategy (i.e. how

well it is expected to perform using different number of UEs relatively the performance on single UE) as follows. The number of calls to the likelihood function that make one UE will match the number of attempts to sample a new "good" point. Therefore, this number will depend on the value of s as $E_1 = \frac{1}{s}$. For instance, with sampling efficiency $s = 10\%$ we might expect that one UE will have to perform on average 10 attempts. At the same time, using the several UEs to sample a number of points at once will reduce the possibility of failure (i.e. the chance that new point will not be obtained). The total failure possibility of all P UEs will be $(1-s)^P$, therefore, the chance that at least one UE will get a "good" point will be $1-(1-s)^P$. Hence, the number of attempts that will be necessary to find a new point for P UEs that will work simultaneously can be defined as $E_P = 1/(1-(1-s)^P)$. According to this, expected theoretical speed-up can be represented as the ratio of the number of evaluations of likelihood that was performed by single UE to the average corresponding number that was made by several UEs, i.e. $S_{theor} = \frac{E_1}{E_P}$. In that, theoretical speed-up for the first parallel strategy will be

$$S_{theor} = \frac{1-(1-s)^P}{s}.$$

Thus, the theoretical speed-up is expected to be bounded above by $\frac{1}{s}$. Indeed, as we add more UEs to perform the algorithm, we increase the probability of obtaining new "good" point on each iteration, therefore, the strategy runs faster. However, the value of this probability may not exceed 1 and once it is reached, the addition of new UEs cease to be beneficial for the strategy.

3.2 Parallel Strategy 2

One might notice that we exploit only one newly obtained point on each iteration of the first parallel strategy. Although, the number of such points may be greater than one, we do not use the rest of them. Hence, we might want to use all of them on a single iteration. In that, after replacing the point with the lowest likelihood value L_{min} by a new point from the set of trial points instead of proceeding to the next iteration of the algorithm, we will continue to pick points from the trial set until it gets empty. Therefore, we will find a new point from the working set with the lowest value of likelihood (current likelihood lower boundary). Next, we remove all the "useless" trial points from the corresponding set, i.e. the points, whose likelihood values are less or equal to the current likelihood boundary (L_{min}). If the given set is not empty, we choose from it one of the remaining points and change it to the corresponding point with the likelihood value corresponding to the L_{min}. Otherwise, we simply start executing the next algorithm's iteration.

We have developed the second parallel strategy that was based on this idea. The corresponding pseudo-code for this parallel algorithm is presented in Figure 3.3.

```
generate initial set of N points
set initial algorithm parameters
while (evidence Z is not converged) do
    find the point with the smallest likelihood L_min
    construct ellipse
    repeat
        sample P new points θ'_1, θ'_2, ..., θ'_P from ellipse
        for i = 1, 2, ..., P do in parallel
            evaluate likelihood value L(θ'_i)
        end
    until (max(L(θ'_i)) > L_min);
    consider Ξ as the set of points θ'_1, θ'_2, ..., θ'_P
    repeat
        replace the point with L_min by the point from Ξ with the smallest L
        find new point with new smallest likelihood L_min
        remove from the set Ξ all points with L < L_min
        increment the evidence Z by the corresponding value
    until (Ξ is empty);
end
```

Figure 3.3: Pseudo code for the second parallel strategy.

Thus, some iteration of the given parallel algorithm might include the replacement of several points instead of one. This situation is possible if during the points swapping we replace the point from the working set that has not yet been replaced. In that, new trial point, which was used for the very first exchange, will not become the new point with the lowest likelihood value. Therefore, since this parallel strategy intend to exploit as many trial points as it can, its possible that theoretical speed-up might exceed the limiting factor $\frac{1}{s}$. However, one might expect that such iterations, which imply replacing several different points from the working set, will be relatively rare. In fact, they will not substantially occur when using a small number of UEs. Furthermore, since all the new points that are exploited during some algorithm's iteration were obtained from the same ellipse, we might face a sampling bias issue. Indeed, if the sampling efficiency s is sufficiently large and, therefore, a large portion of new points, which we use for replacement, will be picked from ellipse, then our current working area in parameter space may be captured inside this ellipse. Hence, using this parallel strategy may be feasible for problems that might have several local likelihood maximums and high value of sampling efficiency s.

3.3 Parallel Strategy 3

We introduce another parallel strategy that is based on a different approach. Rather than to work with one point on each iteration (i.e. construct ellipse that is based on this point and try to replace it by a new point, which is picked from this ellipse), one might work with several points in each step of the algorithm. Based on how many UEs are used with this parallel strategy, we can try to replace the corresponding number of points at each iteration. Each UE will perform the required sequence of steps (creating ellipse for sampling, picking new trial point from it, etc.) for its own point from the working set. We shall consider the proposed method with the following example. Let us have initial working set that contains N points. Some points $y_1, y_2, ..., y_P$ are part of this set and have the values of likelihood such that $L(y_1) < L(y_2) < ... < L(y_P)$ with $L(y_1)$ being the lowest likelihood value L_{min}. Therefore, operating in turn with each of these points, the original Nested Sampling algorithm will perform first P iterations. Although, it is worth noting that the exact order of points processing (i.e. first y_1, then y_2 and so on) will be possible under certain conditions. In fact, it is necessary that after we replace the point y_1 by some new point θ_1', the point y_2 would be the point that has the lowest value of likelihood L_{min} at the moment. Hence, this condition can be described as follows. The point y_1 should be replaced by new point θ_1' if the usual requirement $L(\theta_1') > L(y_1)$ is met. However, some point y_i, whose index i greater than 1, should be replaced by new point θ_i' if the following condition is satisfied: $L(\theta_i') > L(y_{i+1})$.

That is

$$\begin{cases} L(\theta_i') > L(y_{i+1}) & \text{for } i = 1, ..., P-1 \\ L(\theta_P') > L(y_P). \end{cases} \tag{3.1}$$

Thus, original sequential algorithm will perform the first P iterations in the given order. Therefore, we can introduce "lookahead". Assuming that the above-mentioned condition is satisfied we can assign several UEs to simultaneously work with the given points $y_1, y_2, ..., y_P$ on the same algorithm step. The corresponding pseudo-code for this parallel strategy is presented in Figure 3.4.

Thus, at each iteration of the given parallel algorithm, P UEs perform a replacement of P points. Therefore, they will execute the number of steps, which is required to obtain the final result, P times faster than some single UE that will work based on this algorithm.

However, such replacement will be possible only if the previously mentioned condition (3.1) is satisfied. Hence, the resulting theoretical speed-up will depend on the possibility of satisfying this condition. Let us assume that the likelihood value of a new sampling point is equally likely to lie in any of intervals

$$[L(y_1), L(y_2)], [L(y_2), L(y_3)], ..., [L(y_N), \infty].$$

As a result, the probability that the likelihood of the first sampling point θ_1' will be

```
generate initial set of N points
set initial algorithm parameters
while (evidence Z is not converged) do
    find P points with the smallest likelihood values
    L_min = L(y_1) < L(y_2) < ... < L(y_P)
    construct ellipses E_1, E_2, ..., E_P based on these points
    for i = 1, 2, ..., P do
        sample new point θ'_i from the corresponding ellipse E_i
    end
    for i = 1, 2, ..., P do in parallel
        evaluate likelihood value L(θ'_i)
    end
    if (L(θ'_1) > L(y_1)) then
        replace y_1 with θ'_1
        increment the evidence Z by the corresponding value
    end
    for i = 2, ..., P do
        if (L(θ'_{i-1}) > L(y_i)) and (L(θ'_i) > L(y_i)) then
            replace y_i with θ'_i
            increment the evidence Z by the corresponding value
        else
            break
        end
    end
end
```

Figure 3.4: Pseudo code for the third parallel strategy.

greater than $L(y_2)$ will be $\frac{N-1}{N}$. The corresponding value of probability for the next point θ'_2 will be $Prob(L(\theta'_2) > L(y_3)) = \frac{N-2}{N-1}$ and so on. Therefore, the above condition is satisfied with probability equal to the product of all these probabilities (as it is necessary that the corresponding conditions for each point were performed simultaneously). That is the resulting probability will be $\frac{N-1}{N} \times \frac{N-2}{N-1} \times ... \times \frac{N-P}{N-(P-1)} = \frac{N-P}{N}$. Thus, the theoretical speed-up for this parallel strategy will be $P \times \frac{N-P}{N} = P - \frac{P^2}{N}$.

Moreover, we might expect theoretical speed-up reaches this value, if only the sampling efficiency of new points s will have its maximum value, i.e. $s = 1$. We proceed from the assumption that the new sampling points $\theta'_1, \theta'_2, ..., \theta'_P$ have to be "good" and we do not consider the situation when in the course of the strategy, these points will not be obtained because of the small sampling efficiency value. In fact, in real problem cases the value of s will be much less than 1, therefore, this strategy will have little practical use when dealing with them. However, we can use it as a basis for more complex strategy, described in the following section.

3.4 Parallel Strategy 4

The following parallel strategy that we introduce uses the idea, which we exploited as the basis for the previous one. We aim to replace several points on each step of the algorithm in parallel using a number of UEs. In order to preserve "sequential" order of points processing (i.e. points from the working set are replaced as if they were processed by a single UE), it is necessary to satisfy the previously mentioned condition (3.1). Parallel Strategy 3 does not consider the assumption that sampling efficiency s may differ from its maximum value. We shall consider more realistic cases with $s < 1$. As the value of sampling efficiency will be small the UEs that will sample new points from the corresponding ellipses will mostly fail in receiving "good" points (i.e. the points with the likelihood values exceeding corresponding lower limits). Hence, their work efficiency will be insignificant and the resulting theoretical speed-up of the strategy will be low.

However, one might overcome this issue by assigning each UE to continuously sample new points until it manages to obtain acceptable one. Although, this approach will solve the given issue, it will introduce new overheads that are associated with load imbalance. That is, some UEs will have to perform greater number of sampling trials than the others to obtain an acceptable point. While these UEs will continue sampling process, UEs that have received "good" points will be idle. Therefore, the overall execution time of one iteration of the algorithm will not be faster than the execution time of the slowest UE (i.e. the least fortunate one). One might want to balance the workload among UEs by introducing some kind of scheduling, which will guide the UEs that have completed sampling in the corresponding ellipses, in choosing which of the remaining ellipses they will process next. Since initially we can not determine how much work a particular UE will have to perform via processing corresponding ellipse (i.e. we do not know how many sampling attempts it will take in order to obtain satisfying point), it is impossible to define which UEs will complete their sampling first. Thus, using the static schedules (i.e. schedules that try to assign some pre-fixed amount of work to each UE) will be pointless. Hence, the workload schedule for the given case should be a dynamic one. We will assign "free" UEs as they come to the processing of remaining ellipses cyclically. We show this in the following example.

For instance, the given parallel strategy is executed by five UEs. Therefore, in the beginning of each algorithm's iteration we will have five ellipses in the trial set with each UE sampling points from the corresponding ellipse. At some point, three out of five UEs succeed to obtain "good" points, hence, they will remove the corresponding ellipses from the set. Next, we assign the first of them to the first of the two ellipses, which remain in the set. The second UE will process the corresponding second ellipse and the last "free" UE will be assigned to the first ellipse again.

In fact, the number of attempts that is required to receive the "good" point, depend on the value of the corresponding probability. Therefore, similar to the first parallel strategy we increase this value with the number of UEs that process one ellipse (as we add "free" UEs to perform sampling along with the "busy" UE, which processed this

ellipse before). As a result, they will receive the required point faster by simultaneous sampling.

The corresponding pseudo-code for this parallel strategy is presented in Figure 3.5.

```
generate initial set of N points
set initial algorithm parameters
while (evidence Z is not converged) do
    find P points with the smallest likelihood values
    L_min = L(y_1) < L(y_2) < ... < L(y_P)
    construct ellipses E_1, E_2, ..., E_P based on these points
    let the set of these ellipses be H = {E_1, E_2, ..., E_P}
    while (H ≠ ∅) do
        for i = 1, 2, ..., P do
            sample new point θ'_i from some ellipse E_j ∈ H
        end
        for i = 1, 2, ..., P do in parallel
            evaluate likelihood value L(θ'_i)
        end
        for i = 1, 2, ..., P do
            if (L(θ'_i) > L(y_i) and (θ'_i come from E_j)) then
                assign z_j = θ'_i
                remove E_j from the set H
            end
        end
    end
    if (L(z_1) > L(y_1)) then
        replace y_1 with z_1
        increment the evidence Z by the corresponding value
    end
    for i = 2, ..., P do
        if (L(z_{i-1}) > L(y_i)) and (L(z_i) > L(y_i)) then
            replace y_i with z_i
            increment the evidence Z by the corresponding value
        else
            break
        end
    end
end
```

Figure 3.5: Pseudo code for the fourth parallel strategy.

Thus, as a result of each iteration of the given parallel algorithm P UEs perform a replacement of P points from the working set. The stopping criteria in the given pseudo code is based on the fact that the condition (3.1) must be satisfied. In fact, by analogy with the Strategy 3 we aim to preserve the flow of points processing in parallel similar to the flow, which is introduced by using a single UE.

3.5 Parallel Strategy 5

The following parallel strategy uses rather different approach than the previous ones. Instead of trying to parallelise some part of the algorithm by exploiting parallelism that is available on its iteration, one might consider the problem on a more abstract and global level. That is, try to parallelise the whole Nested Sampling algorithm. The order of the original algorithm's work can be represented as follows. Initially we start with the working set that contains a number of points, which were randomly sampled from the given parameter space. Next, we sample new candidate points and evaluate corresponding values of likelihood until we obtain the one that will satisfy the current required condition, i.e. the point, which likelihood value is greater than the current lower bound. In this case we replace the point with the lowest value of likelihood with this newly obtained point and continue the sampling process. Hence, the original sequential algorithm's work can be represented as continuous testing of candidate points and the use of appropriate ones as a replacement for the points from the work set. Moreover, since we exploit "ellipsoidal" sampling in the algorithm to pick new points from the parameter space, all the candidate points are sampled from some current ellipse that is constructed basing on the point, which we aim to replace. As we showed in the previously introduced parallel strategies, we can assign several UEs to generate new candidate points on each algorithm iteration in order to obtain "good" points faster. However, such simultaneous parallel sampling was limited to one algorithm iteration. That is, each UE started the sampling process and finished it according to execution of the current iteration and was not allowed to start the sampling process again until the next iteration will start. Such approach will bound the total amount of available exploitable parallelism as we will be able to use only the parallelism that is available to us during the iteration of the algorithm. Therefore, in order to overcome this limitation we shall introduce the asynchronous sampling of the candidate points. The sampling process and corresponding likelihood evaluation calculations will be performed by a number of UEs, each of which will work autonomously and independently of the other. The generated candidate points will be stored in some shared set. Some single UE will be in charge of validation of this set, i.e. it will continuously pick new points from the candidate set and compare them with the current point with the lowest value of likelihood. In addition, this UE will maintain the set of working points and the global ellipse that will be used by other UEs for sampling new points.

The described scheme of work corresponds to the widely known parallel design pattern, which is called "Master/Worker" and described in detail in [14]. A number of Workers

perform a series of tasks with each worker executing its individual task (in this case we consider UEs that perform sampling of new points and evaluating their likelihoods values as Workers). Furthermore, each task should be independent from other tasks (this condition is satisfied in the given case as there are no dependencies between different candidate points). At the same time, the Master will perform sending of data to Workers for further processing and gathering the output results of their tasks (in this case the UE, which is in charge of validation of candidate points, plays the role of Master). In fact, usually in the given parallel design pattern Master also responsible for the generation of tasks for Workers. However, in this case this is unnecessary as Worker's task is independent from any input data and remains the same throughout the work (sampling new points and evaluate corresponding likelihood values).

The cycle of work of this parallel strategy can be represented as follows. The Master starts its work and generates the initial points for the working set. In addition, it constructs the ellipse based on the point with the lowest value of likelihood and starts the operating cycle of Workers. Each Worker autonomously generates candidate points and calculates their likelihood values. If the obtained point has the required value of likelihood (i.e. greater than the current lower boundary), the Worker will add this point to the corresponding set. Otherwise, it will "dump" this point and continue the sampling process. At the same time, the Master checks whether this set is empty. If the set of candidate points is empty, the Master will continue to wait until the Workers generate new points. In the other case, the Master will pick a point from the candidate set and use it as a replacement for the corresponding point in the working set. Next, the Master will find the new point with the lowest value of likelihood from the working set. Then, it will construct a new ellipse for sampling and set a new likelihood lower boundary for "good" candidate points. Since the candidate set of points might contain some points that no longer satisfy the new condition, the Master will have to remove all such point before picking a new point from this set. The Master continues to work until the corresponding value of evidence Z becomes converged. In this case the Master stops all the Workers and the algorithm finishes.

The pseudo-code for this parallel strategy is presented in Figure 3.6.

It should be noted that the amount of time spent by the Master to process candidate points, which have been added to the corresponding set by Workers, depends on the number of that points. Therefore, if the time required to sample points and calculate their likelihood values will be relatively small, we may face a situation in which the number of candidate points generated by the Workers will be too large. The Master will not be able to cope with the "stream" of new points, hence, the number of points in the candidate set will increase, thereby increasing the Master's run time. Thus, we will face work imbalance between Master and Workers as the work of the latter will increase the Master's workload. This result is highly undesirable, since theoretical speed-up that will be implied by this parallel algorithm will be insignificant. Moreover, this algorithm may take even longer time to execute than the original Nested Sampling algorithm.

However, in cases of interest Worker's task is expected to take considerable amount of time according to the profiling results, presented earlier in Figure 2.3. Hence, one

Master:

generate initial set of N points
set initial algorithm parameters
initialise set of candidate points $\Theta = \varnothing$
find θ with the lowest likelihood value L_{min}
construct ellipse E based on θ
start Workers
while *(evidence Z is not converged)* **do**
    if *($\Theta \neq \varnothing$)* **then**
        remove from Θ all points with likelihood < L_{min}
        replace θ with new point from Θ
        increment the evidence Z by the corresponding value
        find new θ with the lowest likelihood value L_{min}
        construct new ellipse E based on θ
    end
    stop Workers
end

Worker:

set local ellipse $\varepsilon = E$
while do
    pick new point θ' from ε
    evaluate $L(\theta')$
    if *($L(\theta') > L_{min}$)* **then**
        add θ' to Θ
    end
    set local ellipse $\varepsilon = E$
end

Figure 3.6: Pseudo code for the fifth parallel strategy.

might expect that the rate of generation of new points will be low and the number of elements in the corresponding candidate set to be small. In fact, the Master's rate of "consuming" of new points is likely to be fast relative to the pace of points generation, therefore, candidate set will be empty most of the time.

Thus, in the given case we expect that application of this parallel strategy will provide a significant speed-up, although, the issues described above should be taken into account as well.

3.6 Modified Strategies

In the original "CosmoNest" application the time cost, which is required for likelihood evaluation for some point, is not a constant value. In fact, it can be represented as some variable having a normal distribution. That is, the likelihood calculations for each point will require different time periods and hence UEs will spend different amounts of time in order to carry out the corresponding calculations (i.e. unbalanced time costs).

Therefore, the workload balance between the UEs will be broken. Since the fifth strategy is the only one that implies methods for workload managing, the overall performance of the other strategies could be significantly hindered.

Thus, it was decided to add such methods for them. In the case when each UE conducts the required calculations for one point, it is impossible to manage the workload between UEs (as amount of work that corresponds to evaluations for a single point cannot be divided among them). Therefore, we artificially increase the amount of work for each UE by assigning each of them to evaluate likelihoods for k points instead of one. That is, in each parallel strategy UEs will calculate the likelihood values for $k \times P$ points on the corresponding stage. Hence, the number of generated points will be increased. Furthermore, we will be able to manage the workload between UEs by dynamic scheduling, i.e. UEs that already have finished evaluations for the corresponding k points will be assigned to help the other ones, which are still "busy". Hence, the average amounts of work that are executed by UEs are likely to be even. As a result, the given strategies will be capable to "resist" the natural imbalance of the problem (i.e. a significant difference between the amounts of work required to process different points). In addition, we increase the fraction of the algorithm, which is performed in parallel, therefore, the overall speed-ups of strategies will be improved.

However, since in the case of Strategy 5, UEs generate new points asynchronously, unbalanced time costs will not hinder its efficiency, hence it was decided not to modify it.

Chapter 4

Implementation

4.1 Choice of architecture for the implementation of parallel strategies

After parallel strategies have been developed, the next step was to choose a platform for their implementation, i.e. we had to choose what type of architecture of the parallel system we will use to run the implementations of the corresponding strategies. Selection of a specific parallel system architecture affects not only the performance of a parallel application, but also on the methods and tools that are available to the programmer in its development. In general, existing parallel systems can be divided into two basic types by the memory type: machines with Shared Memory (SM architecture) and machines with Distributed Memory (DM architecture).

SM architecture can be defined as follows. Processors in these systems consider the machine's memory as a global memory store. That is, each processor is capable of accessing any memory location, even if the memory may not be physically global, but can be connected through some kind of interconnect mechanism. Thus, due to the interconnect logic physically separated system's memory will be considered as some single object by the system's processors. Communication between different processors is through read / write operations of shared memory blocks. Moreover, the management of such a machine requires single operating system. However, the number of processors in such systems is quite modest due to the hardware limitations (as we increase the number of processors, the characteristics of hardware such as memory bandwidth becomes a bottleneck).

A schematic representation of one example of SM system (in this case the system's memory is physically whole) is shown in Figure 4.1.

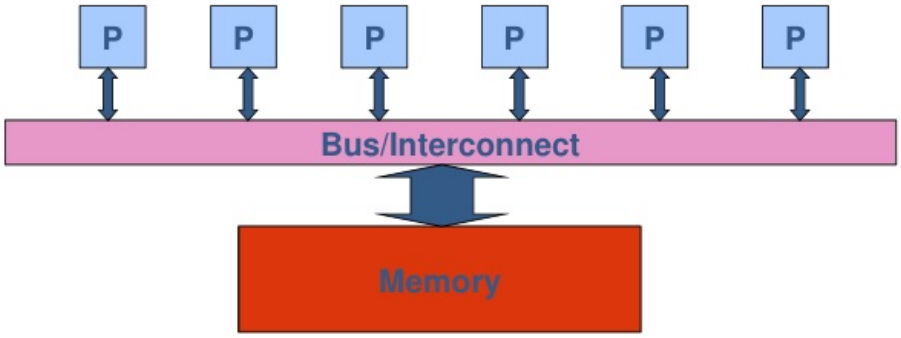

Figure 4.1: Schematic representation of one example of system with Shared Memory architecture[15].

Parallel systems that use a DM architecture, use a different approach and can be described as follows. Each processor has its own private memory that is inaccessible by other processors. All processors are connected with each other by some interconnect mechanism. Therefore, communications between different processors are carried out via explicit data sending and receiving among processors. This "Distributed Memory approach" allows the parallel systems to be highly scalable as in order to rise the number of processors of the system, one might need simply connect additional processors into the interconnect. Hence, such parallel systems can have a significant number of cores, although due to the interconnect characteristics explicit communications in such systems are usually slower than communications in SM systems. Thus, current parallel machines with a high number of processors use mixed DM-SM approach (i.e. "Cluster" architecture). Such machine can described as a number of nodes that are connected with interconnect (DM approach) with each node being built using SM approach.

A schematic representation of such system is shown in Figure 4.2.

In this case, we choose the platform based on the fact how many processors we will need to perform the given parallel strategies. In general, these strategies are some modifications of the original algorithm, which is used in the original "CosmoNest" application. That is, they aim to tackle the same problems with the same problem's sizes as the sequential one does. Moreover, in all these strategies, UEs process the set of points in parallel via simultaneous points sampling and likelihood evaluation, therefore, the number of UEs will depend on the size of this working set of points. Since the original program solves the problems, in which the size of the given set is relatively small (i.e. about 300-500 points), one might expect that execution of the corresponding implemented parallel strategies can only exploit fairly modest number of UEs. Thus, assuming 1:1 mapping of UEs to processors, the choice was made for machine with Shared Memory architecture. Based on this selection, it was decided to use OpenMP as a main implementation tool.

OpenMP is an Application Programming Interface (API) that introduces opportunities to parallelise the original sequential source code via adding parallelisation directives to it. The functionality and syntax of these directives are defined in the current OpenMP

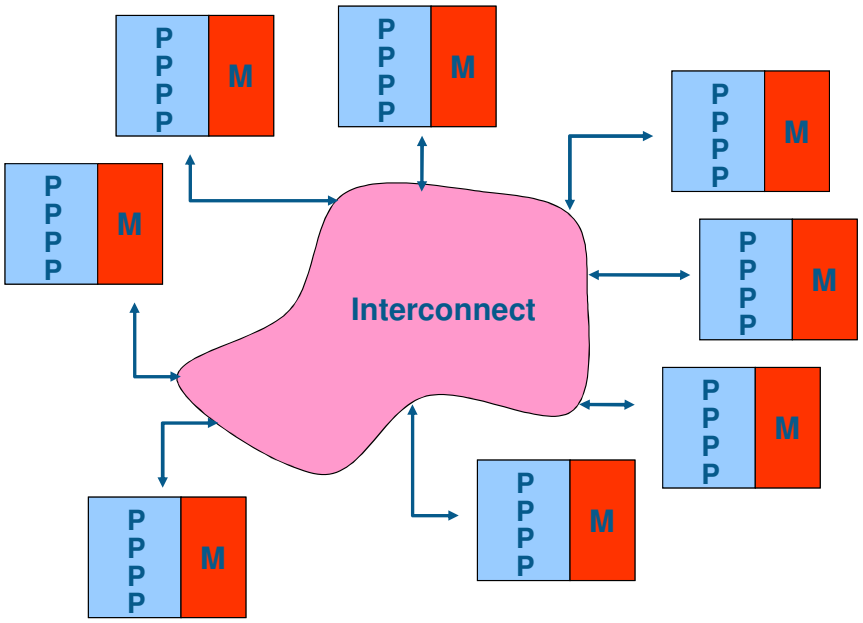

Figure 4.2: Schematic representation of system with “Cluster” architecture [15].

standard [16]. The code segments, to which we add the given directives, are treated by the compiler as parallel regions, i.e. these parts of code will be executed by the several processing elements in parallel. OpenMP introduces definition of “thread” for such an element. Hence an application that is parallelised using OpenMP will be executed by a single thread (master-thread) until the moment when it gets to the parallel region. In this case, master-thread will “spawn” a number of threads, which will execute the corresponding parallel region. After this code segment will be finished, master-thread will continue execution of the program with all other threads being idle until the next parallel region (i.e. a “fork-join” model).

In order to obtain necessary performance results for the further analysis, it was decided to choose MORAR system as a target machine for conducting the runs of the implemented parallel strategies. This machine is a computing cluster, which consists of two Shared Memory nodes. Therefore, for our purposes it is appropriate to use one of these nodes. Each node contains a total of 64 cores and the architecture of these nodes can be described more accurate as “Cache coherent Non-Unified Memory Access” (cc-NUMA node). That is, the cost for accessing some memory location will not be the same for all the cores as the memory is physically divided into several sections. Hence, some cores will be capable to access particular sections faster than others. Since this may have an effect on performance results of the parallel applications, one should consider this feature of the system architecture. Each of MORAR’s node consists of four 16-core 2.1GHz AMD Interlagos processors with 2 GB of RAM each. Furthermore, we will use the Portland Group Compiler for Fortran pgf90, which supports OpenMP.

4.2 Implementation of test cases

4.2.1 Correctness tests

In order to verify the correctness of implemented parallel strategies, we can check the output results of their work in some special cases when the correct analytical answer is known in advance. That is, when the behaviour of the function is known in advance, we can determine whether some strategy found its region of the function's maximum correctly. Therefore, we chose a trivial case of finding the maximum of a likelihood function that has a Gaussian shape to implement the required correctness test. The search is carried out in the N-dimensional cube $[-\frac{1}{2}; \frac{1}{2}]^N$. Hence, we used the following equation for the likelihood function:

$$L(\theta) = \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi}u} exp\Big(- \frac{\theta_i^2}{2u^2}\Big),$$

where u=0.01. Since the Gaussian function has unique global maximum, it will not be difficult to determine whether one or the other strategy works correctly.

4.2.2 Thread-safety tests

We have to check whether the the original "CosmoNest" code's functions and procedures, which are to be used for the implementation of parallel strategies, were thread-safe. That is, it was necessary to ensure that simultaneous execution of these procedures by multiple processors will give the same output results as if the given procedures would be performed using a single processor. Since simultaneous evaluation of likelihood values of points by multiple UEs is the foundation for all introduced parallel strategies, it was necessary to test the corresponding function that calculates likelihood values of some point in the original application.

Hence, for this was implemented following a simple thread-safety test. The likelihood evaluations, which are performed in the beginning of the Nested Sampling algorithm for the points from the initial working set, were performed sequentially (using a single core) and in parallel (using a number of cores). Then, the obtained results were compared.

Unfortunately, the results of this test showed that the given likelihood function in "CosmoNest" is not thread-safe. The reason due to which there is such an effect is the emergence of race conditions. That is, the source code that represents the implementation of the likelihood evaluation contains a number of global variables. These variables may be accessed from any source file, i.e. they are included in the application's global scope. In the case of running the code sequentially, these variables are modified by a single thread. However, when we run the given program on several threads, by default these variables are considered to be shared ones, i.e. all threads work with the same

instance of each of these variables. Hence, each such variable may be modified by any thread at any moment. Since we can not guarantee that all threads execute program with the same rate, some of the threads may be "faster" than others and, therefore, they will modify the values of the global shared variables sooner. As a result, the behaviour of the "slower" threads will change according to that fact. Subsequently, they will produce the different output results. Thus, the thread's behaviour might or might not be different during each program's run depending on its "speed", i.e. we face the race conditions.

In order to overcome the given issues, we shall assign a private copy of each global variable for each thread. As a result, any thread will be able to work independently from others. Therefore, attempts were made to make it so by introducing OpenMP directives. We surrounded the functions that were involved in calculating the likelihood values by special critical OpenMP regions. Such a region can be executed only by one thread at a time. Then, we run the thread-safety test. If this test passed, we would reduce the critical region and run the test again. In the case when this test failed, we investigated the code outside the critical region in order to find the global variables and make them private for each thread. Hence, we get "deeper" into the call graph of this likelihood function and applied the described method for its subroutines. Thus, we intended to get rid of the critical regions and make the likelihood function thread-safe.

However, these attempts were unsuccessful. The given likelihood function is in fact the part of the "CosmoMC" application and consists of high number of its source files. Since we were quite unfamiliar with the source code of this program, the searching of global variables was quite a difficult task. Moreover, race conditions imply the inconsistent application's behaviour. Hence, one might run the given thread-safety test a thousand times and it will be completed successfully and it might fail for the next time. Furthermore, this issue is compounded by the fact that the initial points are generated randomly, therefore, test situations, which expose these bugs, are even harder to reproduce. As a result, searching for program's bugs was cumbersome and time consuming.

Thus, we have spent a significant amount of effort to solve the given problem, but the solution has not been found. It was therefore decided to use a different approach. Instead of using the original likelihood function, we could use a function that has been implemented for the correctness test. In addition, the execution time of this function can be artificially increased to match the corresponding time of the original function. Moreover, we can configure the values of various parameters (e.g. sampling efficiency s) in order to make the test case's conditions close to those on which the original "CosmoNest" works. Hence, we will obtain a performance model of the original application. Since this model will be based on the correctness test, which uses different likelihood function, we can apply the introduced parallel strategies for this test case. Therefore, we will be able to investigate the performance of the given parallel strategies for the original "CosmoNest" program.

4.3 Implementation of parallel strategies

4.3.1 The main functions of the original "CosmoNest" application

The application's main file, which contains the basic "main" function, is the **driver.f90**. The code in this file performs reading of all initial parameters specified by the user, and then it runs the algorithm selected by the user. The source code that represents the implementation of the original Nested Sampling algorithm is presented in the **nested.f90** file.

The general procedure that is responsible for the implementation of this algorithm is `Nestsample`. At the beginning of its work, the procedure calls the subroutine `geninitiallive`, which performs the initialisation of the working set and computes likelihood values for points included in the set. Next, the main working loop of the `Nestsample` procedure performs the corresponding iterations of the original algorithm. In the beginning of each iteration the ellipse, which is used for sampling is constructed. Then, procedure `getnewpbox` is called. Its body contains an infinite loop that executes the required process of sampling a new point and evaluates likelihood value for it. We exit the given loop in the case that a new "good" point is obtained. If 50 attempts to get the point fail, we "rotate" the ellipse, in which we sample points, and the process is repeated for the next 50 iterations. Subroutine `getnewpbox` finishes its work after performing corresponding replacement. Next, control returns to the `Nestsample` procedure. It increments the value of evidence Z and the loop continues. This procedure completes its work if the specified number of iterations has been executed or executes one of the following stopping criteria is satisfied. That is, increase in the evidence value is less than a certain threshold, or this value becomes close enough to the required analytical one.

The pseudo code for the original "CosmoNest" application is presented in Figure (4.3), below.

Thus, during the implementation of the parallel strategies, the modifications were mainly made for the `getnewpbox` procedure. Moreover, changes were made to the procedure `Nestsample`, although it was not modified as much. In the next few subsections, we briefly describe the implementation of each of the parallel strategies.

```
Procedure Nestsample()
    // generate initial set of N points Θ
    call geninitiallive(Θ)

    // set initial values
    Z = 0 // initial evidence
    X = 1 // prior mass
    i =1 // current iteration number

    // Start main loop
    while (i <= maximum iterations number) do
        i = i + 1
        Find the point θ_k with the lowest likelihood value L_min
        Construct sampling ellipse E
        call getnewpbox(Θ, E, θ_k)
        Increment the evidence Z by the corresponding value
        if ( | Z - Z_analytical | < tolerance) then
            break // Z is close enough to analytical value
        if ( | Z_new - Z_old | < threshold) then
            break // the increase of Z is less than some threshold
    end
    return
.....
Procedure getnewpbox(Θ, E, θ_k)
    // set the number of sampling attempts to 0
    num = 0

    // Start main loop
     while do
        Sample new point θ' from the ellipse E
        evaluate L(θ')
        num = num + 1
        if (L(θ') > L(θ_k)) then
            replace θ_k with θ'
            return
        end
        if (num > 50) then // 50 attempts fail
            Rotate the ellipse E
            num = 0
        end
    end
    return
```

Figure 4.3: Pseudo code for the "CosmoNest" program.

We should note that, for all parallel strategies, except the 5th one, the point sampling (generation) were carried out sequentially, in order to avoid possible issues that may be implied by random numbers generation in parallel. In fact, the threads perform simultaneous evaluation of the likelihood values for the generated points for Strategies 1-4. In the case of Strategy 5, parallel random numbers generation is required by its idea. Hence, each thread uses its own rank as the random seed for the point generation. Although, this approach might arise possible issues when we work with real data, we use it for the performance tests and, therefore, they do not affect us.

4.3.2 Implementation of Strategy 1

In order to perform simultaneous points sampling, we added OpenMP parallel directives to the `getnewpbox` procedure. Hence, rather than use only one point, instead we can use P points. Although, they were generated sequentially, their likelihood values were calculated by P threads working in parallel. These new generated points are stored in the additional array. If at least one point from this array satisfies our requirements (i.e. its likelihood value is greater than the current likelihood lower bound), the sampling process is terminated and we perform points replacement. Otherwise, we generate another P new points sequentially and evaluate their values in parallel using P threads. Each thread is given 50 attempts to obtain a "good" point before the sampling ellipse will be "rotated" (i.e. $50 \times P$ attempts in total). In general, the implementation of the given strategy required minimal code changes.

4.3.3 Implementation of Strategy 2

This strategy has been implemented in the same way as the previous one, with the only difference being that function has been added which tried to use the remaining candidate points in the array. This function is called when a good point is found and we perform the corresponding point "swapping". It returns the number of replacements that were made, which is necessary to increase the value of evidence Z. The other changes are similar to the changes that were made during the implementation of the first strategy.

4.3.4 Implementation of Strategy 3

Since in this parallel strategy we use a number of ellipses to sample new points from, the additional array that contains a set of ellipses was introduced. After each of these ellipses was constructed we perform points sampling (with each new point being picked from the corresponding ellipse). Next, likelihood values of these points are evaluated by several threads in parallel. Moreover, we introduced the check, which is an implementation of condition (3.1). If it is satisfied, we conduct the corresponding point replacements. Otherwise, we continue the sampling process until we reach the mark of

$50 \times P$ attempts. In this case we "rotate" every ellipse in the set and start all over again. As this parallel strategy heavily relies on the unrealistic values of sampling efficiency ($s = 1$), its implementation was considered as an intermediate stage in developing of the next strategy.

4.3.5 Implementation of Strategy 4

This parallel strategy implied balancing the workload between different threads. Therefore, we introduced a number of functions for its support, including subroutine that is responsible for assigning threads to the different ellipses in the set. Moreover, in order to support removing ellipses from it, this set was reimplemented using a linked list structure (as it offers more agile and efficient methods for manipulating its data). Furthermore, we implemented corresponding functions to manipulate the list (e.g. list initialisation, removing elements from the list, etc.). We used the modifications of the third strategy for all other aspects.

4.3.6 Implementation of Strategy 5

The implementation of this parallel strategy demanded the most significant changes in the original source code. We introduced parallel region that divided threads into Master and Workers based on their ranks. Therefore, we implemented two different procedures, which represented the working cycles for the corresponding roles.

Each worker started its cycle and finished it depending on the Master's command (i.e. the particular value of global boolean flag). We introduced a global linked list of points, which represented the global set of candidate points that will be filled by the Workers. Worker thread will add the the candidate point into the set, if its likelihood value is greater than the current global lower bound of likelihood. Adding points in the list have been implemented using the OpenMP critical directive. The critical segment of code will be performed by a single thread at a time (otherwise, the list might become corrupted due to the simultaneous attempts of adding new points).

At the same time, Master thread processed the given set and performed the corresponding points replacement. In the case of points "swapping", the Master thread finds the new point with the lowest likelihood value and sets it as a new global lower bound. By analogy with the previous strategies, the Worker threads have $50 \times P$ attempts to find a new "good" point. If they do not succeed, then the Master will "rotate" the sampling ellipse.

Thus, each Worker asynchronously generates new candidate points and adds them into the corresponding set with Master thread continuously picking new points from it.

4.3.7 Implementation of Modified Parallel Strategies

As mentioned previously, in "CosmoNest" each point may require different amount of time to evaluate its likelihood value. Hence, the artificial delays that we introduce to our own likelihood function should be unbalanced, in order to implement the given feature.

In addition, we need to implement the required dynamic workload scheduling for the parallel Strategies 1-4. These modifications of the given parallel strategies can be performed using OpenMP "do" directive. This directive may be represented in the following form: "do [schedule]", where schedule clause represents how the work will be divided among threads. In this case we aim for the cyclic schedule (i.e. each "free" thread will be assigned to evaluate the corresponding likelihood value for some point), therefore, we chose "DYNAMIC,1" option.

Chapter 5

Results and Analysis

Once the parallel strategies were implemented, we conducted a series of test runs for each strategy, to obtain data on the performance of each of them. We configure conditions of these tests in particular way to make them close to the real ones, in which the original application "CosmoNest" works. That is, in these tests search for the likelihood function's maximum regions was carried out in the six-dimensional parameter space with the value of sampling efficiency $s = 9\%$ and the working set containing 300 points. Although the used likelihood function has the shape of a Gaussian, its execution was artificially delayed via introducing additional "waiting" functions. Each thread that performed a call of this function had to wait the given amount of time after calculation of the corresponding likelihood value. In these tests we used two different kinds of delays for each thread. In the first set of tests, the delay was chosen to have static (constant) value in 10 milliseconds. On the contrary, for the second set we used varying delays. Moreover, in order to reduce the execution time of the strategies, we introduced an additional stopping criterion. The strategies stop when the difference between the pre-known analytical value of evidence and the corresponding current evidence value becomes less than a certain threshold.

The runs were carried out on different numbers of threads. As a result of these experiments, the run-times of the parallel strategies that we have received would allow us to obtain the speed-up values for each of the strategies, i.e. we would be able to analyse their scalability. However, the obtained data would represent implementation results. Therefore, instead of analysing just this kind of results, one might want to consider the algorithmic scalability of the given parallel strategies as well.

In this case the algorithmic speed-up may be defined as follows. When some of the presented strategies is executed by a single thread, this thread will perform a particular number of point replacements. Moreover, it will perform some number of likelihood evaluations for new candidate points. When the same strategy is carried out by several threads, the average number of replacements will be the same (but the strategy will be executed faster as the threads will work simultaneously). Therefore, working in parallel each thread will have to perform fewer likelihood evaluations than a single thread

working solo. By analogy with conventional speed-up, which is defined as $\frac{T_1}{T_P}$ (where T_1 run-time for single thread and T_P run-time for P threads), one might represent the algorithmic speed-up as $\frac{N_1}{N_P/P}$, where N_P is the total number of likelihood evaluations that was performed by the P threads.

In fact, since the number of executed evaluations that is performed during some parallel strategy does not depend on its implementation, the performance of algorithmic speed-up will not be hindered by the corresponding implementation overheads. Therefore, we shall consider the algorithmic speed-up as a upper bound for the actual one.

Thus, for each parallel strategy the five runs were conducted on different number of threads (1, 2, 4, 8, 16, 32 and 64 threads). Then, the corresponding average values of T_P and N_P were calculated. As a result, the graphs of the real and algorithmic speed-ups against the number of threads were plotted. These graphs are presented in the following sections. The results of each run are listed in the respective tables, which can be found in Appendix A.1.

5.1 Results for parallel strategies with static delays

5.1.1 Parallel Strategy 1

For the first parallel strategy, in addition to the real and algorithmic speed-ups, we can also obtain the theoretical one. The value of the theoretical speed-up is $(1-(1-s)^P)/s$, where P is a number of threads and s is the value of sampling efficiency (in this case s=9%). Hence, the corresponding results for the first algorithm are presented in Figure (5.1).

The given graph shows that the algorithmic speed-up is almost identical to the theoretical one. The behaviour of the actual speed-up is also similar to the algorithmic one as well. Although, difference between these two speed-ups becomes more noticeable with increasing the number of threads, it remains small enough.

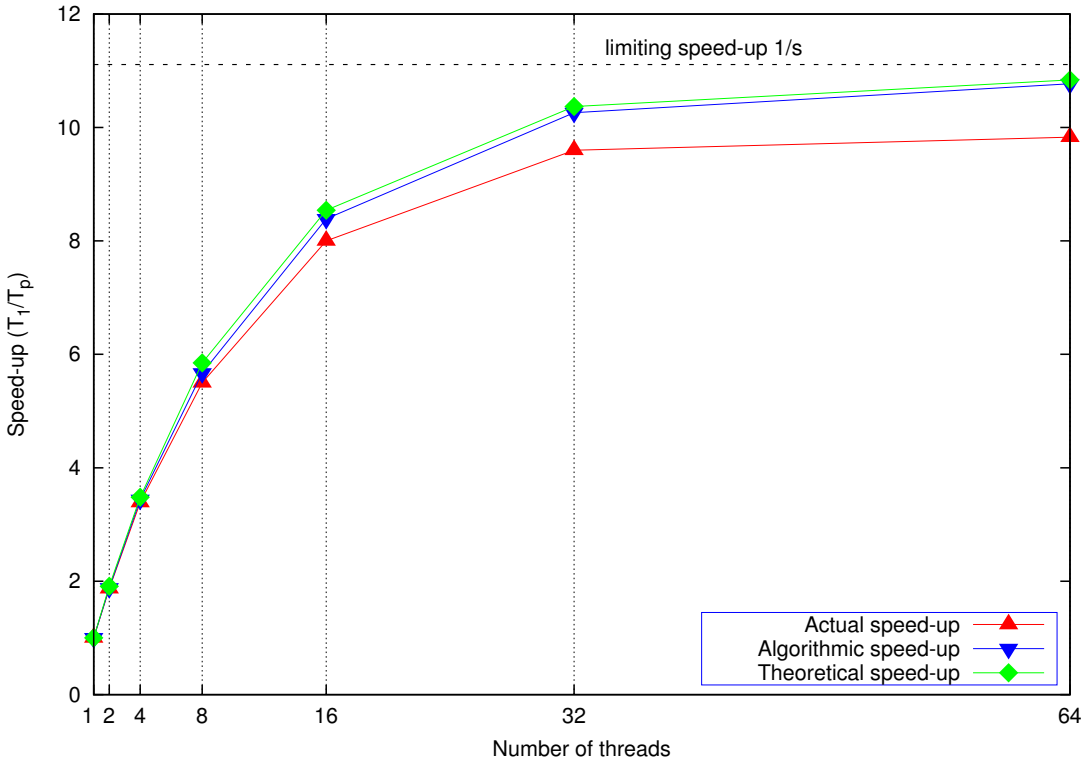

Figure 5.1: Speed-ups for the first parallel strategy versus number of threads

In general, the given parallel strategy shows rather poor scalability as it ceases to follow linear behaviour when the number of threads of more than 8. However, such result was expected. In fact, it is natural, since the theoretical speed-up is limited by upper bound $\frac{1}{s}$ and is not able to exceed this value independently from the number of threads being used. In this case, the value of this upper bound is $\frac{1}{s} = \frac{1}{0.09} \approx 11.1$.

5.1.2 Parallel Strategy 2

The obtained speed-up results for the second strategy are presented in Figure (5.2).

The second parallel strategy shows more efficient scalability than the previous one as in this case the algorithmic speed-up appears to be almost linear. As the number of candidate points becomes greater, the greater number of points may be replaced on each iteration. At the same time, the actual speed-up does not match the algorithmic one starting from 16 threads, although they performed quite similar on 1, 2, 4 and 8 threads. This deviation may be caused by different kinds of overheads that are not considered by the algorithmic speed-up (as it does not depend on the algorithm's implementation and target machine's architecture). Since generation of the candidate points is performed sequentially, before the parallel region, the corresponding data is stored in the memory of the master thread (i.e. in some processor's memory). Therefore, due to the target machine's NUMA-type architecture some threads, which are located "farther" from the

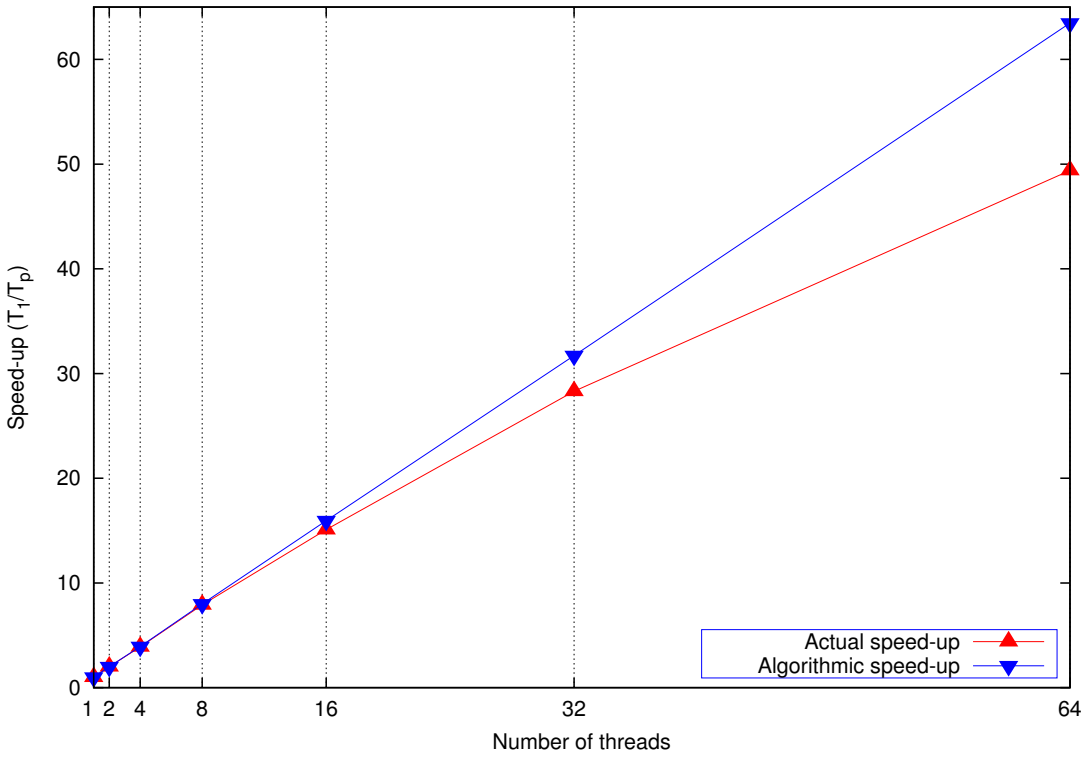

Figure 5.2: Speed-ups for the second parallel strategy versus number of threads

master thread, spend more time to access the required memory locations. Moreover, as we increase the number of threads to run the test, we imply the communications between cores that are located on the different multicore processors. Hence, communication overheads becomes even greater and actual speed-up is hindered.

In addition, with the growth of number of threads the part of the application, which is executed in parallel, is decreased. Therefore, the resulting actual speed-up may become weaker due to Amdahl's Law.

Hence, in order to establish the exact cause of the observed deviations, we analysed the part of the time spent on the execution of the parallel region (we denote it as S_P). We compared this value against the value of obtained overheads, i.e. the difference between the ideal run-time $\frac{T_1}{P}$ and actual run-time T_P. We considered only the cases in which it was the most considerable. The corresponding results are presented in Table (5.1).

Threads	**Actual run-time T_P (sec)**	**Ideal run-time $\frac{T_1}{P}$ (sec)**	**Sequential part S_P (sec)**
1	747.9	747.9	747.9
16	49.65	46.74	3.781
32	26.45	23.37	3.590
64	15.14	11.69	3.453

Table 5.1: Results of the sequential part for Strategy 2 on 16, 32 and 64 threads.

As a result, it was observed that the major part of the obtained overheads came from the sequential part of the code. In fact, one might note that $(\frac{T_1}{P} - T_P) \approx S_P$ for every number of threads. Therefore, it is more likely that the arisen overheads were implied by Amdahl's Law rather than the machine's architecture.

However, the given parallel algorithm's speed-ups are no longer limited by the upper bound $\frac{1}{s}$ as the corresponding speed-ups of the first strategy. Hence, this strategy proves itself as an efficient one with running on great number of threads as its speed-ups remains to be close to the linear. In addition, the actual speed-up is likely to be more close to the algorithmic one as we increase the value of the used delay. Indeed, in that case the overheads will become less significant compared to the values of the obtained speed-up.

5.1.3 Parallel Strategy 3

Graphs for the obtained speed-up results for the third strategy are presented in Figure (5.3).

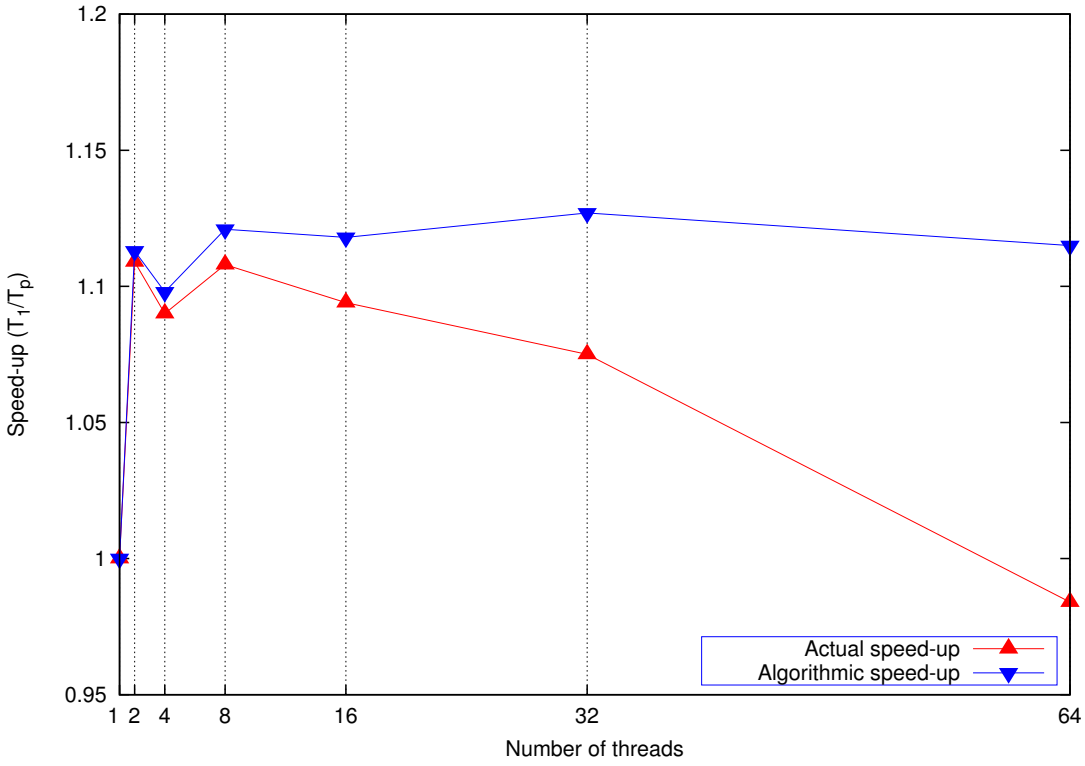

Figure 5.3: Speed-ups for the third parallel strategy versus number of threads

The given graph represents the inefficiency of the third parallel strategy as the corresponding values of its algorithmic speed-up for different number of threads are weak. In fact, these values fluctuate around the value of 1. The maximum value that is reached

by the algorithmic speed-up is less than 1.15 with using 32 threads. Therefore, this parallel strategy does not look promising and improvement from the use of it is almost imperceptibly. Thus, this strategy ceased to be scalable. Hence, since this property is very important for parallel strategies, one might consider the given strategy as impractical and unusable one.

In addition, the corresponding actual speed-up tends to perform even worse as its result values are hindered by implementation overheads (e.g. overheads that are implied by the NUMA-architecture). Therefore, the strategy's run-time on 32 threads as almost identical to its sequential run-time. Moreover, this strategy is executed on 64 threads even slower than on single thread.

However, this outcome is not unexpected for us. This strategy was developed as an intermediate stage for developing the fourth parallel strategy and its beneficial effect was calculated taking into account the unrealistic condition of sampling efficiency s=1. Eventually, since the given implemented performance test implied realistic value of s=9%, this strategy showed rather poor performance. In fact, due to the low value of s each thread will have to perform a number of evaluations (which is proportional to s), in order to obtain new "good" point. Furthermore, to perform replacement of several points on each iteration, it is necessary to satisfy the condition (3.1). Hence, the chance that the majority of the generated candidate points will be used for the replacement is small. As a result, the average number of evaluations, which is performed by some thread during the execution of the strategy, is not likely to decrease significantly as we increase the number of threads.

Thus, taking into account the impracticality of the given parallel strategy, it was decided to abandon it and to undertake further performance analysis without it.

5.1.4 Parallel Strategy 4

The results of performance tests for the fourth parallel strategy are presented in the Figure 5.4.

The given parallel strategy is based on Strategy 3, i.e. we intend to perform replacements of P points via "lookahead". In addition, threads will continuously sample candidate points from the corresponding ellipses until the required points will be obtained. In order to perform balancing of workload between threads, we introduce simple cyclic scheduling.

As can be seen from the given graph, the function of algorithmic speed-up is close to linear (although, it significantly differs from the desired ideal one). The obtained speed-up shows linear growth rate for the number of threads of 32 and less, therefore, this strategy shows decent scalability.

However, function's growth rate becomes significantly reduced as we increase the number of threads. Moreover, starting from 16 threads, the values of algorithmic speed-up cease to be strongly linearly dependent on the number of threads. For instance, the

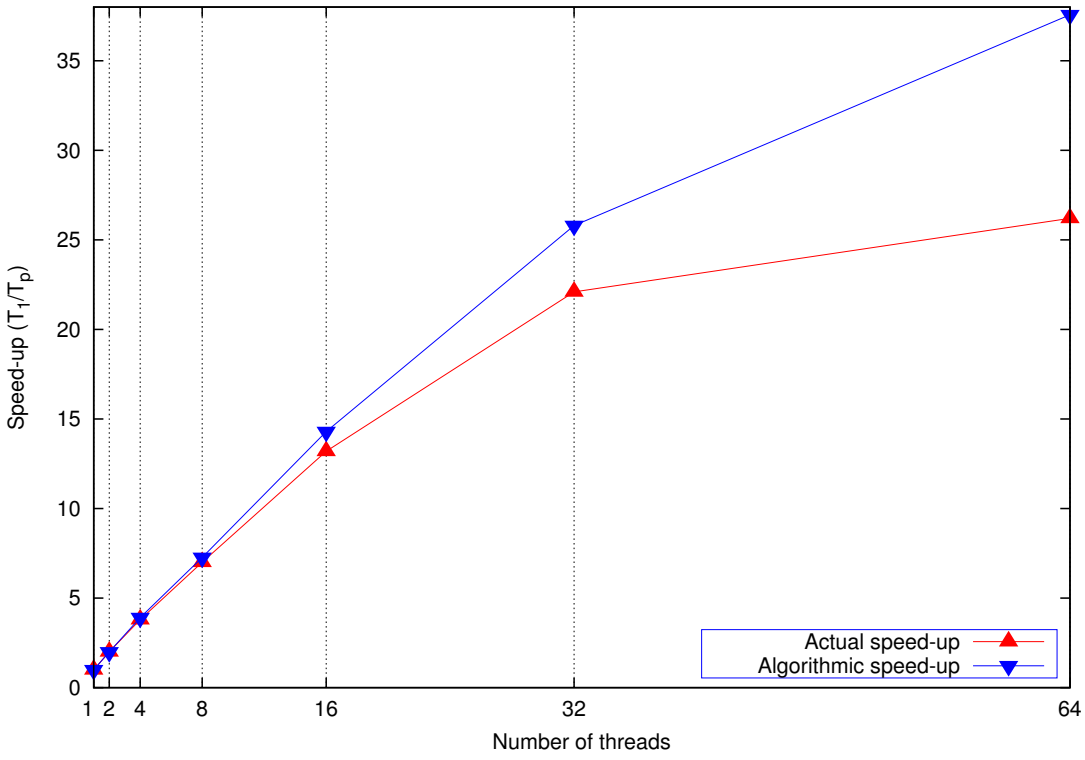

Figure 5.4: Speed-ups for the fourth parallel strategy versus number of threads

value of algorithmic speed-up on 32 threads is about 25. As we increase the number of threads, we aim to perform the greater number replacements of points on each iteration. Therefore, at first sight the given strategy should be executed faster using greater number of threads. However, in order to perform the corresponding replacements, it is necessary to satisfy the mentioned condition (3.1). Moreover, the more threads we use, the more points are generated for participating in "swapping", therefore, for a greater number of points given condition should be performed. Since this criterion should be performed simultaneously for each point, its probability is equal to the product of the probabilities for each of the points. Hence, for large number of points this probability will be small.

Furthermore, the scheduling process is likely to be less efficient for large number of threads. That is, as we set additional "free" threads to help the "busy" ones to sample new points from the corresponding ellipses, we increase the chances to obtain some "good" point. However, if a sufficiently large number of threads already is searching for a point in the given ellipse, the addition of new threads is unlikely to be beneficial. Such situations may arise if the number of ellipses in the corresponding set is small, compared with the number of threads. Hence, such situations will frequently occur at the end of each iteration, if the given strategy is used with a large number of threads.

Thus, for large number of threads the average number of points which were replaced during each iteration will be less than the number of threads. Therefore, the efficiency

and scalability of this parallel strategy will be hindered due to this effect on large numbers of threads.

The obtained actual speed-up correlates with the algorithmic one, although it shows even more modest results as it is hindered by implementation overheads. Similar to the previously presented second parallel strategy, the gap between the actual and algorithmic speed-ups becomes most noticeable above 16 threads. Furthermore, in this case greater amount of work has to be done outside the parallel region as we perform construction of sampling ellipses for each thread as well as point generation. Moreover, we face additional overheads related to workload balancing between the threads. Hence, all these issues deviate the actual speed-up values from the corresponding values of algorithmic speed-up.

Therefore, by analogy with the Strategy 2, to establish the exact cause, we analysed the fraction of time that was spent in parallel region to the total run time. These results are presented in Table (5.2).

Threads	**Actual run-time $\mathbf{T_P}$ (sec)**	**Ideal run-time $\frac{\mathbf{T_1}}{\mathbf{P}}$ (sec)**	**Sequential part $\mathbf{S_P}$ (sec)**
1	749.9	749.9	749.9
16	56.88	46.87	4.670
32	34,00	23.43	4.855
64	28.67	11.72	5.534

Table 5.2: Results of the sequential part for Strategy 4 on 16, 32 and 64 threads.

As a result, it was concluded that the share of the sequential part S_P in the total amount of overheads is not the largest one. In fact, on 64 threads S_P is only 5.5 seconds with the total amount of overheads 16.95 seconds (i.e. the sequential part implies nearly 30% of the total overheads only). Therefore, the sequential part is not the major source of overheads in this case. Hence, it is likely that observed deviations are caused by the other effects such as target machine's architecture.

5.1.5 Parallel Strategy 5

Since the given parallel strategy is based on "Master-Worker" approach, two threads will be the minimum number that is necessary for its execution. Therefore, in order to calculate the values of algorithmic and actual speed-ups, we used the run-time and average number of evaluations, which corresponded to the single Worker-thread. Hence, we consider the results for running test on 2 threads as the data corresponding to the "sequential" execution of the strategy (as the major work is performed by single worker), i.e. $T_1 = T_2$ and speed-up $S_P = \frac{T_2}{T_P}$. Moreover, in this case the number of threads that perform parallel work will be $P - 1$ (as one thread will be the Master). As a result, the ideal speed-up for this strategy will have the value of $P - 1$ on P threads.

Graphs for the obtained speed-up results for the fifth strategy versus used number of threads are presented in Figure (5.5).

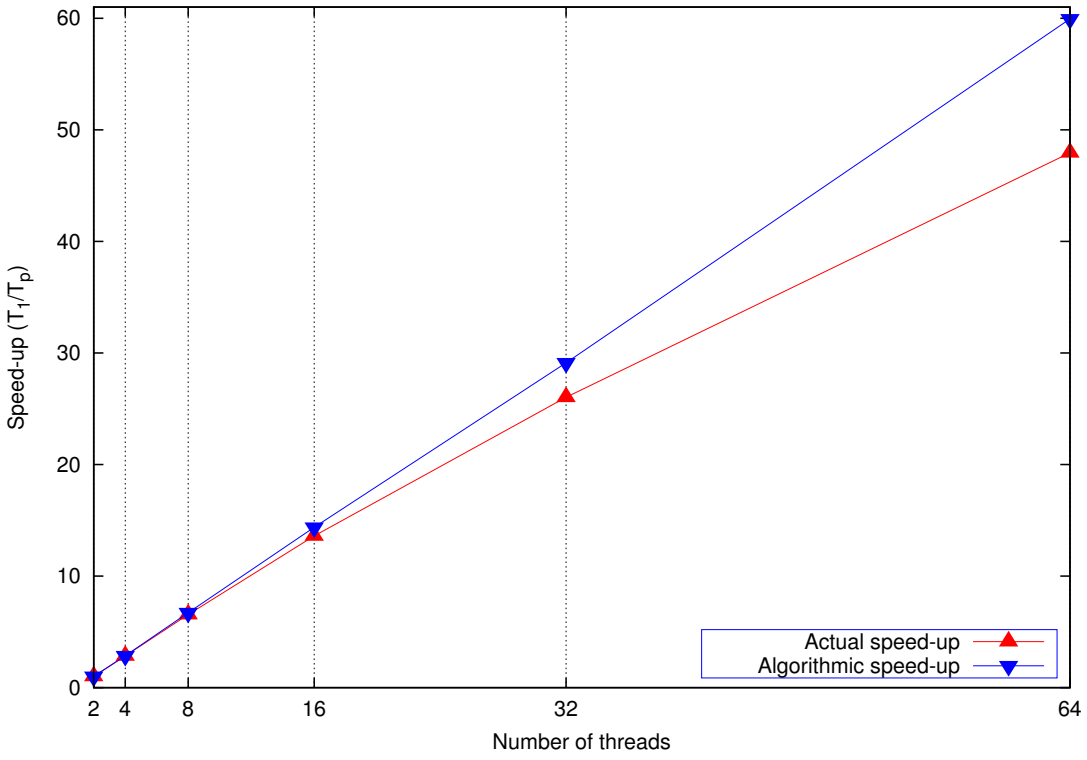

Figure 5.5: Speed-ups for the fifth parallel strategy versus number of threads

The algorithmic speed-up shows the results, which are close to the ideal linear ones. Its function's growth rate remains nearly stable and similar to the value, which corresponds to the direct linear relationship. Therefore, the given strategy presents good scalability. The worker-threads generate new points asynchronously and the corresponding set of candidate points is filled as they become available. The Master-thread processes emerging candidate points and performs replacements. After a particular number of replacements was executed, we achieve the required value of evidence Z and terminate the strategy. Hence, how fast this parallel strategy will produce the output results depends on the "filling" rate of the candidate set. The rate of filling of the set with the points grows with the growing number of workers. As we increase the number of threads, the average number of evaluations, which is performed by some worker-thread, decreases. Thus, effectiveness of the given strategy remains large enough, when using it on a large number of threads.

The actual speed-up faces a number of issues, which are implied by implementation, therefore, its result values are hindered. Addition to the normal overheads, which are implied by the machine's NUMA architecture and which affect all of the presented strategies, Strategy 5 is influenced by the overheads caused by "critical regions" as well. That is, the implementation of this strategy requires introducing the code regions that are performed in parallel by a number of threads, although only one thread at a time is allowed to execute the given region. These regions represent natural bottlenecks for the

threads that work simultaneously. In this case, in order to preserve the set of candidate points, which is implemented as linked list structure, it is necessary to introduce critical region for the addition new points (otherwise, threads will corrupt the list via attempts to add new points to it simultaneously). Therefore, as the number of threads becomes greater, the contention for the critical region may significantly hinder the actual speed-up. Although, the given graph did not show a significant deviation of actual speed-up from the algorithmic one, it is likely that the gap between them will become more notable at larger thread numbers.

In principal, the observed difference between the actual and algorithmic speed-ups for the given parallel strategies will be less, if the time, which is required to calculate likelihood values, increases (i.e. the given static delay increases).

5.1.6 Comparison of the results depending on the value of sampling efficiency

In order to determine the best parallel strategy, we conduct a series of experiments with different values of sampling efficiency s for the given strategies (excluding the third one). One can easily change the value of sampling efficiency by simply changing the value of the enlargement factor for the sampling ellipse. In fact, as the sampling ellipse becomes smaller, the frequency of obtaining new points (i.e. s) rises. On the contrary, the value of s decreases, when we use larger sampling ellipse.

The chosen values of s corresponds to small value (s=3%), large value (s=30%) and standard value (s=9%). The results of these tests will allow us to determine which parallel strategy is the most effective one for the each respective values of s. The value of sampling efficiency s represents the amount of exploitable parallelism that is available for the parallel strategy. The lower this value, the greater the number of threads can be effectively used by strategy, and vice versa. Hence, these tests will show how the strategy's behaviour depends on the amount of available parallelism.

Furthermore, since the run-times for single thread are slightly different for each parallel strategy, it was decided to use the value of $\frac{1}{T_P}$ instead of common speed-up for the analysis. This approach will help us to avoid unintended subjectivity in comparing strategies (as the run-time T_1 no longer influences the corresponding values). Moreover, in this case it will be more appropriate, since the fifth strategy uses at least 2 threads.

These tests were conducted for all parallel strategies, except Strategy 3. The graph that represents the performance of the given strategies for the standard value of sampling efficiency is given in Figure (5.6).

The first parallel strategy shows the weakest scalability of all as the values of its speed-ups are limited by the value of $\frac{1}{s}$. At the same time, all other strategies show rather identical results for a small number of threads (less than 16) with the second one being slightly better. For greater numbers of threads the results of fourth strategy cease to be similar to the corresponding results of the second and the fifth strategies. Although, it

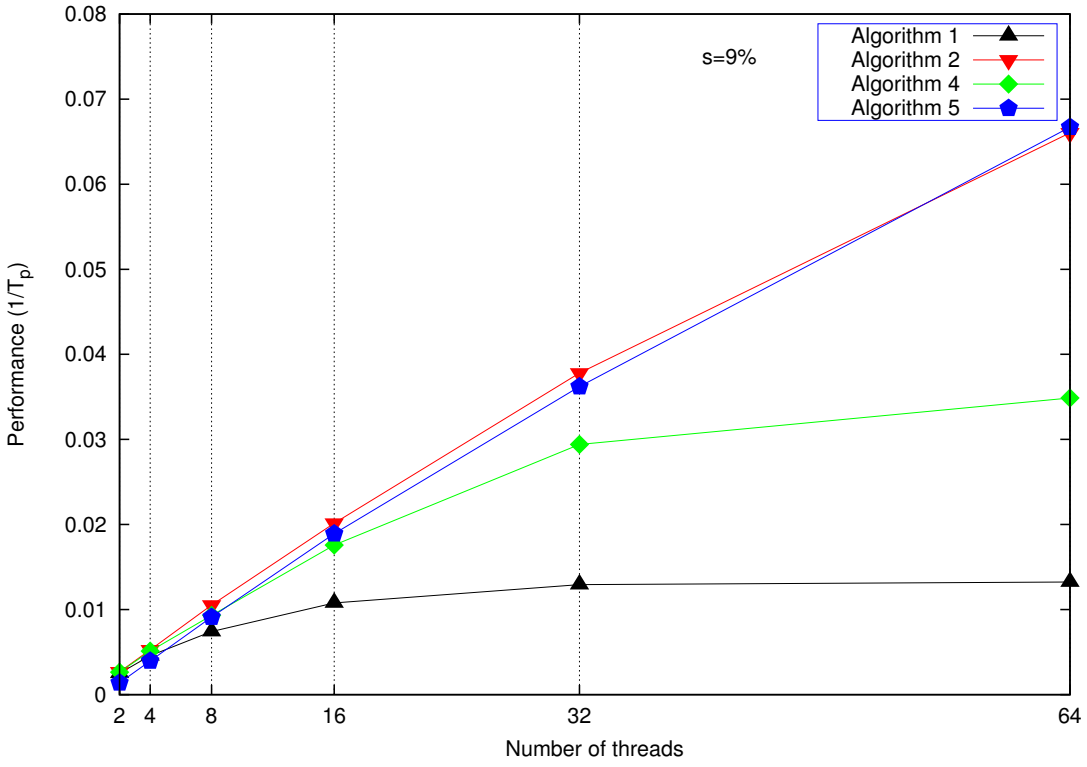

Figure 5.6: Values $\frac{1}{T_P}$ for the parallel algorithms 1, 2, 4 and 5 versus number of threads for sampling efficiency s=9%.

performs better than the first one, the gap between these strategies becomes more significant. As expected, best results were shown by the second and the fifth strategies (as they proved themselves as highly scalable ones). Though, the second strategy performed slightly better than the fifth one, the difference between their values remains insignificant. Moreover, their results are similar on 64 threads.

Thus, for standard value of sampling efficiency s strategies 2, 4 and 5 perform almost identical to each other on small number of threads. Meanwhile, the second and the fifth strategies show the best performance for the number of threads greater than 16.

The next test was conducted using value of sampling efficiency s=3%. That is, the amount of parallelism, which is implied by the given test case, increases.

The corresponding results are presented on graph in Figure (5.7), below:

In general, the given parallel strategies show the similar behaviour as for the previous case. The second and the fifth strategy represent supreme performance, meanwhile the fourth and the first strategy show weaker results. In addition, for small number of threads the gap between these strategies seems rather insignificant, although it becomes more noticeable as we increase the number of threads.

At the same time, the nature of the corresponding results for the Strategies 2 and 5 re-

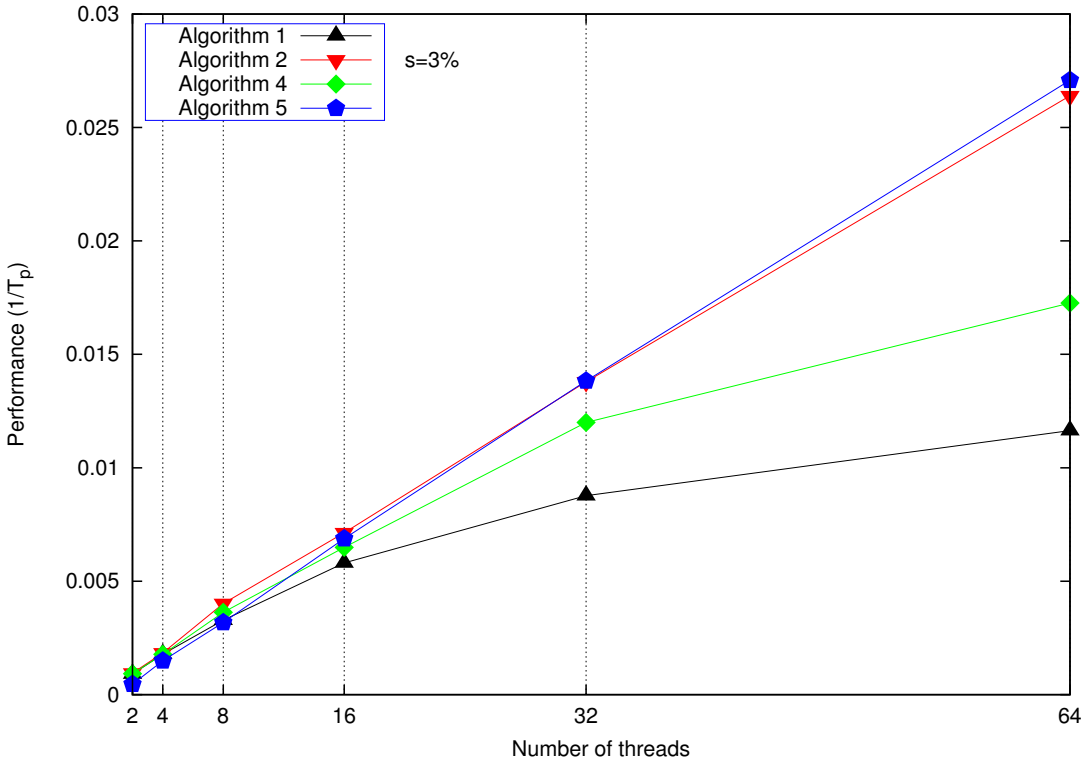

Figure 5.7: Values $\frac{1}{T_P}$ for the parallel algorithms 1, 2, 4 and 5 versus number of threads for sampling efficiency s=3%.

mains essentially the same, despite the significant decrease in the value of s (with the fifth one performing slightly better). In the case of Strategy 5 this decrease leads to the fact that the average number of evaluations, which is performed by some worker-thread in order to obtain new "good" point, rises. Since this effect is valid for all threads, the overall speed-up remains the same (as adding new worker-threads still will be beneficial). Low value of sampling efficiency cause the similar effect on the second strategy. Although the total run-time of the fifth strategy was increased (as it takes longer to receive new "good" point with low values of s), the growth rate of the given strategy's speed-up remains nearly the same. In fact, as we add new threads, we increase the chances to perform the greater number of points replacements on each iteration. Though the sampling efficiency affects the performance of the threads, it does not influence the strategy's scalability. Moreover, since the reduction of the value of s makes it more profitable to use a larger number of threads and in the previous case (for s=9%) this strategy's speed-up has already shown efficient performance, it remains efficient in the given case as well.

However, the first and the fourth strategies showed better results in this case. The gap between these strategies and the previous one became less. Since the value of sampling efficiency decreased, the upper limit value $\frac{1}{s}$ for the first strategy increased, so more threads can be effectively used to sample new points in each iteration. Thereby, in this

case its speed-up becomes more linear (although, its growth is still limited by $\frac{1}{s}$).

Meanwhile, the decrease the value of s makes the load balancing schedule more effective for Strategy 4. Due to the fact that threads have to perform greater number of evaluations to get new "good" point from ellipse, the amount of work, which is implied by sampling process in this ellipse, increases. Hence, the total workload may be divided among threads more easily, i.e. more efficiently. Therefore, the corresponding speed-up tends to be more linear and give the results that are closer to the second and the fifth strategies.

Thus, the strategies that already have proved themselves as highly scalable ones remained the same (as their good scalability is implied by the efficient exploit of the amount of the all available parallelism, therefore, the rise of this amount did not affect them considerably). At the same time, the first and the fourth strategies showed better performance, since their scalability is strongly dependent on the amount of available parallelism.

Finally, we picked a large value of sampling efficiency s=30%.

The graph in Figure (5.8) shows the obtained results.

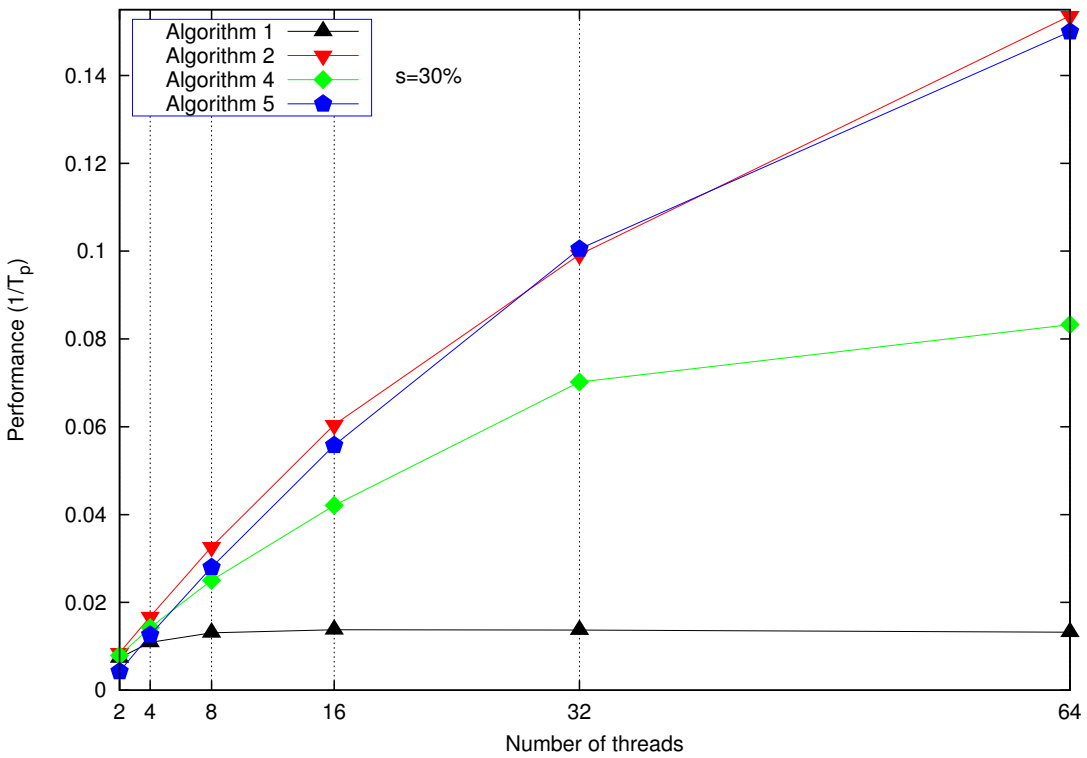

Figure 5.8: Values $\frac{1}{T_P}$ for the parallel algorithms 1, 2, 4 and 5 versus number of threads for sampling efficiency s=30%.

As a result of comparing the data that is presented on the given graph with the corresponding results, which were shown in Figure (5.6) for $s = 9\%$, one might conclude

in that the behaviour of the speed-ups deviated from linear. The observed deviation is caused by the increase in value of sampling efficiency s.

In fact, the first strategy shows almost constant results of the speed-up for the number of threads greater than 8. The large value of s quickly negates the benefits of adding new threads, since the small number of them is sufficiently enough for effective sampling. Hence, in this case Strategy 1 has poor scalability.

For the fourth strategy, reducing the volume of the work, which is executed by thread during sampling process to obtain "good" points, leads to the fact that the total amount of work that is available to the separation between the threads is significantly reduced. Therefore, it becomes difficult to divide the work among threads, and hence to balance the work load between them. As a result, the scalability of this strategy becomes hindered and the obtained speed-up ceases to be linear (this is most noticeable for large numbers of threads).

However, one might note that the speed-up performances of the second and the fifth strategies were changed to a lesser extent than the corresponding results of the previous strategies as their speed-ups remain relatively linear.

In the case of Strategy 5 the large value of sampling efficiency s implies that the new points will be generated at the great rates by the worker-threads. Therefore, the master-thread will not have time to process all the incoming points. It evidently leads to work imbalance as the master-thread will have to deal with the amount of work that is significantly greater than the corresponding amount of work, which is executed by some worker. Hence, the overall run-time for large numbers of threads becomes longer and the growth rate of the speed-up slows down. Although, the speed-up of this strategy is hindered by these issues, it remains to be nearly linear due to the delay, which we used in the given test (as it holds back the pace of generating new points).

At the same time, Strategy 2 exploits the idea of using as many generated "good" points as possible for the replacements on each iteration. The large value of sampling efficiency s leads to the fact that almost every thread generates a "good" point. Therefore, the possibility that these points will "overlap" each other in the process of replacing increases. That is, after replacing the "old" points to the new ones, these new points will be replaced by other new points that have greater values of likelihoods within one iteration. Hence, as we increase the number of threads, the efficiency of exploiting new "good" points decreases. Although, it affects the speed-up of the strategy, in the end Strategy 2 still performs "swapping" of most of the "old" points at each iteration. Therefore, the speed-up, which is obtained by using this strategy, is fairly linear. However, it should be noted that the given parallel strategy may imply the bias sampling as the value of s becomes large. In fact, since the major part of "old" points are replaced with the new ones that were sampled from some ellipse, it is likely that we might become "stuck" in the corresponding region. One should consider this factor, when using this strategy for problems with large values of sampling efficiency s.

Thus, the best speed-up results in all three cases were shown by the second and fifth strategies. In fact, their speed-ups tended to be independent by the value of sampling

efficiency. Meanwhile, the corresponding speed-up of Strategy 1 showed strong dependence from the values of s (though, this was expected). Strategy 4 showed quite decent results, although they could not reach the similar values as the ones of Strategies 2 and 5.

5.2 Results for modified parallel strategies with unbalanced delays

The configuration of performance tests remained the same, though the values of delay was chosen to be defined as $max(1, 10 + 5 * N(0, 1))$ milliseconds, where $N(\mu, \sigma)$ is a normal distribution function of the form

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} exp\Big(- \frac{(x-\mu)^2}{2\sigma^2}\Big).$$

The given tests were performed for each strategy with standard value of sampling efficiency s=9% and for different values of factor k equal, respectively, to 1, 2 and 4. The obtained results for each strategy are presented in the following subsections.

5.2.1 Modified Parallel Strategy 1

In the case of the first strategy, introducing the factor k indicates that at each iteration P threads generate $k \times P$ points, one of which is selected for replacement. The obtained results, which represent the strategy's speed-up against the number of used threads for various values of k, are presented in Figure (5.9).

As expected, using unbalanced delays instead of static ones leaded to the drastic drop of the strategy's performance (as the "old" speed-up for the static delays is significantly greater than the speed-up for the unbalanced delays).

The value of sampling efficiency s affects the average number of attempts, which is required to sample a new "good" point on each iteration. In general, this number will be inversely proportional to the value of s. That is, in this case one might need nearly 11 attempts (as s=9%). As we increase the number of threads, we increase the number of simultaneous attempts that is performed in parallel. Therefore, at some moment the number of threads will be large enough, hence each thread on average would need only one attempt to obtain a new "good" point. Hence, Strategy 1 will reach its maximum available efficiency and additional threads will not make its execution significantly faster. For the standard situation, when each thread generates one point (k=1), such moment comes when the number of threads is greater than 16 (as can be seen from graph above). The growth of value of factor k leads to this moment coming for a smaller number of threads. In this case the strategy's speed-ups ceases to be linear starting from

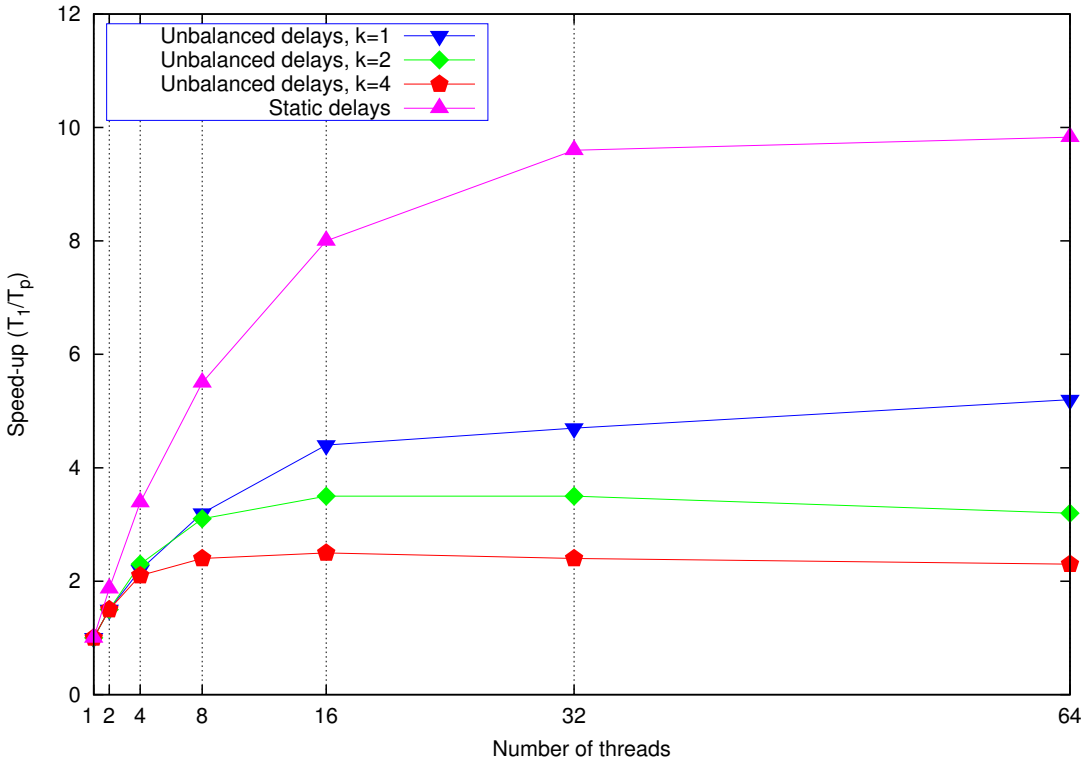

Figure 5.9: Speed-ups for the first parallel strategy versus number of threads with unbalanced delays for $s = 9\%$.

8 threads and 4 threads for the values of factor k equals 2 and 4, respectively. In fact, for these values of k the speed-up remains almost constant on large numbers of threads.

5.2.2 Modified Parallel Strategy 2

For the second parallel strategy we introduced the factor k by the similar implementation as for the first one. On the each iteration $k \times P$ points are generated by P threads. However, in this case we aim to exploit as many generated points as we can (instead of only one point).

The graph in Figure (5.10) shows the obtained results for strategy's speed-ups for different values of k.

Introducing the unbalanced delays significantly hindered the overall strategy's performance since the corresponding results for the case of unbalanced delays and k=1 (i.e. strategy does not intend to handle these delays) are considerably weaker than the results for the static delays.

Furthermore, as can be seen from the graph, increasing the value of the factor k makes the behaviour of the speed-up to be more linear. Since the overall amount of work that

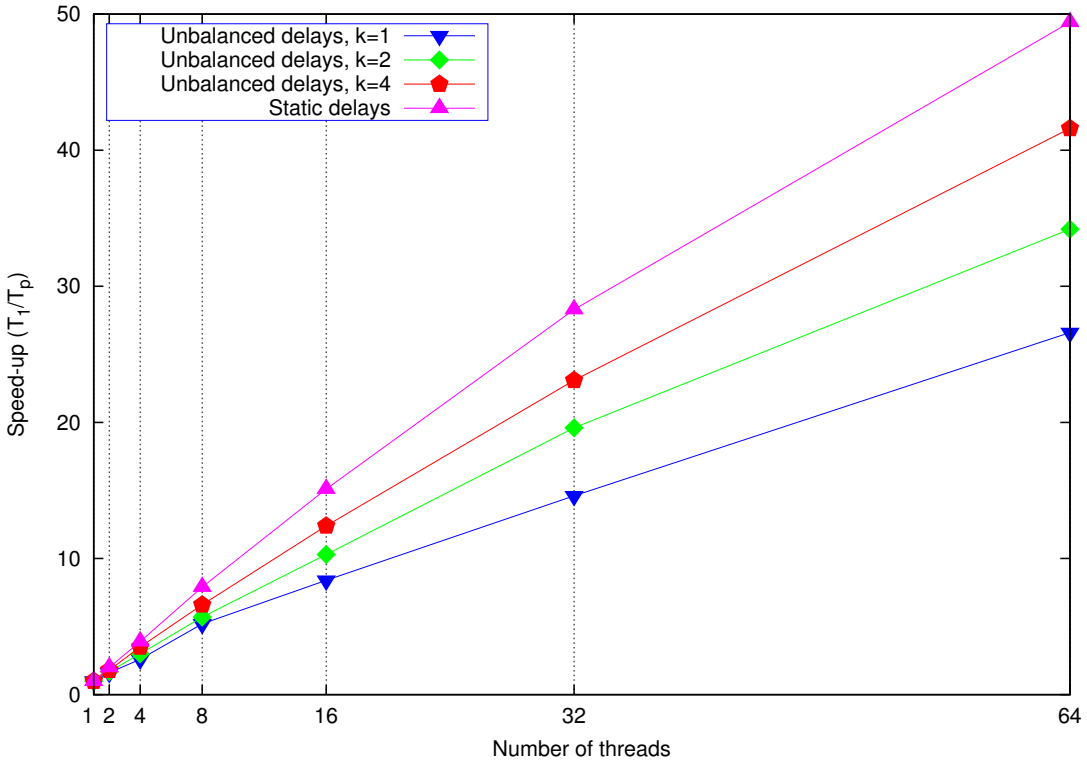

Figure 5.10: Speed-ups for the second parallel strategy versus number of threads with unbalanced delays for $s = 9\%$.

is executed by threads grows as we increase the values of k, the workload balancing becomes more efficient, hence, the speed-up improves. Moreover, although the large numbers of generated points leads to "overlapping" of them during the replacement phase, this strategy aims to exploit all new "good" points. Therefore, as the number of points that can potentially be used for replacement rises, the efficiency of Strategy 2 grows as well. In fact, in the standard case with k=1 the maximum number of points, which can be used during the replacement stage, is P. At the same time, the corresponding maximum values for the cases with k=2 and k=4 will be $2 * P$ and $4 * P$, respectively. Thus, the more replacements being executed during iteration, the faster the strategy will be executed. However, one should consider the already mentioned danger of bias sampling (i.e. this strategy is likely to struggle in solving problems when likelihood function has several local maximas).

5.2.3 Modified Parallel Strategy 4

The fourth parallel strategy already has workload balance mechanism as it cyclically assign the threads, which have obtained "good" points from the corresponding ellipses, to help other threads with sampling process for the remaining ellipses. In the case of static delays, such mechanism would be efficient since each thread executes the same amount

of work considered with likelihood evaluations for some point and assigning additional threads for sampling process increases the probability of obtaining new "good" point. However, in the given case we use the unbalanced delays, hence corresponding likelihood evaluations for different points may take sufficiently different times. Therefore, the given approach in managing workload among threads will be ineffective. The reason for this will be that we do not know the exact amount of work that will be carried out by some thread working in ellipse. Thus, it is likely that the workload for different threads will be uneven.

Hence, we introduce the factor k in the strategy in order to improve the workload balance. In the corresponding stage P threads will generate $k \times P$ new points for a number of remaining ellipses. These points are picked from ellipses according to the threads affinity values, i.e. if some ellipse has affinity with P' threads, then we will sample $k \times P'$ from it. Then, we cyclically assign the threads for likelihood evaluations for the given points. That is, affinity of some thread to particular ellipse is no longer relevant as thread may be assigned to evaluate the likelihood for the point that was sampled not from the ellipse, to which the thread is "tied".

Thus, rather than assigning the threads to work in particular ellipses, we assign them to process particular points instead. Regardless of the number of the remaining ellipses, threads will generate $k \times P$ points. In general, the strategy remains the same: the only difference is the number of points generated by threads. Hence, the workload between threads remains balanced as the amount of work is known (the remaining number of points) and can be easily managed. Furthermore, we overcome the imbalance that is implied by unbalanced delays via using cyclic schedule.

The graph that represents the performances of the given strategy for the standard value of sampling efficiency is given below in Figure (5.11).

As expected, the increase of the amount of work, which is done by threads, leads to that the workload of the threads being managed more effectively. Therefore, for the factor k=2 the corresponding strategy's speed-up shows better performance results than for the case with factor k=1. However, further increase in the value of the factor k resulted in rather slight improvement of the strategy's speed-up (especially for moderate numbers of threads). The efficiency of workload management reaches the maximum value, hence the growth rates of the speed-ups are similar for the factor values k=2 and k=4. On the number of threads is greater than 32, one can observe these rates are falling due to the number of reasons that was already mentioned (e.g. NUMA type of the architecture).

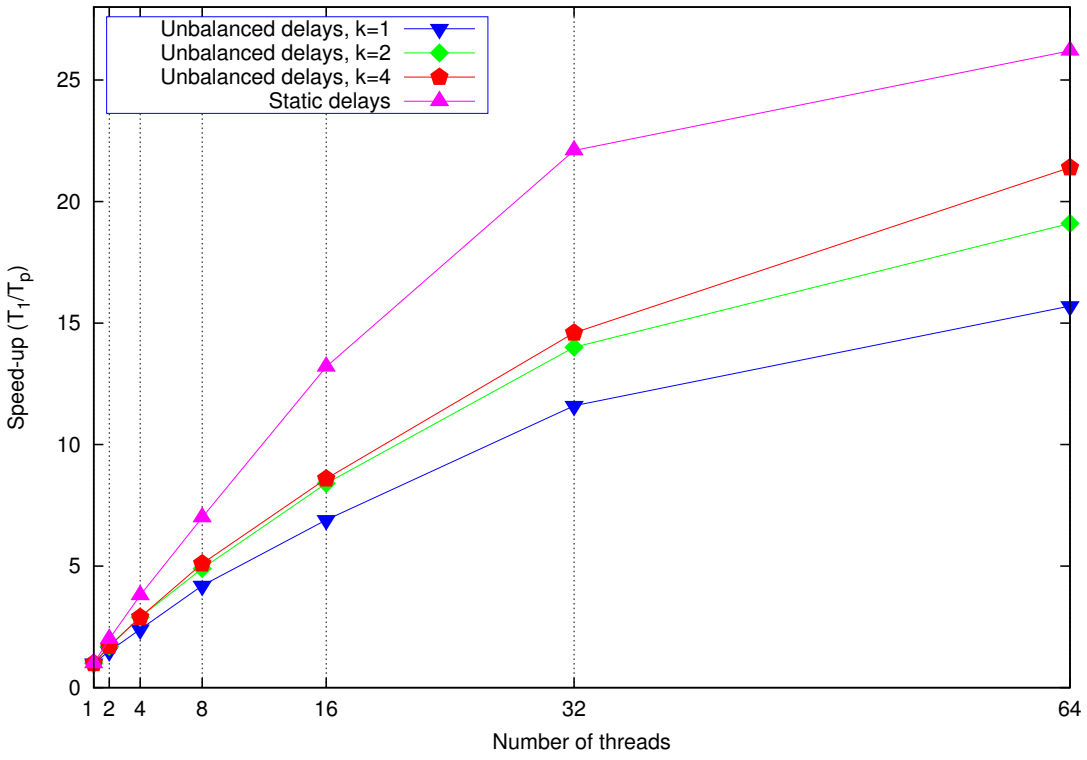

Figure 5.11: Speed-ups for the fourth parallel strategy versus number of threads with unbalanced delays for $s = 9\%$.

Although, this occurs in all cases, the changes of the speed-up's growth rate have less drastic nature in the case of factor k=4. Since the rise of the number of generated points increases the overall amount of work that is performed by threads in parallel, the strategy shows better scalability with the greater value of k.

Thus, the increase of the overall amount of work using the factor of k helps to improve the hindered strategy's speed-up to a certain extent, although we failed to achieve the results that were shown using the static delays.

5.2.4 Modified Parallel Strategy 5

The introduction the factor of k for Strategy 5 would not make sense as this strategy already has methods for workload balancing. In fact, it exploits more abstract approach as a main idea. That is, each worker-thread generates new points asynchronously and independently from other workers. Hence, the fact that the time that is required to obtain a new point will be different for each point will not affect the performance of the strategy. Therefore, it does not require new modifications.

Graphs for the obtained speed-up results for the fifth strategy using unbalanced delays are presented in Figure (5.12).

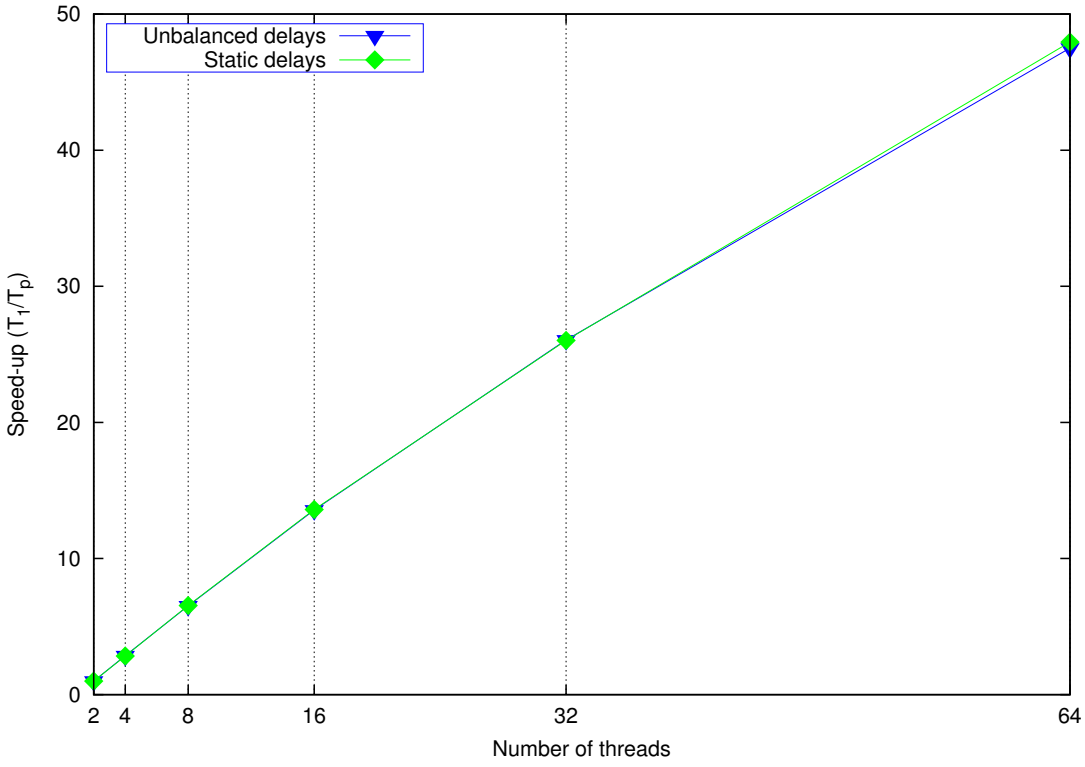

Figure 5.12: Speed-ups for the fifth parallel strategy versus number of threads with static and unbalanced delays for $s = 9\%$.

For the test with unbalanced delays the given strategy showed identical results for the speed-up as for the test with static delays. Since Strategy 5 is based on the "Master/-Worker" pattern, each worker-thread does not depend on other threads. In fact, threads that finish their work before the others do not have to wait for "lagging" ones, hence the strategy's run-time is not hindered by the execution time of the slowest thread. Furthermore, the average rate of the new points emerging in the list remains the same, therefore from the point of view of the master-thread, working conditions remain the same. Thus, as the efficiency of the given strategy in general depends on the rate of processing of candidate point, the speed-up does not change.

Next, by analogy with the previous test, it was decided to compare the best performance results for each strategy to each other for different values of sampling efficiency s using the values of $\frac{1}{T_P}$. Therefore, we conduct a series of performance tests for the each parallel strategy with unbalanced delays and different values of s. For Strategy 1 tests were performed with the value of the factor k=1 and for Strategies 2 and 4 with the value of k=4. The only difference from the previous corresponding tests for the fifth test strategy was that in this case we used unbalanced delays.

The obtained results are presented in the following subsection.

5.2.5 Comparison of the results depending on the value of sampling efficiency

The results for the performances of the parallel strategies with unbalanced delays and the standard value of sampling efficiency s=9% are shown in Figure (5.13):

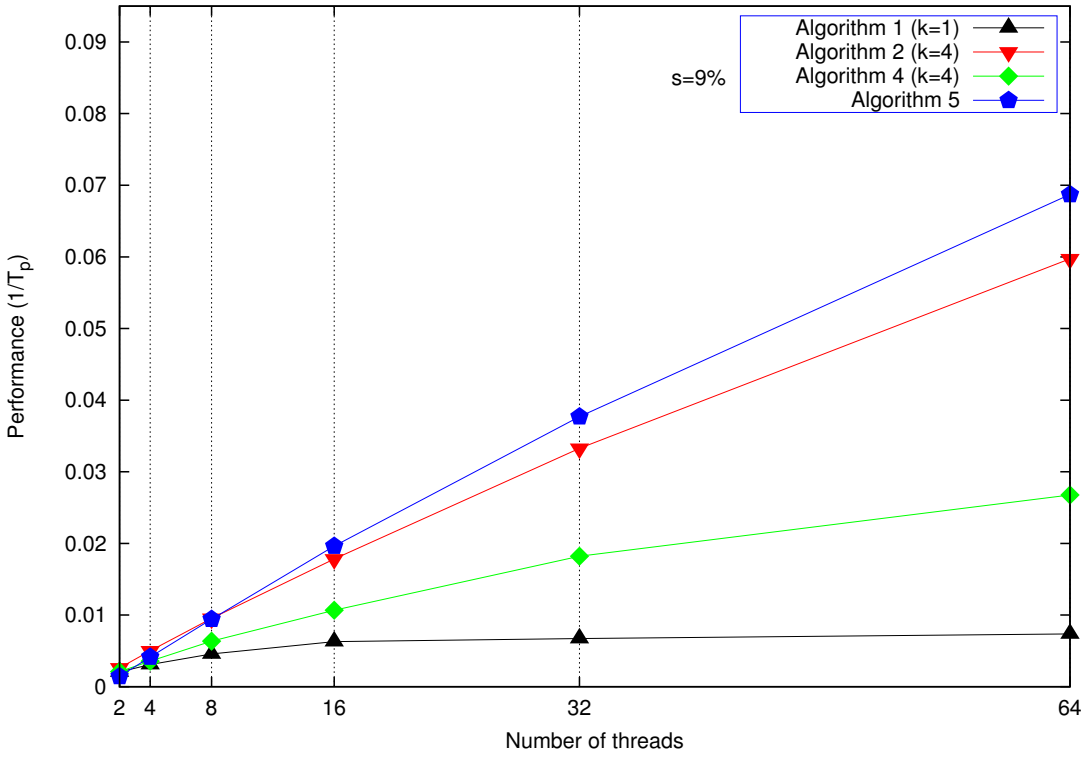

Figure 5.13: Values $\frac{1}{T_P}$ for the parallel algorithms 1, 2, 4 and 5 versus number of threads for sampling efficiency s=9% and unbalanced delays.

In general, the obtained performances of the strategies are rather similar to the corresponding results that were obtained in the case with static delays and the same value of s (graph in Figure (5.6)). In fact, the second and the fifth strategies continue to show better results than the first and the fourth ones. Moreover, in principal, the Strategies 1, 2, and 4 exploit the similar approach to handle the workload imbalance that is implied by unbalanced delays (although, for each of these strategies has been selected then the value of the factor k, which provides better performance). However, the effectiveness of this method was different for the different strategies. Indeed, Strategy 4 showed even weaker results than in the case of static delays, while in the case of the second strategy, the corresponding results were approximately the same.

Furthermore, it is worth noting that, in contrast to the case with static delays, in this situation, the gap between the performances of the second and fifth strategies becomes more significant. In the previous tests with static delays, the performances of these strategies were almost identical (Figures (5.6 - 5.8)). Strategy 5 (which remained un-

modified for the given tests) proves itself to handle the unbalanced delays better than the second one. Although, on the number of threads less than 16 Strategy 2 showed better performance (as the fifth strategy uses one thread as the Master, hence, it uses less number of threads to work in parallel), on the greater number of threads the difference becomes more noticeable. Hence, the introduction of the unbalanced delays allow Strategy 5 to take the lead in terms of the best performance as it was the only strategy whose performance was not hindered.

Next, by analogy with the previous series of tests, we ran tests with different values of sampling efficiency s. Thus, we conducted the set of tests for different value of sampling efficiency, which corresponds to small value (relatively to the standard case with s=9%), i.e. in this case s=3%. The graph in Figure (5.14) shows the obtained results.

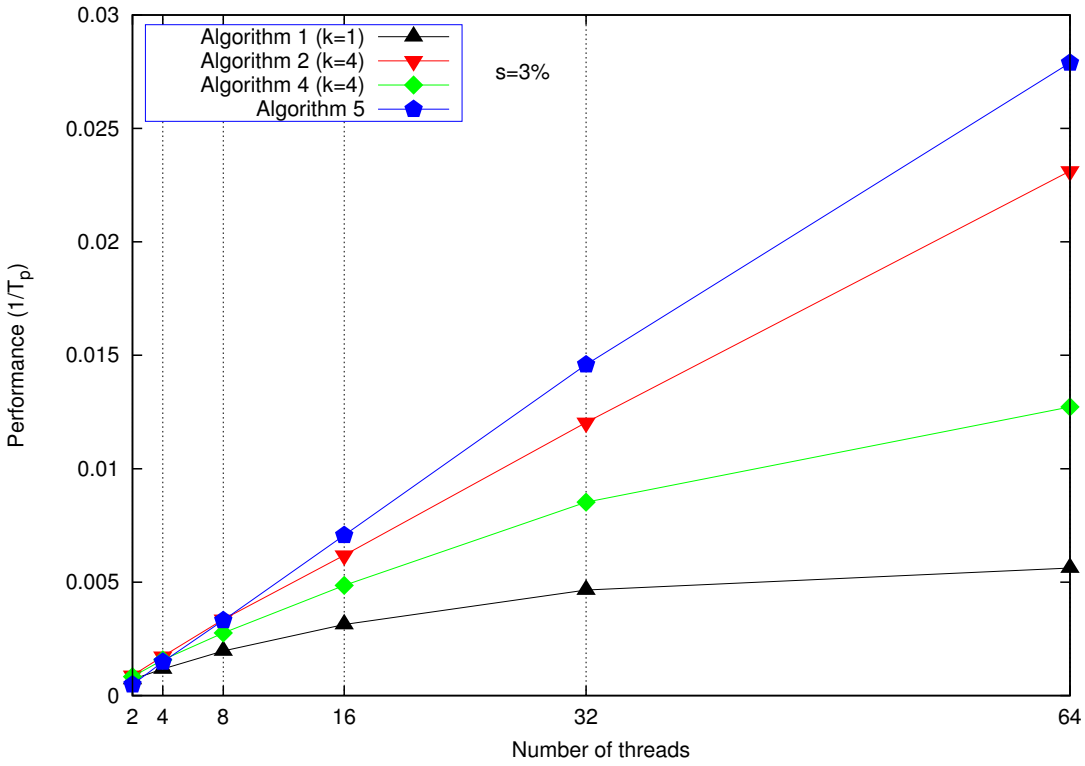

Figure 5.14: Values $\frac{1}{T_P}$ for the parallel algorithms 1, 2, 4 and 5 versus number of threads for sampling efficiency s=3% and unbalanced delays.

The growth of the amount of available exploitable parallelism, which was implied by the decrease of the value of s, made the speed-ups of the given strategies more linear. The similar effect has already been observed in the corresponding case for the static delays. However, in this case it can be stated that the fifth strategy showed the best result. In fact, though the increase of the number of attempts that is required to sample new point rises the time costs considered with points generation, the speed-up's growth rate does not change (as the average time cost for each point remains the same). Therefore, the

gap between the speed-ups of the second and the fifth strategies remained relatively the same as in the case of the sampling efficiency s=3%. Thus, Strategy 5 shows the best results in this case as well.

Then , the last set of tests was conducted for the value of sampling efficiency s=30%. The graph in Figure (5.15) shows the obtained results.

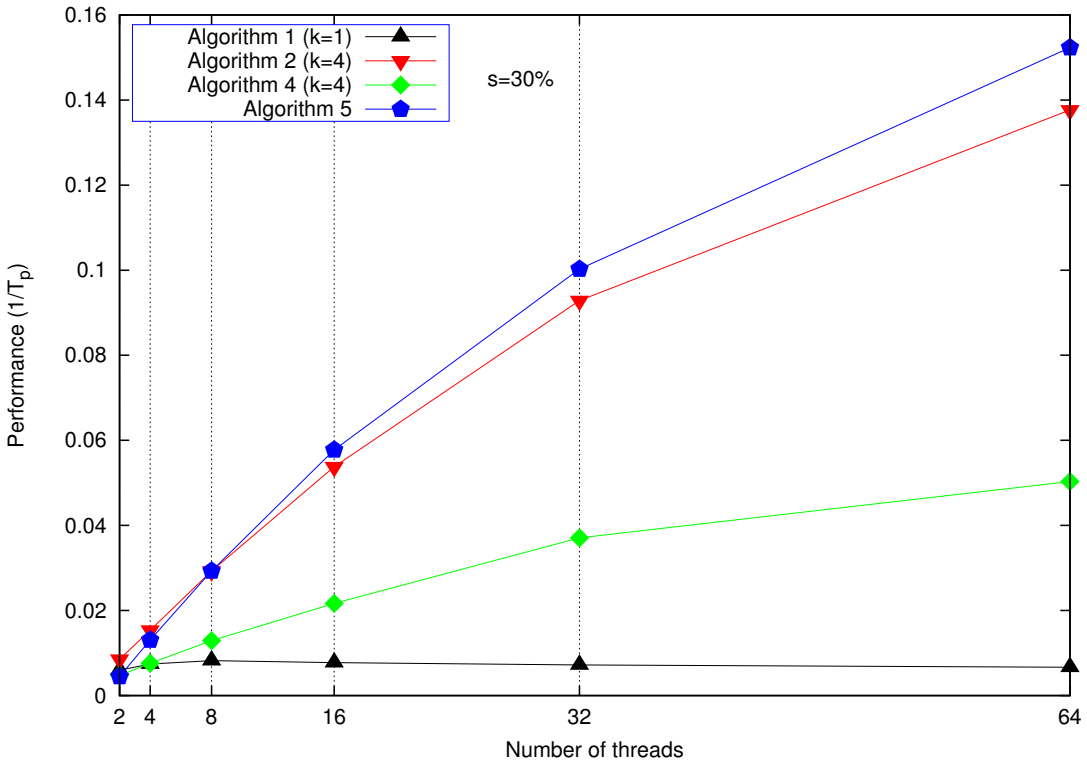

Figure 5.15: Values $\frac{1}{T_P}$ for the parallel algorithms 1, 2, 4 and 5 versus number of threads for sampling efficiency s=30% and unbalanced delays.

As expected, the behaviour of the speed-ups deviated from linear. In general, the decrease of the value of sampling efficiency s for the case with the unbalanced delays had the same effect as in the case of the static delays. However, one might consider the fact that the gap between the speed-ups of the second and the fifth strategies became less than it was for value of s=3%. Since, Strategy 5 is based on the "Master-Worker" approach, reducing the amount of available exploitable parallelism causes more significant negative effect for this strategy than for the others. In fact, Strategies 1, 2 and 4 intend to exploit parallelism that is available during some single iteration, while the fifth strategy uses more abstract approach. Hence, its speed-up is hindered more considerably than the speed-up of the second strategy, therefore the difference between their values is less for the given case.

Moreover, the performances of the first and the fourth strategies show the similar behaviour as they did for the case with static delays. That is, the gap between the performance of the Strategy 4 and the corresponding performances of Strategies 2 and 5

becomes greater. Furthermore, as we increased the value of s, the limiting lower boundary $\frac{1}{s}$ for the first strategy decreased, hence it now may be achieved with fewer threads. As a result, Strategy 1 showed rather poor scalability.

Thus, introduction of the unbalanced delays made the conditions in which our parallel strategies evaluated more realistic. Based on the results of the given performance tests, one can determine that Strategies 2 and 5 showed the overall best results in terms of scaling and efficiency. Parallel Strategy 4 turned out to be less efficient relatively to them since there was the considerable gap between its results and corresponding results of the second and the fifth strategies. In the case of Strategy 1, the speed-up was restrained by the limiting factor $\frac{1}{s}$ (though it was expected), hence its results were much more modest.

Although the results for the second and the fifth strategies differed not so much, however, Strategy 5 showed slightly better results. Furthermore, one should consider the possibility of bias sampling while using the second strategy. Though, this issue was mentioned, we have not conducted any particular tests for the second strategy in order to check whether it is capable to overcome it.

Chapter 6

Conclusions

The goal of this project was the study of different possible methods to parallelise the "CosmoNest" application in terms of the obtained resulting speed-up. In the course of the project we aim to consider and try out different approaches that might be exploited in order to reduce the run-time of the program.

We conducted a set of tests to obtain profiling data for the original sequential code and, hence, to define fraction of the application that should be parallelised. This fraction turned out to be the function which was responsible for evaluation of the likelihood values for the points (Section 2.4). However, it was decided to use more abstract approach instead of simple direct parallelisation of this function. The reason was the fact that the given method would allow the developed solution to be independent of the particular likelihood function (which is based on the background cosmological theory and might change due to some new data). Furthermore, in general this application can be applied in other areas other than Cosmology.

Thus, we intended to modify the original Nested Sampling algorithm that was implemented in the "CosmoNest" application. We introduced five versions of the sequential algorithm wherein each version contained the stages that were to be executed by several processors (Sections (3.1) - (3.5)). Moreover, we chose Shared Memory to be the target architecture and the OpenMP API as the main implementation tool. In addition, we used the MORAR machine as the target system.

Then, the given parallel strategies were implemented and we developed a number of tests to verify them (correctness tests). In addition, we introduced thread-safety tests in order to check whether the functions of the original sequential code might be used simultaneously by several threads. Unfortunately, the given test failed, therefore, we used the correctness test case as a basis for the implementation of the parallel strategies.

The corresponding time costs for the likelihood function in these test cases were artificially increased, in order to make this function's behaviour similar to the real one. That is, we introduced static (constant) delays, hence each thread's call to the given function was delayed by a constant period of time. Next, we conducted series of the performance

tests to receive the speed-ups of the parallel strategies (Sections (5.1 - 5.2)). The performances of the strategies were analysed and compared with each other. According to the results, Strategies 2 and 5 showed the best results (their performances were almost identical), meanwhile the performance of Strategy 4 was rather decent. The remaining strategies showed relatively modest speed-ups.

The next set of tests implied the use of unbalanced delays as this is more in line with the realistic situation. Moreover, we modified the first, the second and the fourth strategies to try to overcome the load imbalance implied by the unbalanced delays. That is, we introduced the factor of k that corresponds to the enlargement of the amount of work executed by the threads.

Thus, by analogy with the previous tests, we obtained the performances of each parallel strategy in the given conditions. In general, the results were relatively the same as Strategies 2 and 5 remained supreme ones, while the first and the fourth strategies showed weaker speed-ups (Section (5.2)). At the same time, the performances of all strategies, except the fifth one, turned out to be weaker for the case with unbalanced delays than with static ones. Hence, in the given case the fifth strategy showed the best speed-up.

6.1 Problems encountered

The most vexing problem in this project was that the original likelihood function was not thread-safe. Although, we managed overcome this issue to some extent, considerable time was spent on attempts to make this function to be thread-safe. Instead of wasting time, we could have concentrated on different aspects (e.g. implementation of different correctness tests).

Furthermore, the installation of the "CosmoNest" application on the MORAR system proved to be cumbersome. Since the given program uses several additional libraries that should be installed as well, we had to deal with portability issues that emerged during the installation process. Moreover, the provided original source code contained some bugs and, therefore, the initial runs of the "CosmoNest" crashed.

In general, we were unfamiliar with the provided code as well as with the scientific area of Cosmology that it tackled. In addition, the original program was a part of another "CosmoMC" application that required some time to study. Though, we have been given the corresponding documentation, it took considerable time become familiar with it. Furthermore, the task of integrating "CosmoNest" and "CosmoMC" applications was complicated by the incompatibility of their versions. That is, each of these programs are developed independently from each other and hence modifications necessary for their integration have not been made.

6.2 Future work

First of all, it is necessary to carry out the modification of the original sequential code, in order to make the likelihood function thread-safe. In fact, it is a prerequisite for the exploiting of the developed parallel strategies. This task is quite time consuming as it is considered with the debugging of a program, which is based on work with random numbers. In addition, it is complicated by the fact that the corresponding functionality is contained in a large number of source files that are related with "CosmoMC". Furthermore, we faced the incompatibility of "CosmoMC" and "CosmoNest". Due to the fact that the versions of these applications were developed independently from each other, the modifications that are required for their compatibility were not made. Hence, the integration of these programs turned out to be rather cumbersome task.

Moreover, we have not conducted any tests to check whether the developed parallel strategies may introduce sampling bias. In particular this applies to the second strategy, since it is based on the approach that implies the replacement of a number of points from a particular region. In addition, we run the developed parallel strategies using likelihood function in the trivial form of Gaussian. Therefore, their behaviour should be studied in the other test cases as well.

Furthermore, the strategies that were presented in the given project are of course not the only possible ones. It is likely that there might be more elegant and efficient solutions that exploit approaches that we did not consider.

Appendix A

Run time results for the parallel strategies

A.1 Results for parallel strategies with static delays

The average results of 5 runs using the delay value 10 milliseconds.

A.1.1 Results for Strategy 1

Threads	Av. run time(sec)	Av. number of evaluations	Speed-up	Algorithmic speed-up
1	2139	213241	1.00	1.00
2	1099	218503	1.95	1.95
4	561.6	222327	3.81	3.84
8	304.5	239154	7.03	7.13
16	172.3	216337	12.4	15.7
32	114.0	344684	18.8	19.8
64	85.91	498975	24.9	27.3

Table A.1: Run time results for Strategy 1 with s=3% and static delays.

Threads	Av. run time(sec)	Av. number of evaluations	Speed-up	Algorithmic speed-up
1	742.78	73957	1.00	1.00
2	396.67	78547	1.87	1.88
4	218.93	85876	3.39	3.44
8	134.95	104393	5.50	5.67
16	92.82	140972	8.00	8.39
32	77.35	230713	9.60	10.26
64	75.55	439596	9.83	10.77

Table A.2: Run time results for Strategy 1 with s=9% and static delays.

Threads	Av. run time(sec)	Av. number of evaluations	Speed-up	Algorithmic speed-up
1	229.01	22726	1.00	1.00
2	135.84	26550	1.69	1.71
4	91.71	35203	2.50	2.58
8	76.40	57822	3.00	3.14
16	72.65	86927	3.15	4.18
32	72.95	216978	3.14	3.35
64	75.65	440095	3.03	3.30

Table A.3: Run time results for Strategy 1 with s=30% and static delays.

A.1.2 Results for Strategy 2

Threads	Av. run time(sec)	Av. number of evaluations	Speed-up	Algorithmic speed-up
1	2059.1	214852	1.00	1.00
2	1046.7	215907	1.97	1.99
4	544.02	215061	3.78	4.00
8	259.94	215368	7.92	7.98
16	140.15	216116	14.7	15.9
32	72.52	216560	28.4	31.8
64	37.88	218848	54.4	62.8

Table A.4: Run time results for Strategy 2 with s=3% and static delays.

Threads	Av. run time(sec)	Av. number of evaluations	Speed-up	Algorithmic speed-up
1	747.9	72356	1.00	1.00
2	373.4	72484	2.00	2.00
4	190.2	73759	3.93	3.92
8	94.95	72421	7.88	7.99
16	49.65	72566	15.1	16.0
32	26.45	73005	28.3	31.7
64	15.14	72960	49.4	63.5

Table A.5: Run time results for Strategy 2 with s=9% and static delays.

Threads	Av. run time(sec)	Av. number of evaluations	Speed-up	Algorithmic speed-up
1	221.1	21664	1.00	1.00
2	117.5	22628	1.88	1.91
4	59.69	22290	3.71	3.89
8	30.61	22235	7.23	7.79
16	16.54	21686	13.4	15.9
32	10.08	21830	21.9	31.8
64	6.511	21696	34.0	63.9

Table A.6: Run time results for Strategy 2 with s=30% and static delays.

A.1.3 Results for Strategy 3

Threads	Av. run time(sec)	Av. number of evaluations	Speed-up	Algorithmic speed-up
1	758.9	75533	1.00	1.00
2	684.2	135690	1.11	1.11
4	696.1	275206	1.09	1.10
8	685.1	538874	1.11	1.12
16	693.9	1081074	1.09	1.12
32	706.1	2145004	1.08	1.13
64	771.2	4337081	0.98	1.12

Table A.7: Run time results for Strategy 3 with s=9% and static delays.

A.1.4 Results for Strategy 4

Threads	Av. run time(sec)	Av. number of evaluations	Speed-up	Algorithmic speed-up
1	2131	211977	1.00	1.00
2	1083	214923	1.97	1.97
4	561.8	221734	3.79	3.82
8	275.5	215184	7.74	7.88
16	154.1	236240	13.8	14.36
32	83.36	245645	25.6	27.61
64	57.93	312205	36.8	43.45

Table A.8: Run time results for Strategy 4 with s=3% and static delays.

Threads	Av. run time(sec)	Av. number of evaluations	Speed-up	Algorithmic speed-up
1	749.9	74330	1.00	1.00
2	379.9	74834	1.97	1.99
4	195.4	76093	3.84	3.91
8	107.3	81762	6.99	7.27
16	56.88	83178	13.2	14.3
32	34,00	92192	22.1	25.8
64	28.67	126630	26.2	37.6

Table A.9: Run time results for Strategy 4 with s=9% and static delays.

Threads	Av. run time(sec)	Av. number of evaluations	Speed-up	Algorithmic speed-up
1	238.3	23340	1.00	1.00
2	126.3	24328	1.89	1.92
4	70.11	26245	3.40	3.56
8	39.98	28440	5.96	6.57
16	23.76	30864	10.0	12.1
32	14.25	31059	16.73	24.1
64	12.01	42010	19.85	35.6

Table A.10: Run time results for Strategy 4 with s=30% and static delays.

A.1.5 Results for Strategy 5

Threads	Av. run time(sec)	Av. number of evaluations	Speed-up	Algorithmic speed-up
2	2161	214999	1.00	1.00
4	671.48	199725	3.22	3.23
8	314.22	216926	6.88	6.94
16	145.4	212737	14.9	15.2
32	72.32	213900	29.9	31.2
64	36.92	212491	58.6	63.7

Table A.11: Run time results for Strategy 5 with s=3% and static delays.

Threads	Av. run time(sec)	Av. number of evaluations	Speed-up	Algorithmic speed-up
2	718.7	71462	1.00	1.00
4	252.9	74836	2.84	2.86
8	109.8	74585	6.55	6.71
16	52.85	74610	13.6	14.4
32	27.62	76090	26.0	29.1
64	14.99	75116	47.9	59.9

Table A.12: Run time results for Strategy 5 with s=9% and static delays.

Threads	Av. run time(sec)	Av. number of evaluations	Speed-up	Algorithmic speed-up
2	233.3	23196	1.00	1.00
4	79.26	23042	2.94	3.02
8	35.70	23043	6.53	7.05
16	17.92	22518	13.0	15.5
32	9.950	22470	23.4	32.0
64	6.668	23070	35.0	63.3

Table A.13: Run time results for Strategy 5 with s=30% and static delays.

A.2 Results for modified parallel strategies with unbalanced delays

The average results of 5 runs using the standard value of sampling efficiency s=9%.

A.2.1 Results for Strategy 1

Threads	Av. run time(sec)			Speed-up		
	$k=1$	$k=2$	$k=4$	$k=1$	$k=2$	$k=4$
1	701.6	724.6	810.7	1.00	1.00	1.00
2	483.3	487.1	525.9	1.45	1.49	1.54
4	322.7	321.6	385.3	2.17	2.25	2.10
8	218.5	237.4	336.8	3.21	3.05	2.41
16	159.2	207.5	318.4	4.41	3.49	2.55
32	148.9	209.0	342.2	4.71	3.47	2.37
64	135.8	225.5	355.1	5.17	3.21	2.28

Table A.14: Run time results for Strategy 1 with different values of k and unbalanced delays.

The following results were obtained using the value factor k=1.

Threads	Av. run time(sec)	Speed-up
1	2028	1.00
2	1399	1.45
4	854.6	2.37
8	508.6	3.99
16	318.8	6.36
32	214.8	9.44
64	177.8	11.4

Table A.15: Run time results for Strategy 1 with s=3% and unbalanced delays.

Threads	Av. run time(sec)	Speed-up
1	216.4	1.00
2	167.9	1.29
4	135.1	1.60
8	121.6	1.78
16	129.3	1.67
32	139.0	1.56
64	150.1	1.44

Table A.16: Run time results for Strategy 1 with s=30% and unbalanced delays.

A.2.2 Results for Strategy 2

Threads	Av. run time(sec)			Speed-up		
	$k=1$	$k=2$	$k=4$	$k=1$	$k=2$	$k=4$
1	724.8	710.6	695.8	1.00	1.00	1.00
2	447.9	409.5	382.4	1.62	1.74	1.82
4	275.2	233.0	199.7	2.63	3.05	3.48
8	139.2	124.6	104.9	5.21	5.70	6.63
16	86.58	69.07	56.12	8.37	10.3	12.4
32	49.61	36.17	30.07	14.6	19.7	23.1
64	27.23	20.78	16.74	26.6	34.2	41.6

Table A.17: Run time results for Strategy 2 with different values of k and unbalanced delays.

The following results were obtained using the value factor k=4.

Threads	Av. run time(sec)	Speed-up
1	2012	1.00
2	1129	1.78
4	577.4	3.48
8	297.8	6.76
16	161.7	12.4
32	83.05	24.2
64	43.24	46.5

Table A.18: Run time results for Strategy 2 with s=3% and unbalanced delays.

Threads	Av. run time(sec)	Speed-up
1	213.1	1.00
2	117.0	1.82
4	65.04	3.28
8	34.18	6.24
16	18.59	11.5
32	10.77	19.8
64	7.262	29.4

Table A.19: Run time results for Strategy 2 with s=30% and unbalanced delays.

A.2.3 Results for Strategy 4

Threads	Av. run time(sec)			Speed-up		
	$k=1$	$k=2$	$k=4$	$k=1$	$k=2$	$k=4$
1	710.0	746.8	801.2	1.00	1.00	1.00
2	459.1	428.6	473.0	1.55	1.74	1.69
4	291.0	259.5	280.2	2.44	2.88	2.86
8	168.7	152.8	157.1	4.21	4.89	5.10
16	103.1	89.37	93.66	6.88	8.36	8.55
32	61.41	53.48	54.89	11.6	14.0	14.6
64	45.18	39.11	37.36	15.7	19.1	21.5

Table A.20: Run time results for Strategy 4 with different values of k and unbalanced delays.

The following results were obtained using the value factor k=4.

Threads	Av. run time(sec)	Speed-up
1	2202	1.00
2	1202	1.83
4	634.5	3.47
8	362.2	6.08
16	205.8	10.7
32	117.3	18.8
64	78.60	28.0

Table A.21: Run time results for Strategy 4 with s=3% and unbalanced delays.

Threads	Av. run time(sec)	Speed-up
1	341.7	1.00
2	212.5	1.61
4	131.7	2.59
8	77.40	4.41
16	17.92	7.40
32	26.98	12.7
64	19.87	17.2

Table A.22: Run time results for Strategy 4 with s=30% and unbalanced delays.

A.2.4 Results for Strategy 5

Threads	Av. run time(sec)	Speed-up
2	691.3	1.00
4	241.5	2.86
8	106.0	6.52
16	50.95	13.6
32	26.53	26.1
64	14.56	47.5

Table A.23: Run time results for Strategy 5 with s=9% and unbalanced delays.

Threads	Av. run time(sec)	Speed-up
2	2089	1.00
4	686.8	3.00
8	301.3	6.93
16	141.4	14.8
32	68.55	30.5
64	35.85	58.3

Table A.24: Run time results for Strategy 5 with s=3% and unbalanced delays.

Threads	Av. run time(sec)	Speed-up
2	222.3	1.00
4	76.64	2.90
8	34.12	6.52
16	17.31	12.9
32	9.974	22.3
64	6.565	33.9

Table A.25: Run time results for Strategy 5 with s=30% and unbalanced delays.

Bibliography

[1] R. Trotta, "Bayes in the sky: Bayesian inference and model selection in cosmology," *Contemporary Physics*, vol. 00, pp. 1–41, March 28 2008. http://arxiv.org/pdf/0803.4089v1.pdf?

[2] European Space Agency (ESA) Planck space telescope project. http://www.esa.int/Our_Activities/Space_Science/Planck.

[3] A. R. Liddle, "Statistical methods for cosmological parameter selection and estimation," *Annual Reviews of Nuclear and Particle Science (ARNPS)*, vol. 59, 24 Mar 2009. http://arxiv.org/pdf/0903.4210v1.pdf.

[4] M. P. Hobson, A. H. Jaffe, A. R. Liddle, M. Pia, and P. David, *Bayesian Methods in Cosmology*. Cambridge University Press, 2010.

[5] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equation of state calculations by fast computing machines," *Journal of Chemical Physics*, vol. 21, pp. 1087–1092, June 1953. http://www.stat.cmu.edu/~acthomas/724/Metropolis.pdf.

[6] W. K. Hastings, "Monte carlo sampling methods using markov chains and their applications," *Biometrika*, vol. 57, pp. 97–109, April 1970. http://www.isds.duke.edu/~scs/Courses/Stat376/Papers/Basic/Hastings1970.pdf.

[7] J. Skilling, "Nested sampling for general bayesian computation," *Bayesian Analysis*, vol. 1, no. 4, pp. 833–860, 2006. http://www.inference.phy.cam.ac.uk/bayesys/nest.pdf.

[8] A. Liddle, P. Mukherjee, and D. Parkinson, "Model selection in cosmology," *Astronomy & Geophysics*, vol. 47, no. 4, pp. 4.30–4.33, August 2006. http://dx.doi.org/10.1111/j.1468-4004.2006.47430.x.

[9] F. Feroz, M. Hobson, and M. Bridges, "Multinest: an efficient and robust bayesian inference tool for cosmology and particle physics," *Mon. Not. Roy. Astron. Soc.*, vol. 398, pp. 1601–1614, 2009. http://arxiv.org/pdf/0809.3437v1.pdf.

[10] CosmoNest Project. http://www.cosmonest.org/.

[11] Cosmological Monte Carlo (CosmoMC) Project. http://cosmologist.info/cosmomc/.

[12] A. Lewis, "Efficient sampling of fast and slow cosmological parameters," *Phys. Rev.*, vol. D87, 2013. http://arxiv.org/pdf/1304.4473v2.pdf.

[13] Wilkinson Microwave Anisotropy Probe Project by NASA. http://map.gsfc.nasa.gov/.

[14] T. G. Mattson, B. A. Sanders, and B. L. Massingill, *Patterns for Parallel Programming*. Addison-Wesley, 2005.

[15] HPC Architectures, "Parallel architectures," *Course slides for MSc in HPC program, EPCC, The University of Edinburgh*, 2012.

[16] Official OpenMP website. http://openmp.org/wp/.