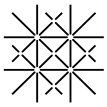


University of Basel

Dynamic loop self-scheduling on distributed-memory systems with Python

Master Research Project

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
High Performance Parallel And Distributed Computing
https://hpc.dmi.unibas.ch

Examiner: Prof. Dr. Florina M. Ciorba
Supervisor:
Dr. Ahmed Eleliemy

Abhilash Mendhe
abhilash.mendhe@stud.unibas.ch
20-067-674

Abstract

Loops are considered the primary source of parallelism in various scientific applications. Scheduling loop iterations across multiple computing resources is a challenging task, i.e., the execution must be balanced across all computing. Several factors can hinder such a balanced execution, and consequently, degrade application performance. Specifically, problem characteristics, non-uniform input data sets, as well as algorithmic and systemic variations lead to different execution times of each loop iteration. Dynamic loop self-scheduling (DLS) techniques mitigate such factors. DLS techniques were originally devised for shared-memory systems. A recently developed MPI library, called LB4MPI enables the use of various DLS techniques on distributed-memory systems. LB4MPI has two versions: one for C and one for Fortran programs. C and Fortran are often used to write scientific applications such as weather forecasting and N-body simulations. At the same time, Python has emerged over the last couple of decades as a first-class data science tool. This project aims to design and implement a Python version or interface for the existing LB4MPI library. The experiments shows the comparison of the results of C, Python and Cython on Mandelbrot set calculation for all DLS techniques. The results shows the best techniques found that can be comparable by C, Python and Cython.

Table of Contents

1 Introduction

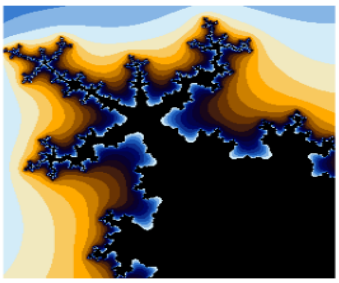
In today's contemporary world, modern problems require modern solutions. We live in a continuously evolving era where problems are exponentially arising, and these problems are not effortlessly solvable. These problems require supercomputers (HPC or high-performance computing) that are executed on massively parallel applications. These problems need complex calculations and have independent loop iterations which can be computed parallelly. Here parallelism plays an important role and with parallel computing, results are obtained faster. But such problem also have irregular loops that does not fully support parallelism. *Therefore, simply parallelizing application is enough? Is there a way to optimize the performance of a parallel application?*

1.1 Applications with independent loop iterations

1.1.1 Mandelbrot Set

- **Mandelbrot Set**

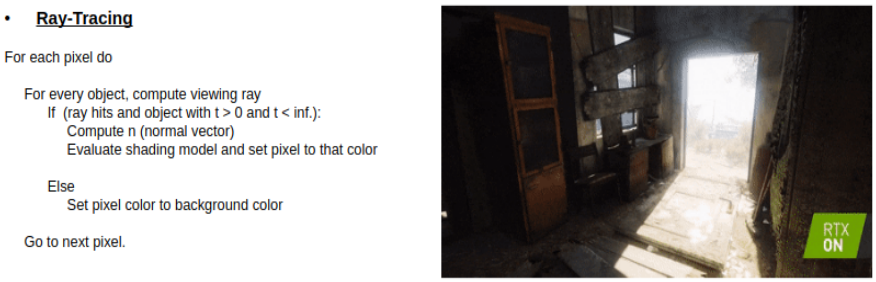
```
For each pixel p of the image:
   Let c be the complex number represented by p
   Let z be a complex variable equal to 0

   While (upto N iterations):
      If |Z| > 4.0:
         Set color the pixel white, end loop and go to next pixel

      Else:
         Set z = z^2 + c

   If while loop ends, then color pixel p to black.

   Go to next pixel.
```



Mandelbrot set is a shape generated from fractal geometry. Unlike classic geometry where it has smooth shapes and curves for e.g. square, circle, triangle etc, fractal geometry are rough and infinitely complex. Fractals have shown their usability in a wide range of domains from Biology and Medicine, image processing, art etc. In biology it explores the potential of fractal geometry for describing and understanding biological organisms, their development and growth as well as their structural design and functional properties. In medicine it helps to contribute to the understanding of pathogenetic processes in medicine.

Above on the left side is the pseudocode of the calculation of mandelbrot set and on the right side is the zoomed in image of mandelbrot set. In the psuedocode, there is an independent loop iteration that is the range from 0 to the total number of pixel values. Every pixel value is a complex number (x + iy). The magnitude of the complex values are taken and run under another loop with the maximum number of iterations. This max iteration is the cut off value which helps to determine whether the pixel value is inside the mandelbrot set or it goes to infinite. This inner loop is a irregular loop which can cause heavy load imbalance.

1.1.2 Ray Tracing



In 3D computer graphics [10], ray tracing is a technique for modeling light transport for use in a wide variety of rendering algorithms for generating digital images.

In above on the left side is the pseudocode of the ray tracing algorithm and on the right side is the image. The algorithm says that, for each pixel, we compute viewing ray. If the ray intersect with an object for t larger than 0, then we compute the normal vector, do the shading and set pixel to that color. Else we set the background color to the pixel. The inner loop causes heavy load imbalance because the inner loop recursively tries to find out the intersection of ray to each object.

1.2 Why Load Balance Is Important?

Many programmers don't think about the load imbalance while writing a parallel application. Load imbalance creates uneven scheduling of tasks or parallel loops in a program which degrades the entire performance of a parallel application. This load imbalance is a serious issue. Here we introduce the concept of load balance where it distributes the tasks evenly to all the available resources. This load balancing utilizes the resources and makes improvements in the performance of a parallel application.

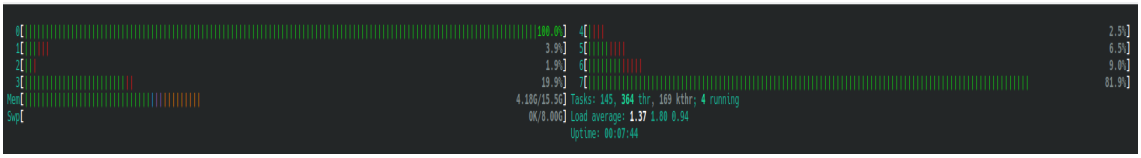


Figure 1.1: Load Imbalance

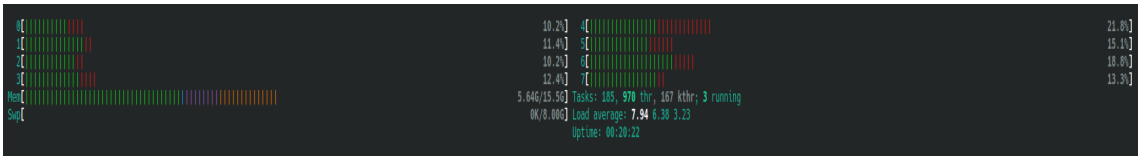


Figure 1.2: Load Balance

1.3 How to achieve load balance?

Load balancing is achieved through loop scheduling techniques. The definition of scheduling says "Ordering (organization) of parallel computations (and their associated data) in processor space and time". There are two different ways to assign the loop iterations, that is either statically or dynamically. In static, the loop iterations are divided by the number of processors to get an equal amount of chunk sizes. These chunk sizes are then distributed over processors. This technique has a low scheduling overhead but high load imbalance because the processors might finish the iterations at different times. In dynamic (self-scheduling or SS), exactly one iteration is distributed among all the available processors. This has a very high scheduling overhead but it achieves perfect load balance. It is better than the static technique. More scheduling techniques are discussed in chapter 2.

There is standardized way to parallelize the code and using certain scheduling techniques. Programming can be done on both shared memory using **OpenMP** and on distributed memory using **MPI**.

This project focuses on scheduling loop techniques on distributed memory using **MPI** using Python. We implemented the DLS techniques for distributed memory on Python using MPI4PY.

1.4 Why Python is on rise?

Python has come into limelight since 2010 and it is still on the rise. Today it is the most preferred programming language because not only it is flexible and easy to learn but it has that ability to speak to the user. Python is very versatile language, from deploying web server application to writing scientific computing application, one can easily do that with Python. Python supports object-oriented approach which makes to write clear and logical code. Python has vast libraries like *numpy, scipy, django, matplotlib, pandas, scikit-learn, tensorflow etc.*

Numpy and SciPy are used for writing scientific applications where as Django is used in web development applications, but currently most widely used libraries are tensorflow and scikit-learn as these libraries are used for building Machine Learning and Neural Network models.

In terms of P3HPC (section 1.6), Python is portable, it can be run on any hardware platforms/architecture and in terms of productivity Python has the ability to quickly implement new applications, features and maintain existing ones. But performance wise Python is poor because it is an interpretable language. Also in Python one cannot declare datatype and

due this Python takes a bit time to learn about the datatype of a given variable. Python has both cons and pros, it depends on application type.

1.5 MPI4PY

MPI4PY is a *Message Passing Interface* for Python users. With the help of this library, Python user has a leverage to write parallel code on multiple processes where each process has its own memory. Thus working of tasks parallelly in isolation manner in different processes.

In MPI4PY, one can do point-to-point communication and collective communication operations. In Point-To-Point operation, data can be shared across different processes using MPI_Send() and MPI_Recv() function. In Collective communication, data can be broadcasted at once to all the process using MPI_Bcast() function. Scattering and gathering of data is also a part of collective communication. With, MPI4PY, one can also do advance collective communication operations like MPI_Reduce(). MPI_Reduce() takes an array of input elements on each process and returns an array of output elements to the root process. The output elements contain the reduced result.

In chapter 3, we discuss about the history and the related work about the load balancing library.

1.6 P3HPC

Performance, Portability, and Productivity in HPC (P3HPC) [11] events provide an opportunity for researchers and application developers to discuss their successes and failures in tackling the compelling problems that lie at the intersection of performance, portability and productivity (P3) in High-Performance Computing (HPC).

- Performance: Running an application at a reasonable fraction of peak performance on a given hardware.
- Portability: Ability to run the code/application on different hardware platforms/architectures with minimum modifications.
- Productivity: The ability to quickly implement new applications, features and maintain existing ones.

2
Background

This chapter gives a short description about Loop Scheduling and all the DLS techniques implemented in the library.

2.1 Loop Scheduling

Loops are finite sequence of instruction are run until a specific conditions are met. Scientific applications containing large loops are time consuming, but parallelizing those loops can significantly increase performance and reduce the execution of time.

Scheduling is the ordering of computation and data in space and time. In that case, distribution of loops in processor units are called loop scheduling. Sometimes the distribution of loops causes a severe load imbalance and then this lead to overhead in computation. To reduce the overhead, few load balancing techniques were introduced. There are two types of load balancing that are, **static** and **dynamic load balancing** via **loop scheduling techniques**.

2.2 Static Scheduling

In STATIC scheduling [1], the chunks are precomputed and are assigned to each processing elements or workers. The task division and assignment do not change during execution.

2.2.1 STATIC

In STATIC chunking, the loop iterations are divided into P equal sized chunks, where P is number of processing elements or MPI ranks.

$$Chunks = \frac{No.of Iterations}{PUs} \tag{2.1}$$

This technique has a low scheduling overhead because of the minimum number of fixed chunks each processor gets. The fixed chunks may give rise in severe load imbalance. Iteration execution times vary.

It can provide good load balancing only if the iterate times are constant and the processors are homogeneous and equally loaded.[5]

2.3 Dynamic Scheduling

In DYNAMIC scheduling [1], the division and execution of tasks are carried during the execution of program. The dynamic loop self-scheduling can be categorized as adaptive and non-adaptive.

2.3.1 Non-Adaptive Technique

This technique calculated the chunks based on certain parameters which are obtained prior the execution of an application.

2.3.1.1 SS

In SS chunking, each processing elements or MPI rank gets only one chunk size out of number of iterations.

$$Chunks = 1 \tag{2.2}$$

This technique perfectly balances the load but with a cost. It has high scheduling overhead. Each processing element requests for the chunk size after completion of previous chunk size and because of chunk size being equal to 1, it has high scheduling overhead.

2.3.1.2 FSC

In *Fixed-Size Chunking* (FSC)[15], it is calculated with a given formula

$$Chunks = \left(\frac{\sqrt{2}.N.h}{\sigma.P.\sqrt{\log P}} \right)^{\frac{2}{3}} \tag{2.3}$$

where,
h = overhead time,

σ = S.D. iterate execution time

With the fixed chunk size, this technique singularly reduces the scheduling overhead while still providing a better load balance.

2.3.1.3 mFSC

In *modified FSC* (mFSC), the number of scheduling events are same as *Factoring* FAC technique, that is it has same number of chunks but with different chunk sizes. [2]

2.3.1.4 GSS

In *Guided Self-Scheduling* (GSS) [18], it uses different style of scheduling that is decreasing chunk sizes across the processing elements or MPI ranks. This technique also provide a good load balance with less scheduling overhead. At every scheduling step, GSS assigns a chunk that is equal to the number of remaining loop iterations divided by the total number of processing elements.[7]

$$Chunks_i = \lceil \frac{R_i}{P} \rceil \tag{2.4}$$

$$R_i = N - Chunks_i \tag{2.5}$$

where,
N = Total number of iterations
P = Number of processing elements or MPI ranks
R_i = Remaining iterations

2.3.1.5 TSS

In *Trapezoid Self-Scheduling* (TSS) [20], it uses the same style of scheduling as GSS uses, that is decreasing chunk sizes across the processing elements.

$$K_i^{TSS} = K_{i-1}^{TSS} - \lfloor \frac{K_0^{TSS} - K_{S-1}^{TSS}}{S-1} \rfloor \tag{2.6}$$

$$S = \lceil \frac{2.N}{K_0^{TSS} + K_{S-1}^{TSS}} \rceil \tag{2.7}$$

$$K_0^{TSS} = \lfloor \frac{N}{2.P} \rfloor \tag{2.8}$$

$$K_{S-1}^{TSS} = 1 \tag{2.9}$$

The TSS uses linear chunk function, which makes the TSS simple enough to be implemented.[20]

2.3.1.6 FAC

In *Factoring* (FAC), it uses the same style of scheduling as GSS and TSS uses, that is decreasing chunk sizes across the processing elements. Moreover it is a better version than GSS significantly dropping the scheduling overhead and perfectly balancing the loads across processing elements. [12]

The chunk calculation is based on probabilistic analyses using the prior knowledge of the μ (mean) and the σ (S.D.) of the loop iterations execution times.

In contrast to earlier methods, this technique schedules iterations in batches of P equal size chunks.[12]

$$Chunks_i = \lceil \frac{R_i}{x_i.P} \rceil \tag{2.10}$$

$$R_0 = N \tag{2.11}$$

$$R_{i+1} = R_i - P * Chunks_i \tag{2.12}$$

$$b_i = \frac{P}{2\sqrt{R_i}}.\frac{\sigma}{\mu} \tag{2.13}$$

$$x_0 = 1 + b_0^2 + b_0\sqrt{b_0^2 + 2} \tag{2.14}$$

$$x_i = 2 + b_i^2 + b_i\sqrt{b_j^2 + 4} \geq 0 \tag{2.15}$$

where,
i = batch index.

One batch is calculated and placed after the previous batch is scheduled.

2.3.1.7 WF

In *Weighted Factoring* (WF)[13], the batch and chunk size are calculated like in *Factoring* (FAC) method. Here, each processor is associated with a weight **w** that represent the relative speeds. It dynamically assigns the decreasing size chunks of iterations to the processor elements.

$$Chunks_{ij} = w_i * FAC\_Chunks_j \tag{2.16}$$

$$\sum_{i=1}^{P} w_i = P \tag{2.17}$$

where,

i = i^th processing element

j = batch chunk index

2.3.1.8 TAP

TAPER (TAP) [16, 7] is a probabilistic model that considers the average of loop iterations execution time μ and σ to achieve a higher load balance than GSS. The goal of any tapering method is to achieve optimally even finishing times while scheduling the smallest possible number of chunks.

$$K_i^{TAP} = K_i^{GSS} + \frac{v_\alpha^2}{2} - v_\alpha \cdot \sqrt{2.K_i^{GSS} + \frac{v_\alpha^2}{4}} \tag{2.18}$$

where,

$v_\alpha = \frac{\alpha\sigma}{\mu}$

2.3.1.9 TFSS

Trapezoid Factoring Self-Scheduling (TFSS)[6] is a trapezoid scheme with stages. The notion of the technique uses the characteristics of two scheduling techniques that are TSS and FAC. To calculate the chunk size, TFSS uses the same formula used in TSS technique. These P chunks are then carried out in stages like Factoring (FAC) manner.

$$K_i^{TFSS} = \begin{cases} \frac{\sum_{j=i}^{i+P} K_{j-1}^{TFSS}}{P}, & \text{if i mod P} = 0 \\ K_{i-1}^{TFSS}, & \text{otherwise} \end{cases} \tag{2.19}$$

2.3.1.10 FISS

Fixed Increase Self-Scheduling (FISS) [17] is a dynamic scheduling technique where the size of chunks gets increased until the remaining chunks are exhausted. In this technique the number of stages required to calculate the chunks are need to be fixed. Once the stages are fixed, the programmer then needs to select the initial chunk size. FISS depends upon the value B (*bump or stage increment*) which is defined by user.

$$K_i^{FISS} = K_{i-1}^{FISS} + \lceil \frac{2.N.(1 - \frac{B}{2+B})}{P.B.(B-1)} \rceil \tag{2.20}$$

where,

$K_0^{FISS} = \frac{N}{(2+B).P}$

2.3.1.11 VISS

Variable Increase Self-Scheduling (VISS) [17] follows the same approach like FISS technique where there is the increase in chunk size. Only difference is that the number of stages to complete all the iterations are not fixed. ted, Variable Increase increases the chunk size similar to the way Factoring decreases the chunk.

$$K_i^{VISS} = \begin{cases} K_{i-1}^{VISS} + \frac{K_{i-1}^{VISS}}{2} & \text{if i mod P} = 0 \\ K_{i-1}^{VISS} & \text{otherwise} \end{cases} \tag{2.21}$$

where,

$K_0^{VISS} = K_0^{FISS}$

2.3.1.12 RND

RND [7] technique makes use of Uniform Random Distribution to arbitrarily choose a chunk size between lower and upper bounds.

$$K_i^{RND} \in [1, \frac{N}{P}] \tag{2.22}$$

2.3.1.13 PLS

Performance-based Loop Scheduling (PLS) [19, 7] technique divides the loop into two parts. The first part of loop is scheduled statically while the the second part is scheduled dynamically. PLS uses Static Workload Ratio (SWR) to determine the amount of iterations to be statically scheduled.

$$K_i^{PLS} \begin{cases} \frac{N.SWR}{P} & \text{if } R_i > \text{N - (N.SWR)} \\ K_i^{GSS}, & \text{otherwise} \end{cases} \tag{2.23}$$

where,

$SWR = \frac{min.iterationexecutiontime}{max.iterationexecutiontime}$

2.3.2 Adaptive Technique

This technique calculates the chunks based on latest information on the state of the both application and system. Adaptive techniques are better than non-adaptive techniques.

2.3.2.1 AWF

In *Adaptive Weighted Factoring* (AWF)[5], is the adaptive version of Weighted Factoring(WF) technique where the weights are adapted during the execution or computation. No profiling is needed because AWF does not require any prior knowledge about the workloads. This technique was originally designed for executing a parallel loop in a scientific application which involves time-stepping. At every time step, AWF updates the relative processor weights w_i. Initial value of w_i is always 1.

2.3.2.2 AWF Variants

Adaptive Weighted Factoring (AWF)[5] has extended to 4 more variants that are AWF-B, AWF-C, AWF-D, and AWF-E. AWF has a drawback where the weight updation happens only after every time step where as in the variants the adaption happens during loop execution.

AWF-B(Batched - AWF) It schedules the remaining iterations by batches. Weights are updated after each batch based upon the timings of previous chunks.

AWF-C(Chunked - AWF) It schedules the remaining iterations by chunks. To overcome the drawbacks of AWF-B as well as AF, FAC and AWF scheduling techniques, AWF-C was

introduced. In AWF-B, the size of the chunks are the fractions of the current FAC batch size, once scheduled, then it cannot be changed. Due to the unchanged batch size the faster PEs gets the remaining chunks of less-than-optimal size from the current batch. Therefore AWF-C recomputes a new batch size each time a processor requests for work. With this strategy, faster processors are assigned larger chunks from all the remaining iterates, not just from the remainder of the current batch.

AWF-D It is similar to AWF-B, but the execution time is redefined to the total chunk time t_{ij}. In AWF-D, t_{ij} includes the time spent by the processor doing other tasks associated with the execution of a chunk of iterates.

AWF-E This strategy is similar to AWF-C, but using total chunk time as in AWF-D

2.3.2.3 AF

The *Adaptive Factoring* (AF) algorithm [3] is a more generalized version than FAC or WF technique. AF is similar to FAC, it also uses the probabilistic model to calculate the chunk sizes, but dynamically. Unlike FAC, where μ and σ are know prior and additionally they are same for all the PEs, where as in AF, the μ and σ are calculated and adjusted during run time. This insures a more efficient method for balancing processor workloads, highly tuned to the rate of change of processor speeds.

$$K_i^{AF} = \frac{D + 2.E.R_i - \sqrt{D^2 + 4.D.E.R_i}}{2\mu_{pi}} \tag{2.24}$$

where,

$D = \sum_{pi=1}^{P} \frac{\sigma_{pi}^2}{\mu_{pi}}$

$E = (\sum_{pi=1}^{P} \frac{1}{\mu_p i})^{-1}$

3
Related Work

In this chapter, past related work is discussed. This project work is the implementation of the existing Dynamic Loop Self-scheduling For Load Balancing (DLS4LB) library in Python. Python is the most widely used programming language for scientific computation. For performing scientific computation, we need a tool that can parallelize and execute these complex applications, and thus we can obtain results in minimal time. Dynamic Loop Self-scheduling For Load Balancing (DLS4LB) is an MPI-Based load balancing library. It is implemented in C and FORTRAN (F90) programming languages to support scientific applications executed on High-Performance Computing (HPC) systems. DLS4LB library is based on the DLB tool developed by Dr. Carino and Dr. Banicescu. It is modified and extended by Prof. Florina M. Ciorba and Dr. Ali Mohammed to support more scheduling techniques. It has 14 DLS techniques. The DLS4LB parallelizes and load balances scientific applications that contain simple parallel loops (1D loops) or nested parallel loops (2D loops). The tool employs a master-worker model where workers request work from the master whenever they become free. The master serves work requests and assigns workers chunks of loop iterations according to the selected DLS technique.
The DLS4LB in Python is an extension of the existing library. It has 6 more DLS techniques [7].

4

Design Implementation

4.1 Mandelbrot Sets

Mandelbrot sets[9] are a set of complex numbers which are generated from quadratic recurrence equation

$$f_c(z) = z^2 + C \tag{4.1}$$

where z does not tend to infinity are in the sets.

```
def calculate_mandelBrot(start, end):

    for i in range(start, end):

        # t1 = time.time()
        x = i // h_pixel
        y = i % h_pixel
        a = (x - (w_pixel/2)) / (w_pixel/4)
        b = (y - (h_pixel/2)) / (h_pixel/4)

        # c = Complex(a, b)
        c_r = a
        c_i = b
        # z = Complex(0, 0)
        z_r = 0
        z_i = 0
        it = 0
        for it in range(0,iterations):

            t = (z_r * z_r) - (z_i * z_i)
            z_i = 2.0 * z_r * z_i
            z_r = t

            t = (z_r * z_r) - (z_i * z_i)
            z_i = 2.0 * z_r * z_i
            z_r = t

            z_r = c_r + z_r
            z_i = z_i + c_i
            # z = add(z, c)

            mag =  (z_r*z_r) + (z_i*z_i)

```

```
34              if mag > 4.0:
35                  pixel_arr[i] = it
36                  break
37
38          if pixel_arr[i] == -1:
39              pixel_arr[i] = iterations
```

Image generated by the mandelbrot set code.

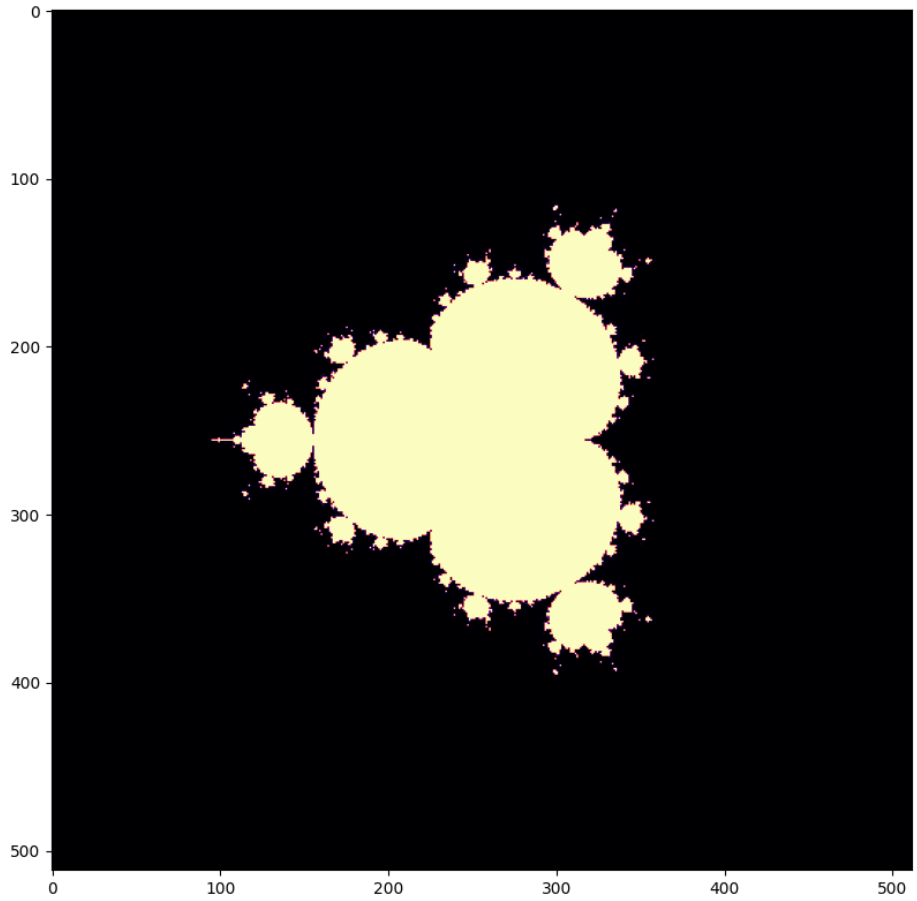


Figure 4.1: Mandelbrot Set Image

4.2 DLS4LB

Dynamic Loop Self-Scheduling for Load Balancing (DLS4LB) is an MPI-based load balancing library. The library exists in two different programming flavors that are C and Fortran. The library is now also available in Python for Python programmers with additional 6 DLS techniques.

The Python library follows the Centralized Chunk Calculation approach (Master-Worker model), where workers request chunk size whenever they are free.

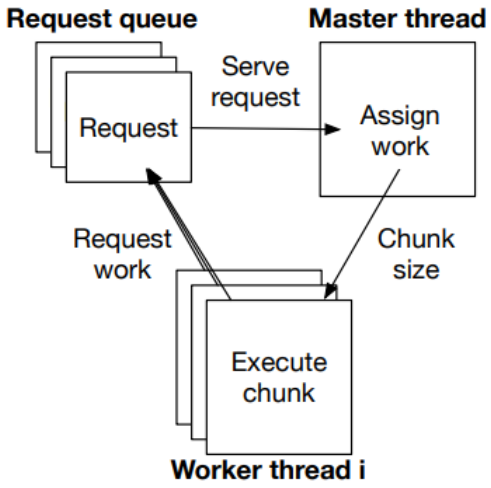


Figure 4.2: Centralize Chunk Calculation Approach[8]

This library only parallelizes and balances the load of scientific applications which contains 1-D loops. One can also parallelize 2-D loops by applying loop optimization techniques such as loop fusion technique which converts 2-D loop to 1-D loop.

4.3 How to use the library?

Listing 4.1: Usage of LB4MPI in Python

```
1  info = infoDLS(comm, size, rank, 0, total_iterations, master, method, .... )
2
3  startLoop(info)
4
5  while not DLS_Terminated(info):
6      start, chunks = DLS_StartChunk(info)
7
8      for i in range(start, start+chunks):
9          ....
10
11     DLS_EndChunk(info)
12
13 DLS_EndLoop(info)
```

Outline of each function and step wise initializing of the DLS functions.

1. First, the object for infoDLS class is created with passing the parameters. Important parameters like MPI communicator (**comm**), size of the communicator (**size**), ranks, initial start of the iterations, end of the iterations, master value and DLS method are required to fully functional of this library.

   By calling the object, it set up values of infoDLS variables and also precomputes required chunks for some DLS techniques like AWF, TSS, FISS, VISS etc.

2. Second, startLoop() function is called. It plays a major role for sending few chunks of the iterations to the available workers (MPI ranks).

3. Third, in while loop parameter we set function *DLS_Terminiated*() which return boolean value 0 or 1. This loop plays a major role in terminating the loop when there are no remaining chunk sizes.

4. Fourth, inside while loop, *DLS_StartChunk*() and *DLS_EndChunk*() functions are called.

   The *DLS_StartChunk*() returns start value and chunk size value.

5. At last, *DLS_Endloop*() is called.

4.4 Flowchart

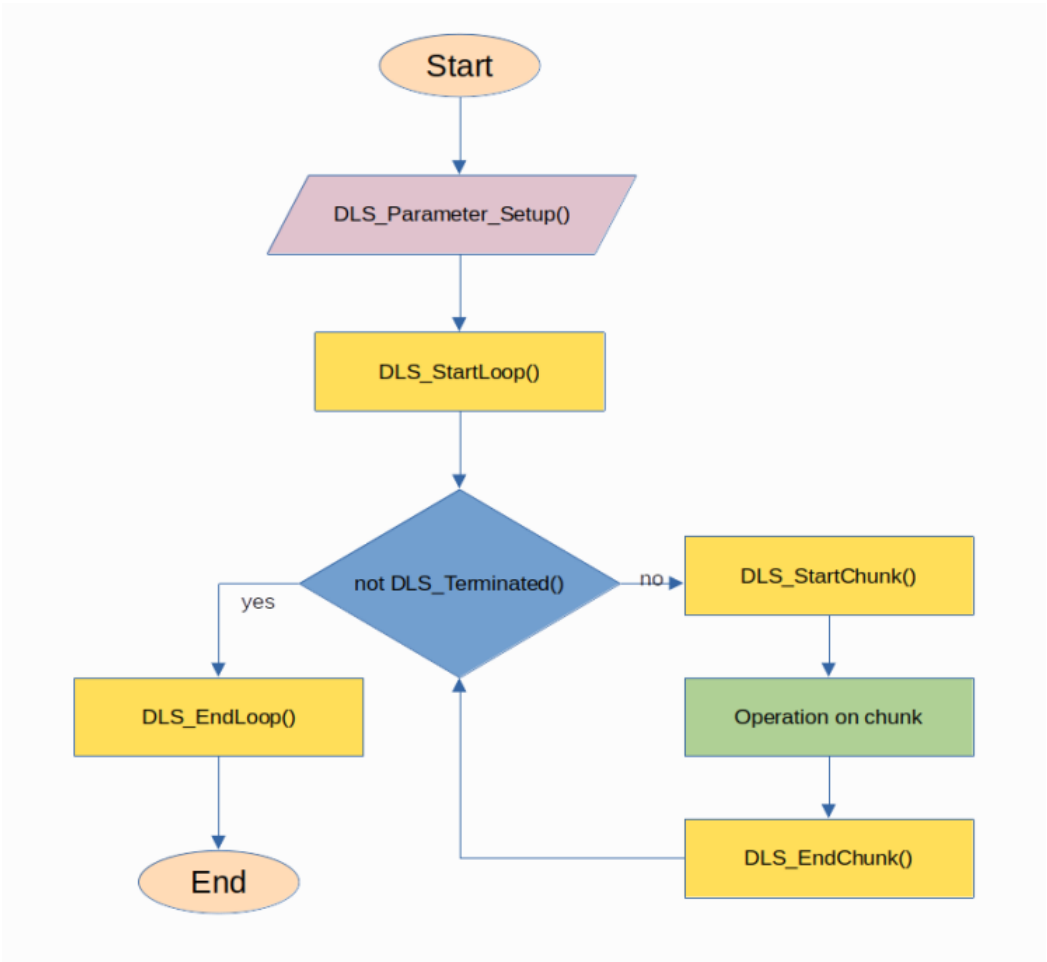


Figure 4.3: LB4MPI Single For Loop

4.4.1 DLS Parameter Setup

In this function, user needs to provide some parameters to initialize variables of a class or struct called **infoDLS** in respect to run the further function. Certain parameters like size of the communicator, range of ranks, assigning master, breakAfter value, requestWhen value, and probeFrequency are initialized in this function.
Some predetermined values are set up which are used to calculate various chunking techniques like FSC, AF, AWF etc.

4.4.2 DLS Start Loop

1. In Start loop function, some more parameters are set up before entering the loop and also sending initial work to all process.

2. Parameters like first iteration, last iteration, setting go to work to true, and initializing batch remaining, number of chunks and batch size to zero.

3. At the end of the function, work is sent to all the process from range 1 - N including MASTER rank by calling **SendChunk()** function. Distribution of chunk size is only done by the foreman (MASTER rank).

4. If the chunkstart is greater than lastiter then we terminate the worker(rank), because no chunks left to compute.

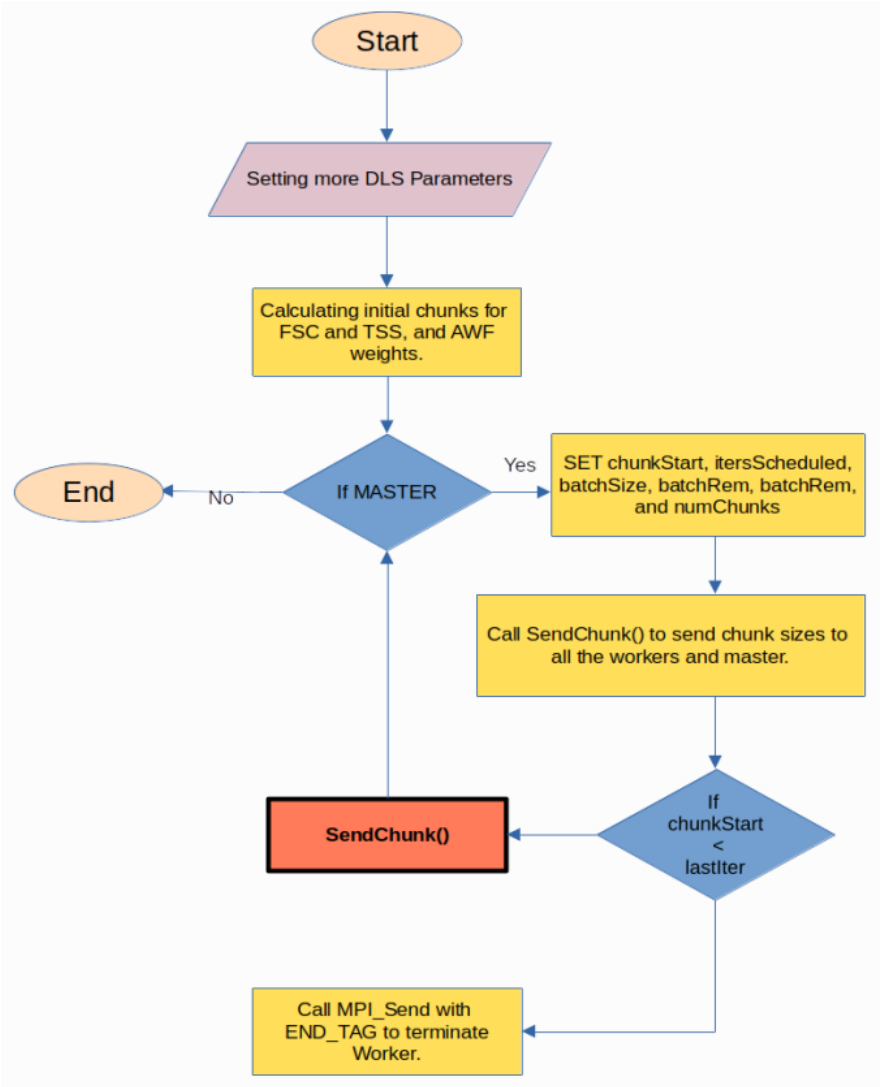


Figure 4.4: StartLoop() function

4.4.3 SendChunk() Function

1. In **SendChunk()** function another function is called **GetChunkSize()**. This function calculates the chunk size from one of the given techniques like STATIC, SS, GSS, TSS etc.

2. Now the chunk is distributed to all the ranks. Except MASTER rank, the chunk is send to all other process by calling **MPI_Send()** function with WRK_TAG.

3. After distribution of the respective chunk size, a **SetBreak()** function is called which breaks the chunk into smaller sizes to work efficiently.

4. DLS_StartLoop() function ends and a while loop starts calculating the start chunk and end chunk for the respective iterations.

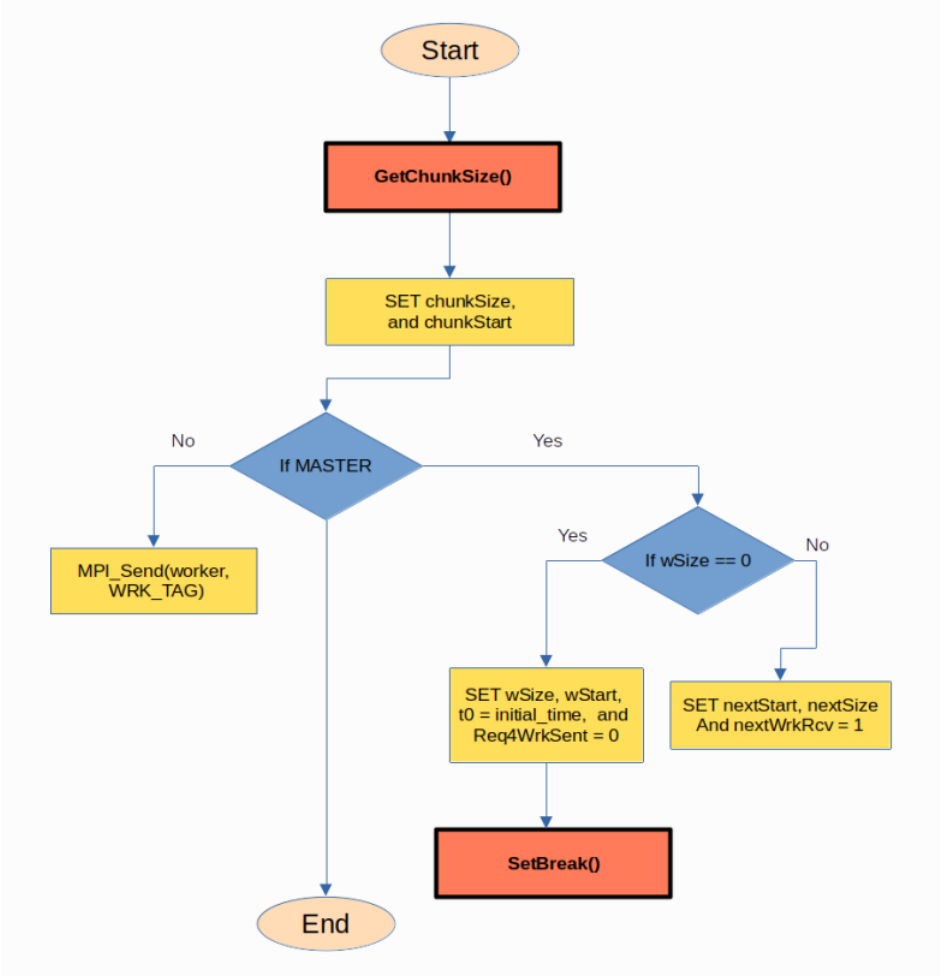


Figure 4.5: SendChunk() function.

4.4.4 SetBreak() Function

1. In SetBreak(), the smaller chunk size are calculated and stored in a variable called probFreq if breakAfter is less than 0. This is for the master worker.
2. One can set the breakAfter variable for different smaller chunk sizes.
3. For slave workers, if the requestWhen is less than 0, then the chunk sizes are break down and calculated, and are sent to the slaves.

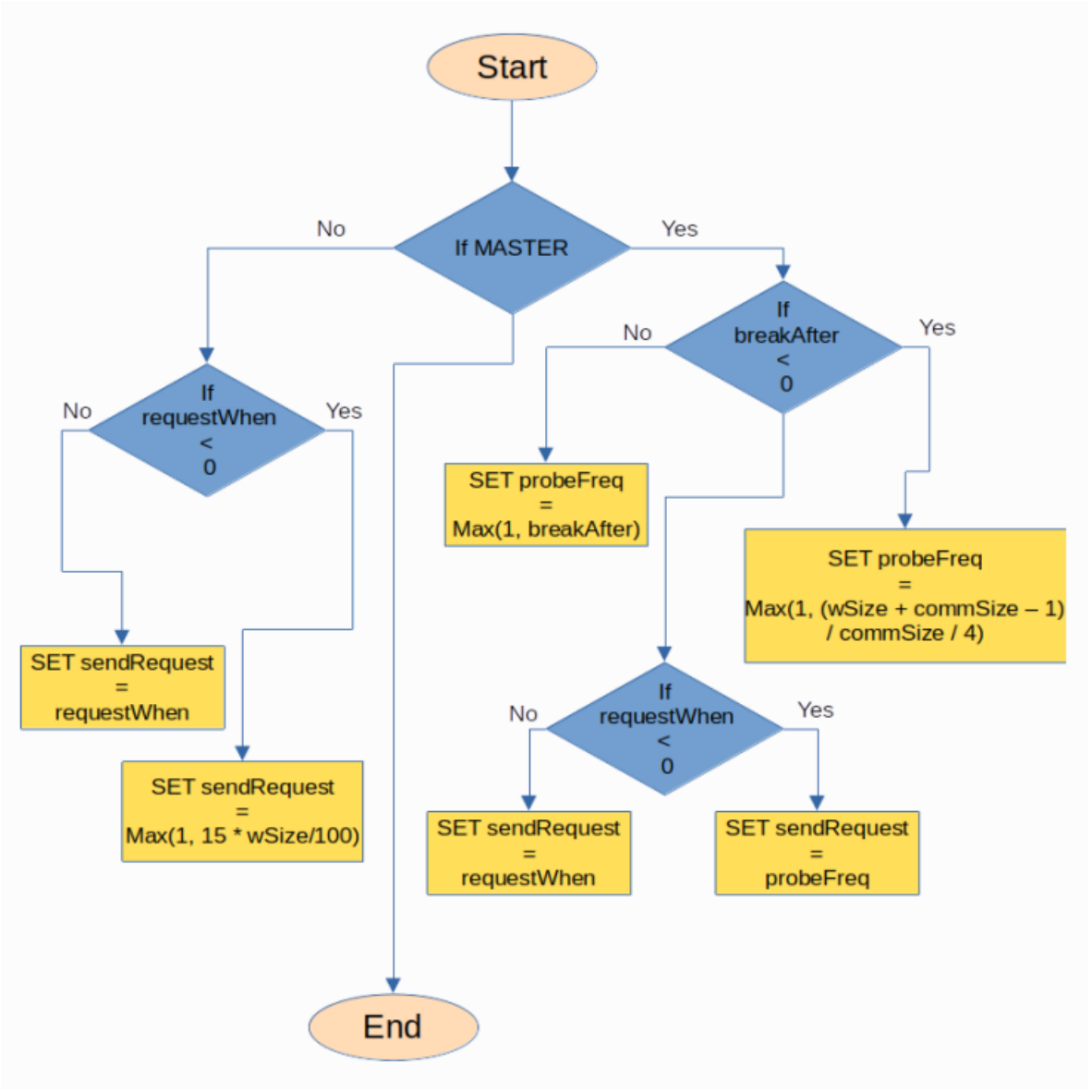


Figure 4.6: SetBreak() function.

4.4.5 DLS_Terminated

A function name **DLS_Terminated()** is passed as a parameter in the while loop. **DLS_Terminated()** plays important role to terminate the loop.

Inside **DLS_Terminated()**, two important variables name **gotWork** and **wSize** helps in taking the decision to end the loop or to continue the loop.

If both **gotWork** and **wSize** variables are zero then it terminates the loop.

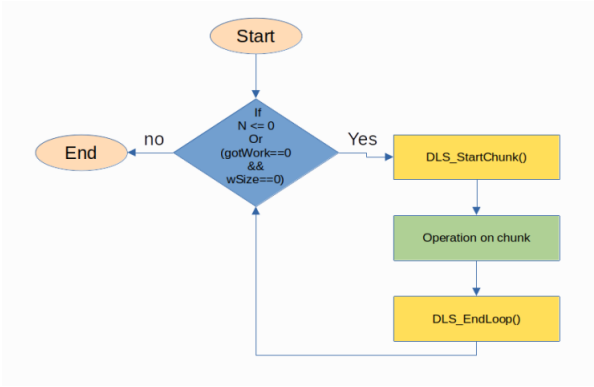


Figure 4.7: DLS_Terminated() function.

4.4.6 DLS_StartChunk

In **DLS_StartChunk()** it updates chunk size and the start value of the iteration.

This function also plays an important role in handling requests. Below are defined with respective to there TAGS.

- **WRK_TAG** - This message is only received by workers only. Distribution of chunk size takes place by setting the **wSize** and **wStart** variable of the worker process. After distribution, **SetBreak()** function is also called which breaks the chunk size into more smaller pieces.
- **REQ_TAG** - The message is only received by the foreman or MASTER. If any unfinished iterations left, then it again assigns the remaining chunk size to the requested worker and worker then starts executing it.

  If no iterates left, it then sends an end message to terminate the loop.
- **END_TAG** - This message is received by both MASTER and worker processes. This message helps in terminating the loop.

At the end of the function it also updates **chunkStart** and **chunkSize** variable to determine the next iteration.

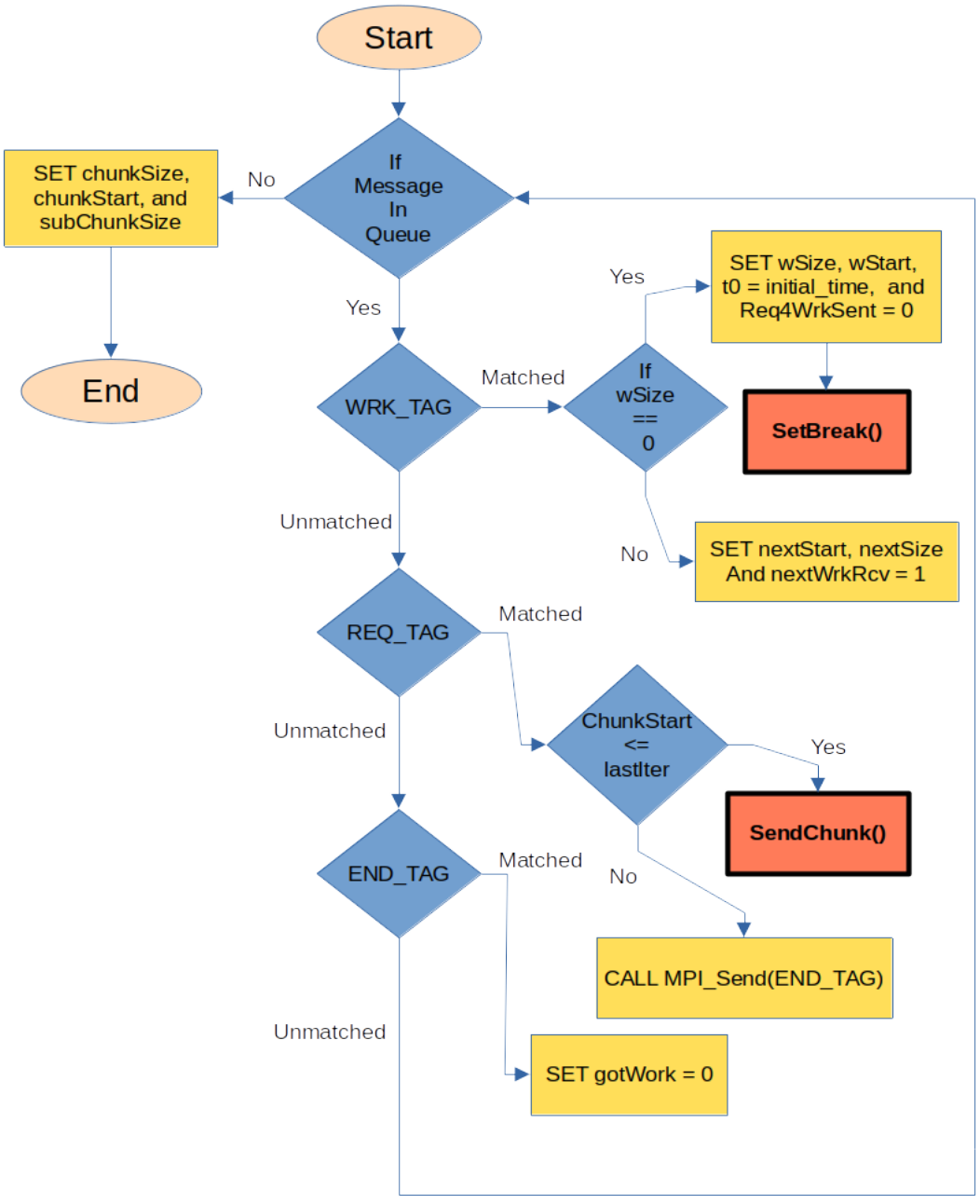


Figure 4.8: DLS_StartChunk() function.

4.4.7 DLS_EndChunk

In **DLS_EndChunk()**, it also updates the **wSize** and **wStart** variable in order to execute further iterations.

This function also keep track of the execution time of each process.

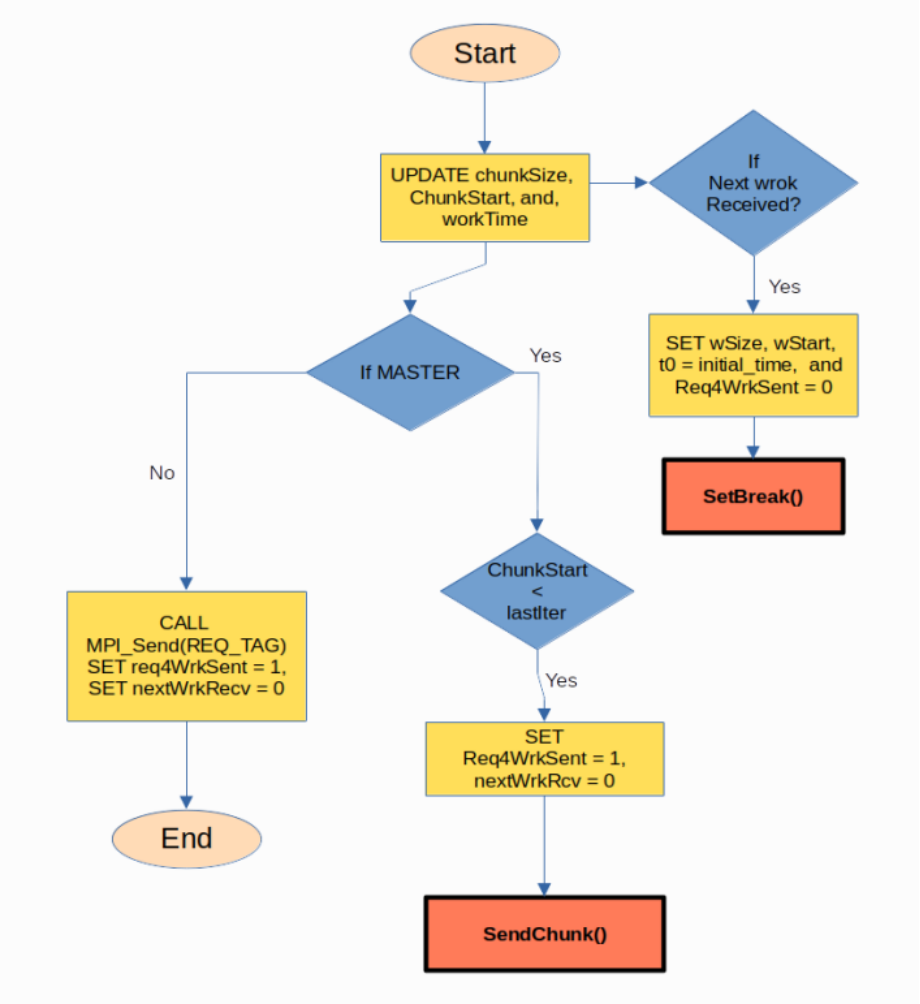


Figure 4.9: DLS_EndChunk() function.

5

Performance Evaluation and Discussion

The chapter depicts the time results of mandelbrot application implemented with all DLS (Dynamic Loop Self-Scheduling) techniques produced in C, Python and Cython.

5.1 Design of Factorial Experiments

Factors	Values	Properties
Applications	Mandelbrot	Image Size = 1024 x 1024 (Total number of loop iterations) Max iterations = 10,000
Scheduling Techniques	Static	Plain parallelization, N/P distribution of chunks
	SS, GSS FSC, mFSC TSS, FAC, WF TAP, TFSS FISS, VISS RND, PLS	Dynamic and Non-Adaptive Techniques
	AF, AWF AWF-B,C,D,E	Dynamic and Adaptive Techniques
System/Platform	miniHPC	Cores=20, Freq.=2.4GHz, Processor=Intel Xeon E5 - 2640 Memory=64GiB, Network=Intel Omni-Path 100 L1=32KB, L2=256KB, L3=25MB Number of nodes = 16 nTasks-per-node = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13 14, 15, 16 Total ranks = 16, 32, 48, 64, 80, 96, 112, 128, 144, 160 176, 192, 208, 224, 240, 256
Experiment Repeatations	10	

5.2 Relative Difference between C and Python

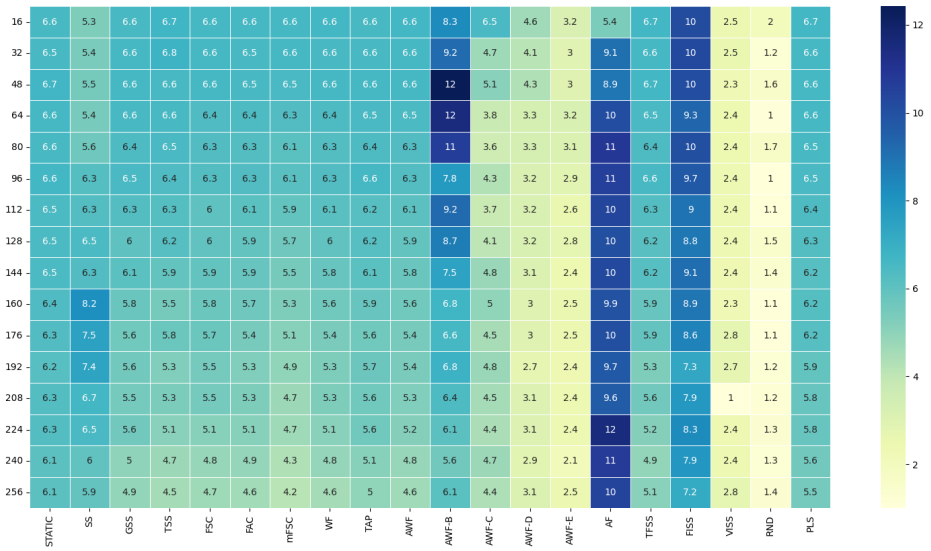


Figure 5.1: Relative Difference

5.3 Relative Difference between C and Cython

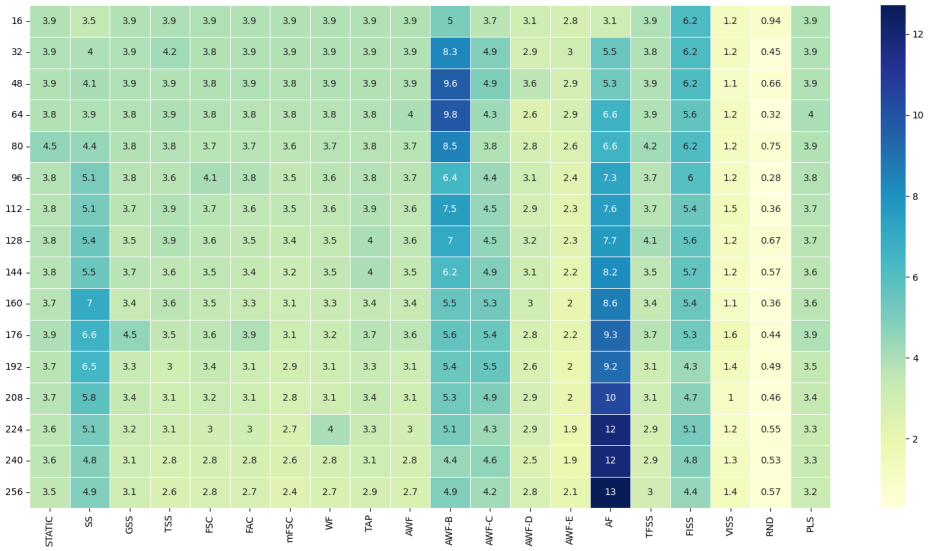


Figure 5.2: Relative Difference

5.4 Relative Difference between Python and Cython

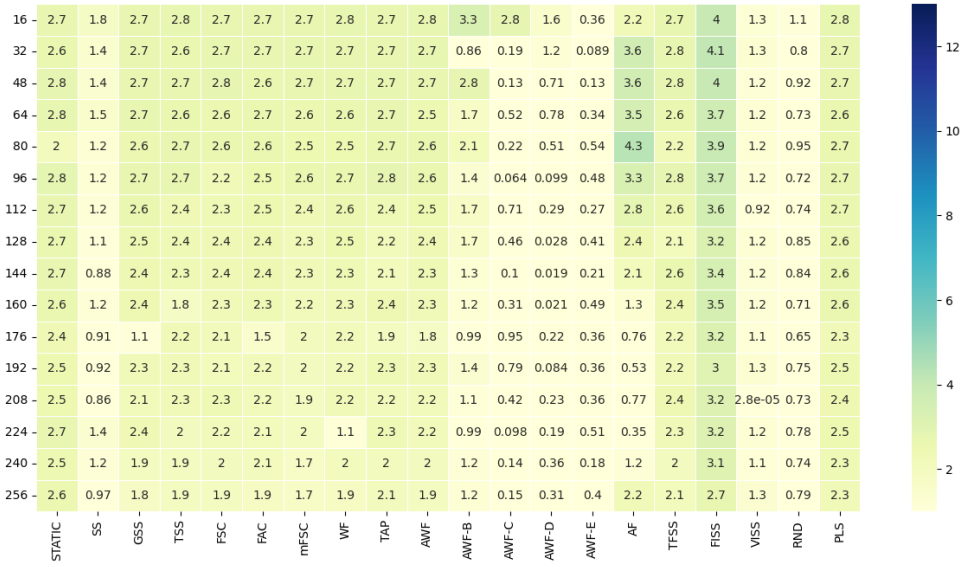


Figure 5.3: Relative Difference

For all ranks from 16 to 256, we have a graph plots for C, Python and Cython version. We have observed that for all the tests ranks and for all the programming version, STATIC has performed the worst. Parallel per execution time is the highest for STATIC. Another thing we observe that AWF-D and FISS are the second worst technique to perform poor. All other techniques achieve comparable performance.

From rank 128 onwards, we observe some unexptected behaviour from SS and AWF-B technique. Along with STATIC, AWF-D and FISS techniques, they started to exhibit poor performance.

Another unusual behaviour we observe that for AWF-D and AF technique we observe that there is no difference in the parallel execution time. We expected Cython to perform better than Python version as Cython is designed to give C like performance.

Overall we find that RND technique performed the best. The parallel loop execution time for RND techqniue found to be the lowest among all other techniques.

To compare the relative performance between C and Python, and C and Cython, we have a heatmap. This heatmap show the relative difference of the parallel loop execution time of all the programming version. On the Y axis of the heatmap we have all the ranks from 16

to 256, and on the X axis we have all the scheduling techniques. The color value has range from 0 to 14. 0 is the lightest green color which indicates that there is very less relative difference and 14 which is the darkest blue indicates that there is high relative difference between the performance of the programming language.

We observe that the first 10 scheduling techniques and the last PLS technique depicts moderate relative difference of the parallel execution time. For C and Python the range of values are between 4 to 6 where as for C and Cython the range is from 3 to 5. Cython clearly shows that it has performed better than Python. We also observe that AF has very high relative difference between C and Python, and C and Cython. For AF, C has performed well than both Python and Cython. But RND technique has a very low relative difference that range from 0 - 2. We found that bot Python and Cython version has nearly performed well with the C version.

6

Conclusion and Future Work

The results concluded from the experiments depicts that in Python for some dynamic (adaptive and non-adaptive) loop scheduling techniques, it is 55 % slower to C, where as in Cython for the same techniques like Python shows 34.1 % slower to C. But few DLS techniques like AWF-E, VISS and RND performed well in both Python and Cython, and the timing results were some what near to the C results.

RND found to be 8.3 % slower in Python when compared to C and 6.1 % slower in Cython when compared to C. RND performed better because in RND the chunks sizes are calculated using random function from range 1 to N/P. No additional calculation required. And also why RND performed because of the specified range. Value 1 indicates the size of SS technique where as static chunk is N/P. These ranges gives a random value that ensures both load balancing and less scheduling overhead.

AF(Adpative Factoring) technique performed the worst in both Python and Cython when compared to C. The reason why AF performed worst is because it is a very complex scheduling technique. The profiling values required by AF that are μ and σ are calculated during runtime, and also AF does not uses fixed weights, those are also calculated dynamically during runtime.

LB4MPI has potentiality extensions in future like, one can extend with more DLS scheduling techniques that can give optimum performance with less scheduling overhead and high load balance. This library only supports applications with independent loop iteration but some time step applications like Computational Fluid Dynamics (CFD) has dependent loop iterations, therefore one can extend this library for dependent loop iterations. One can also implement a decentralize chunk calculation approach because the current version uses master-worker model and master can be a bottleneck.

Overall, some DLS techniques used in Python has achieved the performance but not all has. In performance wise C is always better than Python but productivity wise Python is better than C. Cython is somewhat acceptable where it shows both productivity and performance but only for few DLS techniques but not for all.

Bibliography

[1] Mohammed Ali et al. "An approach for realistically simulating the performance of scientific applications on high performance computing systems". In: *Future Generation Computer Systems* 111 (2020), pp. 617–633.

[2] Ioana Banicescu, Florina Ciorba, and Srishti Srivastava. "Performance Optimization of Scientic Applications using an Autonomic Computing Approach, in Scalable Computing". In: (Jan. 2013), pp. 437–466.

[3] Ioana Banicescu and Vijay Velusamy. "Load balancing highly irregular computations with the adaptive factoring". In: *Proceedings 16th International Parallel and Distributed Processing Symposium.* IEEE. 2002, 12–pp.

[4] Ricolindo L Carino and Ioana Banicescu. "A tool for a two-level dynamic load balancing strategy in scientific applications". In: *Scalable Computing: Practice and Experience* 8.3 (2007).

[5] Ricolindo L Cariño and Ioana Banicescu. "Dynamic load balancing with adaptive factoring methods in scientific applications". In: *The Journal of Supercomputing* 44.1 (2008), pp. 41–63.

[6] A. Chronopoulos et al. "A Class of Loop Self-Scheduling for Heterogeneous Clusters". In: Oct. 2001, pp. 282–291. ISBN: 0-7695-1390-5. DOI: 10.1109/CLUSTR.2001.959989.

[7] Ahmed Eleliemy and Florina M Ciorba. "A distributed chunk calculation approach for self-scheduling of parallel applications on distributed-memory systems". In: *Journal of Computational Science* 51 (2021), p. 101284.

[8] A. Mohammed F. Ciorba A. Cavelan. "Chapter 8: Resilience and Reproducibility High Performance Computing". In: *University of Basel* (Spring 2021, HPC).

[9] "https://en.wikipedia.org/wiki/Mandelbrot$_set$". In: ().

[10] "https://en.wikipedia.org/wiki/Ray$_t racing_($graphics)". In: ().

[11] "https://p3hpc.org/". In: ().

[12] Susan Flynn Hummel, Edith Schonberg, and Lawrence E Flynn. "Factoring: A method for scheduling parallel loops". In: *Communications of the ACM* 35.8 (1992), pp. 90–101.

[13] Susan Flynn Hummel et al. "Load-sharing in heterogeneous systems via weighted factoring". In: *Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures.* 1996, pp. 318–328.

[14] Jonas Korndörfer et al. “LB4OMP: A Dynamic Load Balancing Library for Multi-threaded Applications”. In: (June 2021).

[15] C.P. Kruskal and A. Weiss. “Allocating Independent Subtasks on Parallel Processors”. In: *IEEE Transactions on Software Engineering* SE-11.10 (1985), pp. 1001–1016. DOI: 10.1109/TSE.1985.231547.

[16] Steven Lucco. “A Dynamic Scheduling Method for Irregular Parallel Programs”. In: *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*. PLDI ’92. San Francisco, California, USA: Association for Computing Machinery, 1992, pp. 200–211. ISBN: 0897914759. DOI: 10.1145/143095.143134. URL: https://doi.org/10.1145/143095.143134.

[17] Teebu Philip and Chita R. Das. “Evaluation of Loop Scheduling Algorithms on Distributed Memory Systems”. In: *Proc. of Intl Conf. on Parallel and Distributed Computing Systems*. 1997.

[18] Constantine D Polychronopoulos and David J Kuck. “Guided self-scheduling: A practical scheduling scheme for parallel supercomputers”. In: *Ieee transactions on computers* 100.12 (1987), pp. 1425–1439.

[19] Wen-Chung Shih, Chao-Tung Yang, and Shian-Shyong Tseng. “A performance-based parallel loop scheduling on grid environments”. In: *The Journal of Supercomputing* 41 (Sept. 2007), pp. 247–267. DOI: 10.1007/s11227-007-0115-7.

[20] Ten H Tzen and Lionel M Ni. “Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers”. In: *IEEE Transactions on parallel and distributed systems* 4.1 (1993), pp. 87–98.