University of Basel

Asynchronous execution of multiple loops in parallel programs using LB4MPI

Bachelor Thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
HPC Group
https://hpc.dmi.unibas.ch/

Advisor: Prof. Dr. Florina M. Ciorba
Supervisor: Dr. Ahmed Hamdy Mohamed Eleliemy

Olivier Heinz Mattmann
olivier.mattmann@unibas.ch
2019-051-523

14.5.2022

Acknowledgments

I would like to express my gratitude for the continuous support I received from my supervisor Dr. Ahmed Hamdy Mohamed Eleliemy and the patience he displayed. His guidance helped me navigate through this interesting and challenging area of computer science. I am also very grateful to my advisor Prof. Florina M. Ciorba for allowing me to complete my bachelors thesis in the HPC research group and providing constructive and motivating feedback during the research group meeting. Lastly, I would also like to thank the entire HPC research group for the helpful ideas in response to challenges I was facing during this thesis.

Abstract

Loops are a frequently occurring control structure in scientific applications. When iterations of these loops are independent of one another, it is possible to distribute the workload among multiple processing elements to execute them in parallel. Various techniques have been developed to schedule loops such that performance is maximized and the workload is well balanced among processing elements. These techniques have been implemented in libraries where each loop is scheduled and executed synchronously. One such library is LB4MPI, which uses the message passing interface for inter-process communication. This thesis extends LB4MPI to support asynchronous execution of multiple loops. The performance evaluation of the extension showed that in applications with high load imbalance, the static and dynamic non-adaptive scheduling techniques perform better when loops are executed asynchronously rather than synchronously. Results also showed that in applications with low load-imbalance synchronous execution of loops outperforms asynchronous execution.

Table of Contents

Acknowledgments **ii**

Abstract **iii**

1 Introduction **1**

2 Background **3**

2.1 Loop Scheduling 3

2.2 Loop Scheduling techniques 4

2.2.1 Static Loop Scheduling (SLS) 4

2.2.2 Dynamic Loop Self-Scheduling (DLS) 4

2.2.2.1 Non-Adaptive Scheduling Techniques 5

2.2.2.2 Adaptive Scheduling Techniques 5

2.3 Message Passing Interface 6

2.4 LB4MPI 7

3 Related Work **10**

4 Methodology **11**

4.1 Implementation 11

4.2 Verification 14

5 Performance Evaluation and Results **16**

5.1 Pi-Solver and STREAM Triad 16

5.2 Mandelbrot 17

5.3 Computing System 17

5.4 Design of Factorial Experiments 18

5.5 Results and Discussion 18

5.5.1 Pi-Solver and STREAM Triad 18

5.5.2 Mandelbrot 19

5.5.2.1 Original and Extended Performance 22

5.5.3 Scheduling Visualization 22

6 Conclusion **25**

6.1 Future Work 25

Bibliography **26**

Appendix A Figures **29**
A.1 Performance Results 29
A.2 Scheduling Visualization of mandelbrot application 29

Appendix B Code **33**
B.1 LB4MPI 33

Declaration on Scientific Integrity **46**

1 Introduction

Loops are a frequently occurring control structure in scientific applications. When the iterations of these loops are independent of one another, we have the option to distribute the iterations among several processing elements (PEs) and execute them in parallel. The workload of each iteration may be heterogeneous, which can lead to unbalanced load distribution among the PEs and deteriorate the performance gained by parallelization. Different loop scheduling techniques were proposed to combat the load imbalance and optimize the performance of such parallel programs. Libraries have been developed that implement a variety of scheduling techniques, one of which is LB4MPI [11]. LB4MPI is a Dynamic Loop Self-Scheduling (DLS) library that utilizes the message passing interface as its channel of communication and is based on a coordinator-worker model. LB4MPI allows the user to schedule individual loops with 14 different techniques to choose from. Once such a loop has been executed in parallel, the PEs are synchronized. After which, possibly another loop can be scheduled. With a high load imbalance, the arrival time at this synchronization point can vary, causing wasted time where faster PEs have to wait idly. This thesis aims to relax this synchronization in-between scheduled loops and evaluate the possible impact on performance by extending the LB4MPI library to allow for asynchronous execution of multiple loops. The chosen approach is to schedule multiple loops at the same time. These loops are then all worked on together by all PEs simultaneously until all loop iterations have been scheduled and executed. This approach has two main advantages to possibly increase performance. (1) It allows PEs that are significantly faster than others to make progress on other loops without having to wait idly. (2) It allows for more balanced resource utilization on the computing nodes. This approach of asynchronously executing loops allows PEs to execute parts of a computationally intensive loop while other PEs execute parts of a memory intensive loop. The risk of this extension is that the additional overhead produced by scheduling multiple loops at once could deteriorate the performance. The coordinator may get overwhelmed by the increased work requests, to which he must respond. The thesis is structured as follows. Section 2 introduces the concept of loop scheduling and the scheduling techniques considered in this thesis. Additionally, the Message Passing Interface standard is shortly described, followed by an introduction to the LB4MPI library. Section 3 section gives reference to past work relevant to this thesis. Section 4 describes the implementation

of the extension and how it was verified. Finally, Section 5 presents the experiments carried out, to compare the performance of synchronous to asynchronous execution of multiple loops.

2

Background

Scheduling is the assignment of workloads or tasks to system resources over a period of time [16]. Those system resources may be processors, network links, or nodes of a computing cluster. Scheduling software that performs the scheduling of tasks is often designed to optimize resource usage and thus the overall performance of the system. Optimally allocating system resources is non-trivial when the tasks to schedule vary in workload and the system resources are heterogeneous. In the following, scheduling is addressed in the context of loop scheduling.

2.1 Loop Scheduling

Loops constitute a significant source of parallelism in scientific applications [17]. When loop iterations are independent of each other, the loop iterations can split into chunks, and these chunks are subsequently distributed among workers to be executed in parallel. This parallelization of loop iterations can lead to a significant increase in performance on high-performance computing systems. But the increase in performance can, in turn, quickly deteriorate due to load imbalance. The load imbalance is caused by problem, algorithmic, or systemic characteristics. Problem or algorithmic characteristics involve irregular workloads per loop iteration caused by conditional statements. Systemic characteristics refer the computational speed of the individual processing elements (PEs), available network bandwidth, or latency. Changes in such systemic characteristics are also called perturbations [12]. High-performance computing (HPC) systems are often built incrementally and thus commonly consist of heterogeneous PEs. Additionally, multiple users may have their applications running simultaneously, increasing the network's latency.

Different loop scheduling techniques can be applied to minimize the impact of the irregularities mentioned above and load imbalances. The scheduling techniques vary in complexity, and have varying operating costs, referred to as scheduling overhead. Careful consideration is vital to determine the best fitting technique for a specific application and high-performance computing system.

2.2 Loop Scheduling techniques

Scheduling techniques can be divided into two categories, **static** and **dynamic**. Each category has its advantages and disadvantages. They differ in what time the scheduling decisions are made. The following two chapters shortly introduce both categories and the associated scheduling techniques considered in this work.

2.2.1 Static Loop Scheduling (SLS)

Static loop scheduling techniques resolve scheduling decisions before the applications are executed. The chunk sizes and assignment to PEs are determined before the execution, and thus they remain fixed. Static scheduling techniques produce the most negligible overhead of all considered scheduling techniques as minimal communication and chunk calculations are required. Parallel applications with low load-imbalance executed on a homogeneous computing system perform very well with static scheduling, because each PE receives a similar workload. This thesis considers static block scheduling [20]. The loop iterations are divided into chunks with size equal to the total number of iterations divided by the number of PEs, resulting in single chunk per PE.

2.2.2 Dynamic Loop Self-Scheduling (DLS)

Dynamic loop self-scheduling, also known as DLS, assigns chunks of iterations to PEs when they are free and request work. The assignment and chunk size calculation happens during the execution. One can differentiate between **non-adaptive** and **adaptive** DLS techniques. The key difference between the two types of techniques is the point when the information is gathered, upon which the techniques base their scheduling decisions. Non-adaptive scheduling techniques base their scheduling decisions on information obtained before execution, while adaptive scheduling techniques base their decisions on information gathered during the execution [15]. This thesis considers the following dynamic scheduling techniques:

non-adaptive	Self-Scheduling (**SS**) [23]
	Modified Fixed-Size-Chunking (**MFSC**) [10]
	Guided Self-Scheduling (**GSS**) [22]
	Trapezoid Self-Scheduling (**TSS**) [24]
	Factoring 2 (**FAC2**) [18]
	Weighted Factoring (**WF**) [19]
adaptive	Adaptive Weighted Factoring (**AWF**) [9]
	Adaptive Weighted Factoring - Batch (**AWF-B**) [12]
	Adaptive Weighted Factoring - Chunk (**AWF-C**) [12]
	Adaptive Weighted Factoring - Batch with scheduling overhead (**AWF-D**) [12]
	Adaptive Weighted Factoring - Chunk with scheduling overhead (**AWF-E**) [12]
	Adaptive Factoring (**AF**) [8]

Table 2.1: Dynamic scheduling techniques considered in this thesis

2.2.2.1 Non-Adaptive Scheduling Techniques

Self-scheduling (**SS**) is a technique where a PE is assigned a chunk with size 1 during the execution when it becomes idle, and requests work. This technique consistently achieves a good load balance but not necessarily good overall performance [21]. The drop in performance is caused by the significant overhead produced, growing in the number of iterations of the loop. Each iterate has to be requested individually, resulting in frequent communication between PEs. SS produces the most overhead of the considered scheduling techniques.

MFSC is a modified version of the fixed-size chunking technique (FSC). Similar to SS, MFSC and FSC assign chunks of fixed size. To calculate the chunk size in FSC the scheduling overhead h and the standard deviation σ of the loop iterations have to be known. For MFSC, this requirement of apriori knowledge is no longer necessary. As a result of the bigger chunk sizes, the scheduling overhead is decreased compared to SS.

Guided self-scheduling (**GSS**) assigns chunks with variable sizes. The chunk sizes decrease as more iterations of a loop are scheduled and correspond to the remaining unscheduled loop iterations divided by the number of processing elements. The larger chunk sizes at the beginning allow for reduced overhead, while the decreasing chunk sizes allow for good load balancing.

Trapezoid self-scheduling (**TSS**) attempts to achieve good load balance while keeping the overhead small. TSS assigns chunks in decreasing size like GSS, but unlike GSS, the chunk size decreases linearly. Due to the linearity of the chunk size calculation, only a small overhead is induced.

FAC2 is a practical implementation of Factoring (FAC). FAC takes a probabilistic modeling approach to determine batch sizes. A batch is a part of the total loop iterations, which subsequently is split into chunks to assign to each PE. The calculated batch sizes of this technique maximize the probability of load-balanced execution. The model utilizes the mean of iteration execution time μ and their standard deviation σ, which have to be known before execution. The practical implementation (FAC2) alleviates the need for μ and σ and instead, half of the remaining iterations are assigned to a batch. Chunks sizes are calculated by dividing the batch size by the number of PEs.

Weighted Factoring (**WF**) is similar to FAC in that it also divides loop iterations into batches. This thesis considers the practical implementation of WF, where the batch sizes are equal to those of FAC2. Unlike Factoring, WF determines the chunk size for each PE proportional to the weight associated with each PE. The weights must be determined before the execution and stay unchanged during the program's execution. The use of weights results in unequal chunk sizes in a batch.

2.2.2.2 Adaptive Scheduling Techniques

Adaptive techniques make use of statistics gathered during the execution of the program. Those statistics include the measured time to execute the assigned chunks and sometimes also the time required for the chunk assignment. This allows the scheduling techniques to take into account variances in the computing system characteristics, which possibly change during the execution.

Adaptive Weighted Factoring evolved from the Weighted Factoring technique. **AWF** allows for the weight associated with each PE to change during the execution in contrast to WF. It is designed for time-stepping applications and adapts the weights after each time step. The adaptation considers the cumulative performance measured by the loop execution time during previous time steps.

AWF-B is a variation of AWF where the requirement of a time-stepping application is removed. Instead, AWF-B adapts the weights after each batch of the scheduled loop. A batch is a portion of all loop iterations, which gets assigned to PEs in smaller chunks. Like AWF, the adaptation takes into consideration cumulative performance.

AWF-C is a further variation of AWF where the weights are adapted after each chunk instead of after each batch Thus AWF-C is adapting its weights even more frequently than AWF-B and should balance the load better at the cost of increased scheduling overhead.

AWF-D is similar to AWF-B in terms that it does not need to be a time-stepping application, and the weights are adapted after each batch of iterations. Unlike AWF-B, AWF-D considers both the cumulative loop execution times and the time required for the chunk assignment and bookkeeping.

AWF-E is similar to AWF-C, as for both techniques the weights are recomputed after each chunk execution. Analogously to AWF-D, it considers the cumulative loop execution times in addition to the time required for the chunk assignment and bookkeeping.

Adaptive Factoring (**AF**) is a similar approach to the FAC technique. AF dynamically estimates the mean and standard deviation of the iterate execution times during runtime. FAC requires these statistics to be known before the execution and assumes them to be equal on all PEs, whereas AF adapts the weights for each PE during execution.

2.3 Message Passing Interface

The Message Passing Interface (MPI) is a library specification for message-passing, proposed as a standard by a broadly based committee of vendors, implementors, and users [2]. MPI allows processes that are possibly distributed among several computing nodes to exchange messages. The involved processes are referred to as ranks and can be part of groups, called communicators. The functions relevant in this thesis are shown in Table 2.2.

MPI_Send	Send a message to a specified rank of a communicator
MPI_Recv	Receive a message of communicator
MPI_Probe	Check if a message is available to receive
MPI_Gather	Gather information from all ranks in a communicator
MPI_Barrier	Synchronize all ranks in a communicator

Table 2.2: Relevant MPI functions

Most of these functionalities can be synchronous, i.e., blocking, or asynchronous, i.e., non-blocking. Sent messages can have a tag specified for the receiver to distinguish among different message types. Various proprietary or free implementations of this standard exist.

2.4 LB4MPI

LB4MPI is a DLS library available both in the C and Fortran programming language, which utilizes the Message Passing Interface to distribute the workload among PEs. The library offers users with limited programming experience a tool to easily utilize the scheduling techniques introduced in Section 2.2 to parallelize their applications with only a small number of changes to their code. LB4MPI is an extension of a load-balancing tool initially developed as part of a paper by Carino and Banicescu [11] published in 2007. LB4MPI has been used on multiple occasions to conduct research in dynamic loop scheduling. Eleliemy and Ciorba [15] proposed a distributed chunk calculation approach which was novel to the library because it previously only supported a centralized chunk calculation approach. This thesis extends the C version of LB4MPI with the centralized chunk calculation approach, which is available open-source [6], and addresses the impact of asynchronous execution of multiple loops on performance.

As mentioned, LB4MPI utilizes a centralized chunk calculation approach. One MPI rank is assigned to be foreman and is responsible for calculating chunk sizes and distributing them among the worker MPI Ranks. In the approach used, the foreman acts as a worker himself and executes chunks of iterations, periodically checking for new requests. Three types of messages are exchanged between the MPI ranks. Whenever a worker is ready for a new chunk, it sends a request message to the foreman, possibly containing statistics necessary to use adaptive scheduling techniques. In response to request messages, the foreman sends work messages back to the workers that contain information about the assigned chunk. When all loop iterations are scheduled, the foreman responds to request messages with an end message, signaling to the worker that no more work is available. LB4MPI uses an **infoDLS** struct for every rank to store information about the MPI environment, and the required values to utilize the implemented scheduling techniques. Each function mentioned below takes a pointer to such a struct to read or modify the members. An Illustrative example usage of the library can be seen in Listing 2.1.
The library offers the following API functions and are briefly described:

- **void DLS_Parameters_Setup(MPI_Comm icomm, infoDLS *info, ...);**
- **void DLS_GroupSetup(MPI_Comm comm, int, infoDLS *iInfo, infoDLS *jInfo);**
- **void DLS_StartLoop(infoDLS *info, int firstIter, int lastIter, int method)**
- **int DLS_Terminated(infoDLS *info);**
- **void DLS_StartChunk(infoDLS *info, int *start, int *chunk_size)**
- **void DLS_EndChunk(infoDLS *info);**
- **void DLS_EndLoop(infoDLS *info, int *nIter, double *workTime)**
- **void DLS_Finalize(infoDLS *info);**

DLS_Parameters_Setup and **DLS_Group_Setup** initialize infoDLS with the parameters supplied by the user. DLS_Group_Setup is for a 2-Layer scheduling approach that

this thesis will not consider. The parameters of DLS_Parameters_Setup allow a user to tune the library to a certain computing environment and behavior of the ranks, such as how early a rank should request the next chunk of work or how many MPI ranks are involved in the parallel execution.

DLS_StartLoop is called before the loop is scheduled. The parameters are used to initialize the loop-specific values stored in infoDLS, such as the number of total iterations and the scheduling method used. The foreman additionally sends the first chunk assignment to all workers before returning.

Calling **DLS_Terminated** returns 1 when an end message was received and 0 when there is still work. A while loop is constructed with this function call as a stopping condition. This essentially creates a while True loop until all iterations are scheduled. Inside the while loop, **DLS_StartChunk** is called. In this function, workers receive work and end messages, and request messages are received and responded to by the foreman. The work messages update values in the infoDLS struct, which are subsequently used to update the values of the supplied parameters indicating the start index of the chunk and the chunk size to be executed next. After the execution of the chunk, the worker calls **DLS_EndChunk** where request messages are sent to the foreman. This behavior repeats in the while loop until an end message is received in DLS_StartChunk.

After the while loop, each rank calls **DLS_EndLoop** where the number of iterations executed and the total work time can be obtained for each worker. This call is also the point of synchronization where each MPI rank waits for the remaining ranks to leave the preceding while loop. After this synchronization, more loops could be scheduled before **DLS_Finalize** has to be called, which frees all the heap-allocated memory used by LB4MPI. This thesis explores the effect of relaxing the synchronization in DLS_EndLoop and allowing the execution of chunks of multiple loops asynchronously.

```
1  infoDLS iInfo;
2  MPI_Init(&argc, &argv) // initialize MPI environment
3  DLS_Parameters_Setup(MPI_COMM_WORLD, &iInfo, numProcs, requestWhen, breakAfter,
4                       minChunk, h_overhead, sigma, nKNL, xeon_speed, KNL_speed);
5  int start, chunkSize;
6  int nIter;
7  double workTime;
8  DLS_StartLoop(&iInfo, firstIter_1, lastIter_1, method_1);
9  while (!DLS_Terminated(&iInfo)) {
10     DLS_StartChunk(&iInfo, &start, &chunkSize); // get chunk start and size
11     calculate_chunk_loop_1(start, chunkSize);
12     DLS_EndChunk(&iInfo, &nIter, &workTime); // possibly request next chunk
13 }
14 DLS_EndLoop(&iInfo, &nIter, &workTime); // workers synchronize here
15 //..
16 // possibly more loops
17 // ...
18 DLS_StartLoop(&iInfo, firstIter_n, lastIter_n, method_n);
19 while (!DLS_Terminated(&iInfo)) {
20     DLS_StartChunk(&iInfo, &start, &chunkSize); // get chunk start and size
21     calculate_chunk_loop_n(start, chunkSize);
22     DLS_EndChunk(&iInfo, &nIter, &workTime); // possibly request next chunk
```

```
23 }
24 DLS_EndLoop(&iInfo, &nIter, &workTime); // workers synchronize here
25 DLS_Finalize(&iInfo);
26 MPI_Finalize();
```

Listing 2.1: Illustrative example how LB4MPI can be used to schedule multiple loops in a synchronized fashion. Initialization of used variables is omitted due to space reasons.

3
Related Work

Most DLS implementations make use of the master-worker execution model. In each scheduling step, the master calculates the chunk size and assigns it to the PE where the work request came from. One such implementation is the distributed self-scheduling scheme (DSS) [13] which is designed for distributed memory systems. In DSS, the master is the central entity, which calculates chunk sizes and assigns them to workers. The speed of the worker is taken into consideration when calculating the chunk size. A hierarchical distributed self-scheduling scheme (HDSS) [14] is proposed which differs from DSS in that a global master assigns work to local masters, which in turn assign work to the workers. The hierarchical scheme is similar to the two-level dynamic load balancing strategy present in LB4MPI and aims to improve the scalability of the self-scheduling schemes. Unlike LB4MPI, in DSS and HDSS, the master is not executing chunks himself and is only responsible for the chunk calculation. The communication between the MPI ranks is two-sided, meaning both master and workers send and receive messages.

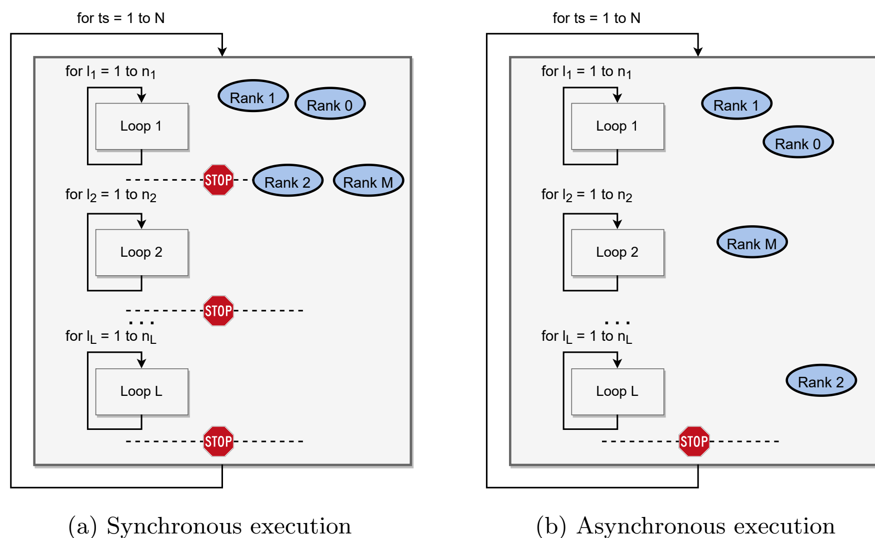
DLS implementations employing one-sided communication exist as well, such as the dynamic load balancing library (DLBL) [7]. Like LB4MPI, it uses a master-worker execution model. The library is a collection of functions referred to as handlers. When the master receives work requests, it first calculates the chunk size and then calls a specific handler such that the worker can obtain the assigned work without further communication.

This thesis is motivated by the fact that in the mentioned DLS libraries in this section, loops are scheduled in a synchronized fashion. This synchronization can accumulate time spent waiting for slower workers to finish their work due to load imbalance. The approach of asynchronously scheduling and executing multiple loops is proposed to minimize wasted time. The synchronous and asynchronous execution of multiple loops is then compared by their performance.

4 Methodology

4.1 Implementation

The described implementation of LB4MPI in Section 2.4 is limited to holding and maintaining information about a single loop at a time. This limitation forces us to synchronize the workers after each loop we want to schedule. Suppose the coordinator, who schedules the loops, signaled to everyone that the current loop has been completely scheduled while one rank still has not finished executing his last received chunk. Without synchronization, the coordinator could already have sent out the first chunk for the next loop. The worker still executing the chunk of the previous loop would receive both the end message and work messages and subsequently compute the received chunk for the wrong loop before proceeding to the next loop.



(a) Synchronous execution

(b) Asynchronous execution

Figure 4.1: Example of a time-stepping application with multiple loops. 4.1a shows the synchronous execution where ranks are synchronized after each loop in a time-step. 4.1b shows what the extension tries to achieve. Workers can execute chunks of multiple loops asynchronously and are only synchronized at the end of a time-step.

LB4MPI is extended such that multiple loops can be scheduled at the same time, and

the chunk executions of the loops interleave, i.e., workers can execute chunks for all loops until all loops are completely scheduled. The extension maintains backward compatibility such that applications written using the original version of LB4MPI still function correctly. The user can choose to execute multiple loops synchronously or asynchronously.

The extension modifies the infoDLS as seen in Listing 4.1 and 4.2. Members of the struct which hold information specific to a loop are changed to pointers to store information about multiple loops at once. The required memory for these members is heap-allocated once it is clear how many loops are to be scheduled.

```
typedef struct
{
  MPI_Comm comm, crew;
  int commSize, crewSize;
  int foreman, myRank, firstRank, lastRank;
  int method;
  int firstIter, lastIter, N,
      itersScheduled;
  int batchSize, batchRem,
      minChunkSize, maxChunkSize;
  int minChunk, breakAfter, requestWhen,
      chunkFSC, chunkMFSC;
  int chunkStart, probeFreq,
      sendRequest, subChunkSize;
  int numChunks, numENDed, finishedOne;
  int myExecs, myIters;
  int rStart, rSize, wStart, wSize,
      nextStart, nextSize;
  int gotWork, req4WRKsent,
      nextWRKrcvd;
  double kopt0, workTime;
  double t0, t1, sumt1, sumt2,
      mySumTimes, mySumSizes;
  double *stats;
  double h_overhead;
  double sigma;
  double *weights;
  int TSSchunk;
  int TSSdelta;
  int timeStep;
  } infoDLS;
```

Listing 4.1: Original infoDLS struct of the library before the extension. Information about a single loop can be held.

```
typedef struct
{
  MPI_Comm comm, crew;
  int commSize, crewSize;
  int foreman, myRank, firstRank, lastRank;
  int *method;
  int *firstIter, *lastIter, *N,
      *itersScheduled;
  int *batchSize, *batchRem,
      *minChunkSize, *maxChunkSize;
  int minChunk, breakAfter, requestWhen,
      *chunkFSC, *chunkMFSC;
  int *chunkStart, *probeFreq,
      *sendRequest, *subChunkSize;
  int numChunks, *numENDed, *finishedOne;
  int *myExecs, *myIters;
  int *rStart, *rSize, *wStart, *wSize,
      *nextStart, *nextSize;
  int *gotWork, *req4WRKsent,
      *nextWRKrcvd;
  double *kopt0, *workTime;
  double *t0, *t1, *sumt1, *sumt2,
      *mySumTimes, *mySumSizes;
  double *stats;
  double h_overhead;
  double sigma;
  double *weights;
  int *TSSchunk;
  int *TSSdelta;
  int *timeStep;
  int numLoops;
  int curLoop;
  double *tExclude;
  } infoDLS;
```

Listing 4.2: Extended infoDLS struct where loop specific members are changed to pointers and 3 new members are added. Information about multiple loops can be held.

In addition to the modified members, three new members are introduced to the struct. The integer *numLoops* denotes the number of loops to execute asynchronously. The integer *curLoop* is used to keep track of which loop a worker is currently computing a chunk for and a pointer to a double array *tExclude* helps to keep track of how much time was spent in each loop. This last addition is only required for AWF-D and AWF-E because they require the bookkeeping time for each loop to adapt the weights of each worker correctly. An example program is provided in Listing 4.3 to illustrate the usage of the extended library.
The extension introduces five new API functions:

- **void DLS_NumLoops(infoDLS *info, int n);**

- **void DLS_StartMLoops(infoDLS *info, int *firstIters, int *lastIters, int *imeths);**
- **int DLS_MTerminated(infoDLS *info);**
- **void DLS_TargetLoop(infoDLS *info, int l);**
- **void DLS_EndMLoops(infoDLS *info, int *niters, double *worktime);**

DLS_NumLoops updates the new member numLoops in the infoDLS struct with the parameter supplied. This function needs to be called before DLS_Parameters_Setup such that the appropriate amount of memory is allocated. If the function is not called before DLS_Parameters_Setup, it is assumed one loop is scheduled at a time.

DLS_StartMLoops is the new counterpart to DLS_StartLoop. The function body of DLS_StartMLoops is an extended version of DLS_StartLoop's function body. The modification allows us to initialize one or more loop-specific members of infoDLS in one function call, such as the starting index and last index for each loop. An arbitrary combination of the offered scheduling techniques can be chosen when scheduling multiple loops. Since this extended function can also handle the initialization for only one loop, DLS_StartLoop is modified to call DLS_StartMLoops internally.

DLS_MTerminated analogously to DLS_StartMLoops is the new counterpart to DLS_Terminated. The extended version generalizes the DLS_Terminated to work with one or multiple loops. Using DLS_MTerminated as a stopping condition, one iteration of the while loop computes one chunk of each loop if work is available. DLS_MTerminated only returns 1 if there is no more work available for all the loops scheduled asynchronously. If only one loop is scheduled at a time, then the behavior of DLS_MTerminated is the same as DLS_Terminated. As such, DLS_Terminated calls DLS_MTerminated internally.

DLS_TargetLoop is used to select the loop for which the next chunk start and chunk size is obtained. This is done by setting the new member curLoop in the infoDLS struct to the appropriate value. The value is subsequently used by **DLS_StartChunk** to provide the chunk information for the correct loop. A level of complexity is added to DLS_StartChunk because work, request, and end messages for other than the currently set target loop can be received in this function. Therefore, each message is extended to specify which loop the message refers to. This additional information allows updating values in the infoDLS struct and responding to requests in a targeted manner for each loop. When a work message for a different loop than the target loop is received, the worker skips the target loop to avoid further time probing for messages. The worker continues with a chunk size of 0, therefore proceeding to the next loop where work may be available now. This skipping of loops will be of relevance in the discussion in Section 5.5.1.

DLS_EndChunk is modified to send request messages if needed for the currently targeted loop, after which another loop can be targeted to obtain and execute a chunk.

When an end message is received for each asynchronously executed loop, the while loop breaks—letting the worker proceeds to **DLS_EndMLoops**. This function is the point of synchronization when multiple loops are executed asynchronously. Suppose an application contains three loops that should be executed in parallel. Usage of the synchronized version

of LB4MPI would result in three synchronization points where workers possibly wait idly. Using the extension to execute the three loops asynchronously would only result in one synchronization point where workers possibly wait idly. The asynchronous approach allows progress on all loops simultaneously and should decrease the overall time spent waiting idly.

```
infoDLS iInfo;
MPI_Init(&argc, &argv) // initialize MPI environment
DLS_NumLoops(&iInfo, n);
DLS_Parameters_Setup(MPI_COMM_WORLD, &iInfo, numProcs, requestWhen, breakAfter,
                     minChunk, h_overhead, sigma, nKNL, xeon_speed, KNL_speed);
int start, chunkSize;
int nIters[n];
double workTimes[n];
int firstIters = {0,...,0};
int lastIters = {lastIter_loop_1,...,lastIter_loop_n};
int methods = {method_loop_1,...,method_loop_n};
DLS_StartLoop(&iInfo, firstIters, lastIters, methods);
while (!DLS_MTerminated(&iInfo)) {
    DLS_TargetLoop(&iInfo, 0);
    DLS_StartChunk(&iInfo, &start, &chunkSize); // get chunk start and size
    calculate_chunk_loop_1(start, chunkSize);
    DLS_EndChunk(&iInfo, &nIter, &workTime); // possibly request next chunk
    //..
    // possibly more loops
    // ...
    DLS_TargetLoop(&iInfo, n-1);
    DLS_StartChunk(&iInfo, &start, &chunkSize); // get chunk start and size
    calculate_chunk_loop_n(start, chunkSize);
    DLS_EndChunk(&iInfo, &nIter, &workTime); // possibly request next chunk
}
DLS_EndMLoops(&iInfo, nIters, workTimes); // workers synchronize here
DLS_Finalize(&iInfo);
MPI_Finalize();
```

Listing 4.3: Illustrative example how LB4MPI can be used to schedule multiple loops in an asynchronous fashion. Initialization of used variables is omitted due to space reasons.

4.2 Verification

The extended LB4MPI library is tested with two forms of verification. The first verification is used to confirm and verify that all loop iterations are scheduled and executed correctly with and without synchronization. A small parallel application is used, which computes the sum and product of a sequence of numbers in two separate loops. The results of all workers are combined and checked for correctness. All considered scheduling techniques pass this verification with synchronization among loops and without.

The second form of verification compares the performance of the original, unmodified LB4MPI library with the modified version of LB4MPI, where the loops are still scheduled and executed in a synchronized fashion. This second verification attempts to raise the confidence that no bugs were introduced in the chunk calculations of the scheduling techniques in the process of the implementation. Comparing the unmodified LB4MPI library to the

extended version also allows for estimating the additional overhead the modification to the code produced. This verification is performed with the Mandelbrot application introduced shortly in Section 5.2. The verification, however, does not guarantee that no bugs exist when multiple loops are scheduled and executed in an asynchronous fashion. The results of the second verification are presented in Section 5.5.2, along with the performance results of the Mandelbrot application used for performance evaluation.

The source code of the modified/new functions is provided Appendix B.1

5

Performance Evaluation and Results

The performance of both the synchronous and asynchronous execution of multiple loops using LB4MPI is evaluated by a design of factorial experiments presented later. Additionally, visualizations are presented illustrating the asynchronous execution of multiple loops in comparison to synchronous execution. Two time-stepping applications are evaluated with different properties each.

5.1 Pi-Solver and STREAM Triad

The first application contains two HPC kernels. A kernel is a function with specific properties. Pi-Solver is a computationally intensive (CPU bound) kernel seen in Listing 5.1. The name suggests that the number pi is estimated with this kernel. This was not the case in the version used for the evaluation. However, the computationally intensive property remains and is sufficient for the evaluation. The second kernel is STREAM Triad. STREAM is a benchmark to measure sustainable memory bandwidth [1]. STREAM Triad is one of many kernels used in STREAM benchmark and can be seen in Listing 5.2. Three arrays of equal size are initialized and used for small computations in this kernel. The array size has to be chosen big enough such that the arrays don't fit in the cache, and thus main memory has to be accessed often during execution. This kernel is chosen to complement the CPU bound kernel with a memory intensive (memory bound) kernel. The combination of the two kernels has the purpose of investigating whether asynchronous execution of both loops leads to a performance increase since it allows workers to perform memory intensive tasks while other workers are executing computationally intensive tasks. Synchronized execution of both loops forces all workers to either execute CPU bound or memory bound tasks at a time. The Tasks in this application show a low load imbalance.

```
for (int i = 0; i < lastIter; i++ {
    x = (i+0.5)*stepLength;
    sum += 4.0/(1.0+x*x);
}
```

Listing 5.1: Pi-Solver kernel, stepLength is decreased with timesteps

```
for (int i = 0; i < arraySize;i++) {
    A[i] = B[i] + 2 * C[i];

}
```

Listing 5.2: STREAM Triad kernel, each array is accessed once per iteration

5.2 Mandelbrot

The second application is a time-stepping application to compute the Mandelbrot set. It is a computationally intensive application with a high load imbalance. The *original* Mandelbrot set is the set of complex numbers c in the complex plane for which z stays bounded in equation 5.1 [25]. This application uses equation 5.2. Different values for c result in a varying number of iterations needed to check for divergence. A visualization of the Mandelbrot set produced by the application can be seen in Figure 5.1.

$$\begin{aligned} z_0 &= 0 \\ z_{n+1} &= z_n^2 + c \end{aligned} \tag{5.1}$$

$$\begin{aligned} z_0 &= 0 \\ z_{n+1} &= z_n^4 + c \end{aligned} \tag{5.2}$$

Figure 5.1: Mandelbrot set generated with the application used to evaluate performance. The set is computed 3 times for a total of 960'400 complex numbers. 10'000 max Iterations to check for divergence are used.

The Mandelbrot set is computed three times in separate loops in each time step. A different property characterizes each loop. The first loop has a constant load imbalance over all time steps, while the load imbalance increases and decreases for the second and third loop, respectively. The results of the computations are not collected as the execution time is the metric of interest and not the result itself. This application aims to investigate the impact of the relaxed synchronization among loops in applications with high load imbalance on performance.

5.3 Computing System

The computing system on which the performance is evaluated is a small high-performance computing cluster referred to as miniHPC and is located at the University of Basel. It serves as a platform for educational purposes while also offering a fully controllable research environment for scientific experiments furthering HPC research. The cluster is made up of four

types of nodes. The nodes are interconnected by Ethernet with 10 Gbit/s speed and an Intel Omni-Path network with 100 Gbit/s speed. The fast Omni-Path network is structured in a two-level fat-tree topology, reserved for high-speed communication between nodes. The experiments are performed on 16 nodes containing two Intel Xeon E5-2640 v4 CPUs. For further information about miniHPC, the reader is referred to [3].

5.4 Design of Factorial Experiments

The design of factorial experiments is presented in Table 5.1. The Pi-Solver + STREAM (Triad) application used an array size of 20'000'000 and 100'000'000 iterations per Pi-Solver loop. The number of time steps is set to 10'000. For the Mandelbrot application, the number of time steps is set to 200. Each of the three Loops computes the Mandelbrot set for 262'144 complex numbers, and the maximum number of iterations to check for divergence is set to 10'000. Each scheduling technique is used with and without synchronization among the loops. SS and AF have been excluded from the Pi-Solver and STREAM Triad experiments due to their excessively high execution times. Each experiment is repeated 20 times. All scheduling techniques used the default chunk parameter of 1, representing the smallest chunk size that can be scheduled. The applications are compiled with Intel compiler at version 2021.4.0 and executed on 16 Intel Xeon E5-2640 v4 nodes with 16 MPI ranks per node. This configuration results in a total of 256 ranks per experiment. For each experiment, the total parallel execution time of all time-steps is measured along the parallel loop execution time of each loop to evaluate the performance.

Factors		Values	Properties
Applications		Pi-solver + STREAM (Triad)	$T = 10000$ |Total loops = 2 |Modified loops = 2 Pi-Solver: $N = 100000000$ STREAM Triad: $N = 20000000$
		Mandelbrot	$T = 200$ |max Iterations/pixel = 10000 Total loops = 3 |Modified loops = 3 L1: $N = 262144$ (constant load-imbalance) L2: $N = 262144$ (increasing load-imbalance) L3: $N = 262144$ (decreasing load-imbalance)
LB4MPI **without** synchronization among loops LB4MPI **with** synchronization among loops	Scheduling techniques	STATIC	Straightforward parallelization
		SS, MFSC, GSS, TSS, FAC2, WF	**Dynamic** and **non-adaptive** self-scheduling techniques (SS excluded for Pi-Solver+STREAM Triad)
		AWF, AWF-B, AWF-C, AWF-D, AWF-E, AF	**Dynamic** and **adaptive** self-scheduling techniques (AF excluded for Pi-Solver+STREAM Triad)
Chunk parameters		default 1	represents the smallest chunk size a rank can obtain with a given self-scheduling technique
Computing system		miniHPC	16 Dual socket Intel Xeon E5-2640v4 nodes 64 GB DDRAM per node Nonblocking fat-tree topology Fabric: Intel OmniPath - 100 Gbps 16 MPI ranks per node, Total of 256 MPI ranks
Metrics		Performance per loop	Parallel loop execution time T_{par}^{loop}

Table 5.1: Design of factorial experiments resulting in a total of 960 experiments

5.5 Results and Discussion

5.5.1 Pi-Solver and STREAM Triad

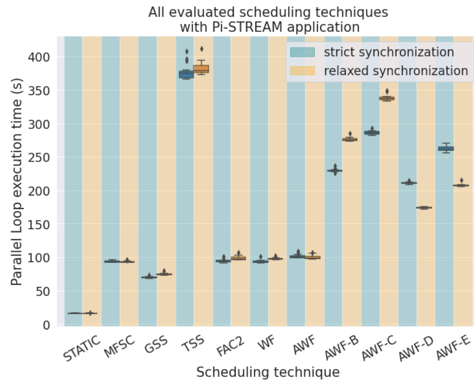
Figures 5.2a to 5.2d show the performance results of the Pi-Solver + STREAM Triad application. In Fig. 5.2a all evaluated scheduling techniques are contained in one boxplot

as an overview. Overall the static and non-adaptive scheduling techniques performed better with synchronous and asynchronous execution than the adaptive techniques. The low load-imbalance of the application can explain this result. When the difference in workload per iteration is negligible, static scheduling performs the best because of the minimal scheduling overhead. As the produced overhead of the scheduling technique increases, so does the parallel loop execution time. An outlier that needs addressing is the TSS scheduling technique. It unexpectedly showed the highest parallel loop execution time of all techniques while also having the highest variance among the scheduling techniques. Figure 5.2d shows that most of the parallel loop execution time for TSS was spent in Pi-Solver. Performing the computations in Pi-Solver takes only a small amount of time compared to STREAM. Thus, a significant amount of time is spent probing for new messages. This indicates that the foreman is overwhelmed with requests and negatively impacts the performance due to delayed work messages. A bug in the code of the library related to TSS can almost certainly be ruled out, as the comparison of the unmodified and the extended library showed no such indication as later seen in Section 5.5.2.1.

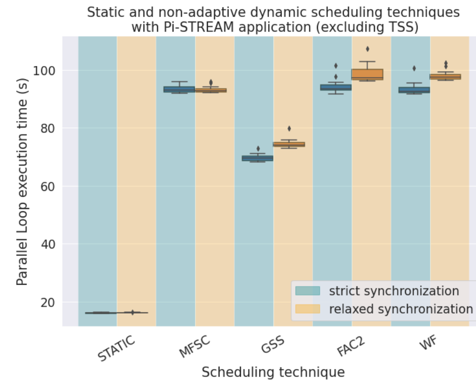
For this application, the asynchronous execution results in worse performance compared to the synchronous execution in almost all cases. Time spent in Pi-Solver increases when comparing synchronous to asynchronous executions. Requests referring to either loop can arrive at any time. Due to the skipping mechanism mentioned in Section 4.1 and the long probing times for Pi-Solver, almost all work messages for the STREAM kernel are received while waiting for Pi-Solver work messages. Thus the average parallel loop execution time of STREAM in the asynchronous case decreases compared to the synchronous case. Interestingly AWF-D and AWF-E performed better with asynchronous execution than with synchronous execution. Both scheduling techniques take chunk assignment time into account when adapting the weights. This could be the reason for the performance increase as chunk assignment time constitutes a large part of this application. Whether the combination of CPU bound and memory bound kernels executed asynchronously leads to better or worse performance is not conclusive. This could partly be attributed to the high difference in work time for each kernel with the chosen experiment parameters. More iterations per time-step of Pi-Solver to increase work time may offer more conclusive results.

5.5.2 Mandelbrot

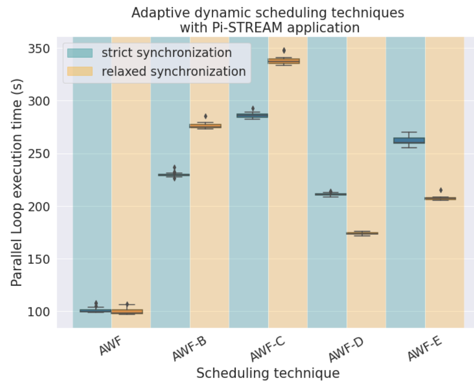
In Figures 5.3a to 5.3d the performance results of the Mandelbrot application are shown. One can see that, similarly to the Pi-Solver and STREAM application, static and non-adaptive dynamic scheduling techniques outperform the dynamic adaptive scheduling techniques. The applications are executed on homogeneous nodes with the same hardware. Additionally, no perturbations in the systems are present as the experiments are run exclusively with no other applications running on the same node. As a result, the adaptive scheduling techniques show worse performance, as they work exceptionally well when perturbations are present in the system. When no perturbations are present, the additional scheduling overhead deteriorates the performance. Fig. 5.3b shows that non-adaptive dynamic scheduling techniques perform better than static scheduling. The load imbalance



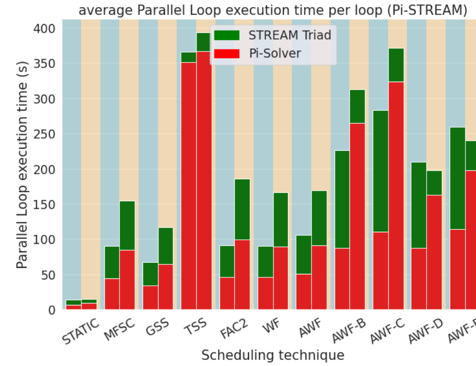
(a) Boxplot of every evaluated scheduling techniques.



(b) Boxplot of same results as Figure 5.2a, only showing static and non-adaptive dynamic scheduling techniques. TSS has been excluded for better scaling in the plot. A plot with TSS can be found in Appendix A.1a



(c) Boxplot of same results as Figure 5.2a, only showing adaptive dynamic scheduling techniques.

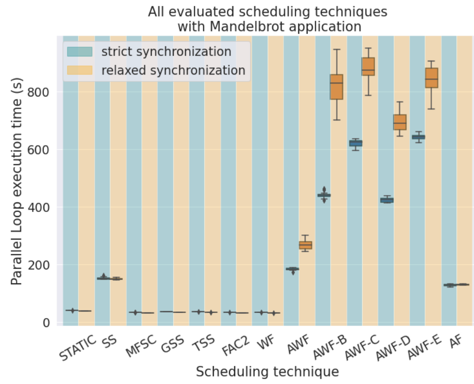


(d) Barplot showing the parallel loop execution time of each individual loop where the red bars represent the time for Pi-Solver and the green bars represent the time for STREAM Triad.

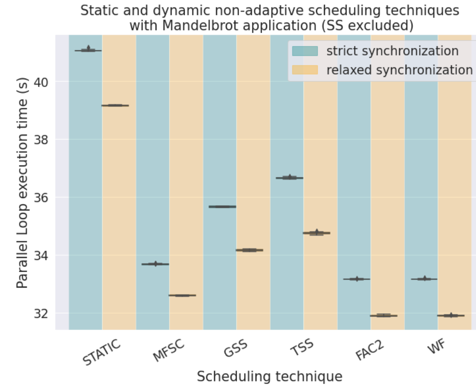
Figure 5.2: Pi-Solver + STREAM Triad performance results. Light blue background corresponds to results with synchronization among loops and light orange background corresponds to results obtained from the experiments with relaxed synchronization among loops (asynchronous). The x-axis shows the scheduling techniques and the y-axis shows the parallel loop execution time in seconds.

of this application is quite high, as for each iteration, the workload can vary significantly. Consequently, the load is distributed better when using dynamic non-adaptive scheduling techniques. The results show lower parallel loop execution times with static and dynamic non-adaptive scheduling techniques when loops are executed asynchronously. Due to the high load imbalance, it is likely that some workers have faster execution times than others. When loops are executed synchronously, the faster workers spend time waiting idly to synchronize with the slower workers. Asynchronous execution allows the workers to make progress on all loops, eliminating the idle time between loop executions. The only possible idle time with asynchronous execution is produced in DLS_EndMLoops, after each time step. The overall idle time is reduced by executing loops asynchronously. Fig. 5.3d shows

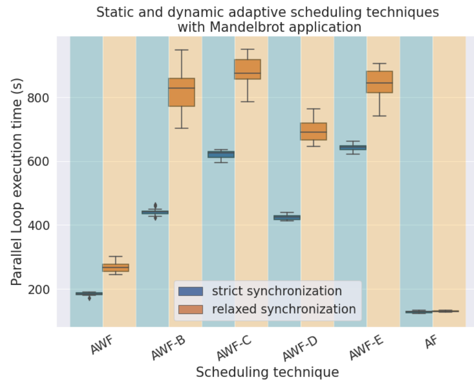
that the summed average parallel loop execution time per loop with asynchronous execution is actually higher than with synchronous execution. This can be explained by the fact that more time is spent probing for messages with asynchronous execution as the foreman has to handle more requests at once. The additional probing time is compensated by shorter idle time of workers, resulting in a better overall performance.



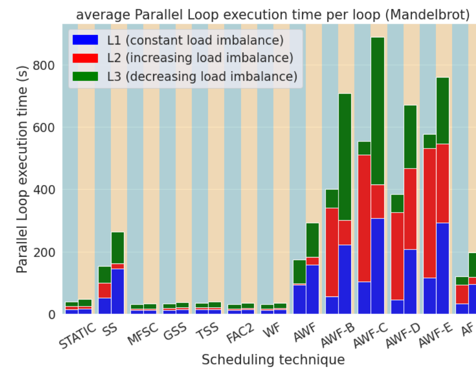
(a) Boxplot of every evaluated scheduling techniques.



(b) Boxplot of same results as Figure 5.3a, only showing static and non-adaptive dynamic scheduling techniques. SS has been excluded for better scaling but a plot with SS included can be found in Appendix A.1b



(c) Boxplot of same results as Figure 5.3a, only showing adaptive dynamic scheduling techniques.



(d) Barplot showing the parallel loop execution time of each individual loop in the Mandelbrot application where the blue bars represent the average parallel loop execution time of loop 1 with constant load imbalance. Loop 2 has increasing load imbalance over the times-steps and is represented with the color red. Loop 3 has decreasing load imbalance over the time-steps and is represented with the color green.

Figure 5.3: Mandelbrot performance results. Light blue background corresponds to results with synchronization among loops and light orange corresponds to results obtained from the experiments with relaxed synchronization among loops. The x-axis shows the scheduling techniques, and the y-axis shows the parallel loop execution time in seconds.

Dynamic adaptive scheduling techniques show a worse parallel loop execution time

when executed asynchronously, as seen in Figure 5.3c. Once again, asynchronous execution leads to more time spent probing for messages. This time is magnified as the chunk calculation and assignment time of the techniques increases. In this case, the reduced idle time cannot compensate for the probing time, and as a result, asynchronous execution shows worse performance.

5.5.2.1 Original and Extended Performance

This section compares the original and the extended version of LB4MPI. The Mandelbrot application was compiled with both the original and extended library. Five repetitions of the synchronized version of the Mandelbrot application are compared. The configuration used is the same as displayed in the table of experiments. The results in Figure 5.4 show

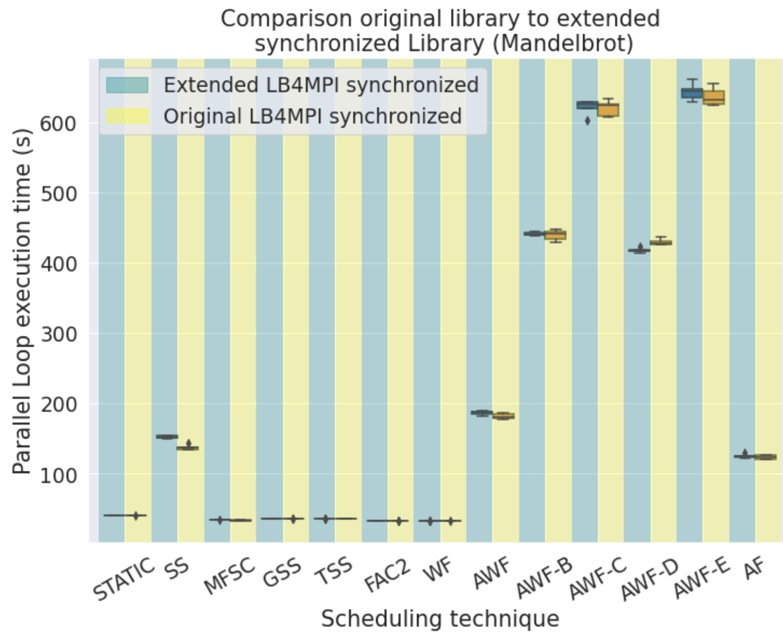


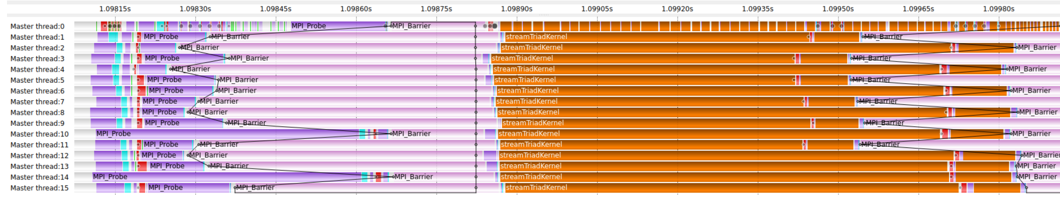
Figure 5.4: Boxplot comparing synchronous execution with the extended library to synchronous execution with the original library. Results for the original library are indicated by a yellow background while the results of the extended library are indicated by a light blue background. The x-axis shows the scheduling techniques and the y-axis shows the parallel loop execution time.

that the additional scheduling overhead produced by the extension is minimal. The almost equal performance indicates that no bugs were introduced by the extension when executing loops synchronously. This raises the confidence in the obtained and discussed results.

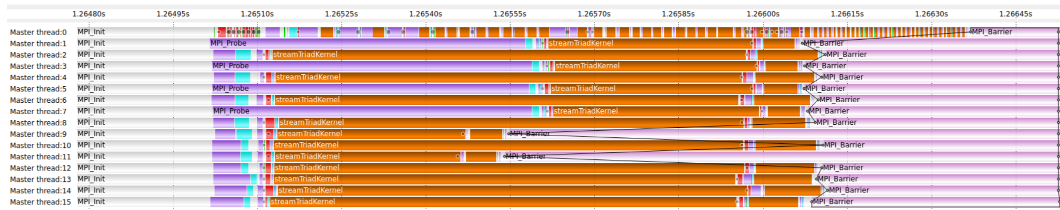
5.5.3 Scheduling Visualization

Parallel applications are more challenging to understand than sequential applications. Processes can be at different points in a program at a specific point in time. This is also the case when loops are scheduled dynamically. Score-P is a performance measurement infrastructure for profiling, event tracing, and online analysis of parallel HPC applications

such as the applications used in this performance analysis [4]. This infrastructure was utilized to create traces of the applications used in this thesis. The traces are subsequently visualized with Vampir [5]. It allows seeing the interleaved chunk executions of multiple loops when loops are executed asynchronously.



(a) Static scheduling, synchronous



(b) Static scheduling, asynchronous

Figure 5.5: Visualization of the Pi-Solver and STREAM application of one time-step. For both figures static scheduling was used. Orange marks STREAM Triad while green (barely visible) marks Pi-Solver. Time spent in an MPI_Barrier and MPI_Probe is indicated by pink and purple color, respectively. Red indicates MPI_Send and light blue indicates MPI_Recv.

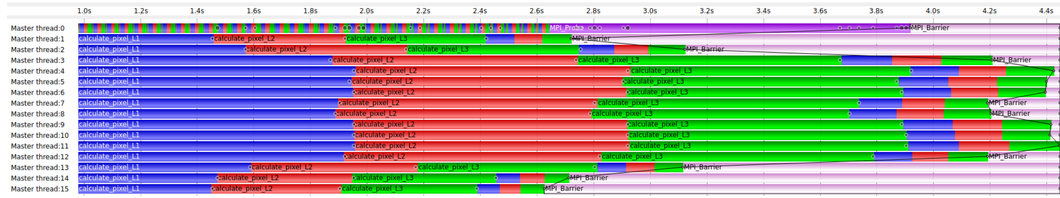
Figures 5.5a and 5.5b show one time-step of the Pi-Solver and STREAM application. The execution time of Pi-Solver iterations only takes a fraction of the STEAM iteration execution time, even if the number of loop iterations for Pi-Solver is increased significantly. Both figures also show that, as suspected in the discussion of the performance results, a significant time is spent probing for messages and waiting to synchronize. This could indicate a poor choice of parameters chosen for the this application supporting the inconclusive result in Section 5.5.1. Nevertheless, it can be observed that the loops are executed asynchronously as Fig. 5.5b shows one set of MPI_Barriers, while Fig. 5.5a shows two sets of MPI_Barriers.

The Mandelbrot application allowed for more illustrative visualizations. Figures 5.6a and 5.6b show the results of static scheduling. Synchronization among loops shows little difference in visualizations when comparing scheduling techniques. Asynchronous execution, on the other hand, produces visualizations that can differ drastically with the scheduling technique applied shown in Figures 5.6b to 5.6d. Static and dynamic non-adaptive techniques show a clear structure, while adaptive techniques appear quite chaotic. This behaviour can be explained by adaptive chunk calculation for each worker.

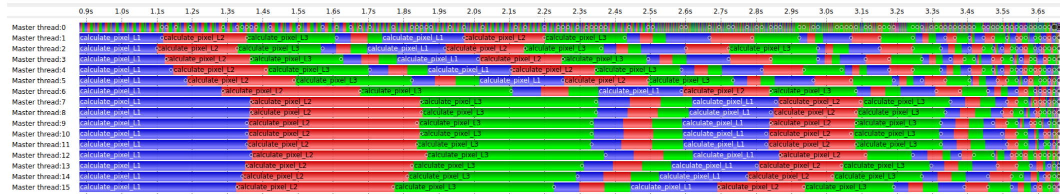
Visualizations of further scheduling techniques are provided in Appendix A.2

Figure 5.6: Visualization of the Mandelbrot application of one time-step. Blue marks L1, red marks L2, and green marks L3. Time spent in an MPI_Barrier and MPI_Probe is indicated by pink and purple color, respectively.

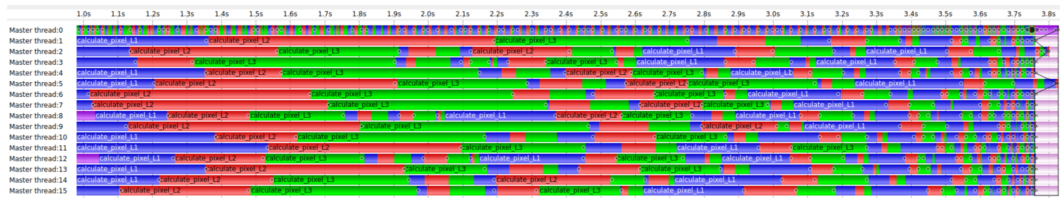
(a) Static scheduling, synchronous



(b) Static scheduling, asynchronous



(c) FAC2, asynchronous



(d) AWF-B, asynchronous

6 Conclusion

The main contributions of this thesis are the extension of LB4MPI, allowing asynchronous execution of multiple loops, and a performance analysis comparing synchronous to asynchronous execution of multiple loops. The results showed that applications with high load-imbalance static and dynamic non-adaptive scheduling techniques perform better when the synchronization between loops is relaxed, and multiple loops are executed asynchronously. The opposite has been observed for dynamic adaptive scheduling techniques, where asynchronous execution worsens the performance. The results also show that synchronous execution performs better than asynchronous execution in applications with low load-imbalance performance. Results regarding the impact on applications' performance where kernels use different system resources were not conclusive.

6.1 Future Work

There are still possible optimizations in the way the extension is implemented. Currently, when a worker who has no current chunk to execute for the targeted loop, it has to wait in DLS_StartChunk until some message is received. This probing for messages could be further relaxed such that a worker performs a non-blocking probe and leaves DLS_StartChunk immediately to directly move on to the next loop where work has possibly already been assigned.

The performance evaluation was conducted where for each experiment, only a single scheduling technique was used. Evaluating combinations of scheduling techniques in an experiment was out of scope for this thesis but could present an exciting direction for future research.

LB4MPI allows tuning the behavior of the workers, such as how early new work is requested or how often the foreman stops executing chunks to check for new messages. The experiments used the default parameters. Exploration of different such parameters may offer further insight.

Additionally, only two quite simple applications were used to evaluate the performance. More complex applications with different properties could be part of future research—especially applications where different system resources are used.

Bibliography

[1] MEMORY BANDWIDTH: STREAM BENCHMARK PERFORMANCE RESULTS, . URL https://www.cs.virginia.edu/stream/.

[2] Message Passing Interface, . URL https://www.mcs.anl.gov/research/projects/mpi/.

[3] miniHPC | High Performance Computing Group, . URL https://hpc.dmi.unibas.ch/en/research/minihpc/.

[4] Score-P - HPC Wiki, . URL https://hpc-wiki.info/hpc/Score-P.

[5] Vampir - Visualization and Analysis of Parallel Applications, . URL https://tu-dresden.de/zih/forschung/projekte/vampir/index?set_language=en.

[6] DLS4LB, May 2022. URL https://github.com/unibas-dmi-hpc/DLS4LB. original-date: 2019-08-05T15:20:08Z.

[7] M. Balasubramaniam, K. Barker, I. Banicescu, N. Chrisochoides, J.P. Pabico, and R.L. Carino. A novel dynamic load balancing library for cluster computing. In *Third International Symposium on Parallel and Distributed Computing/Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, pages 346–353, July 2004. doi: 10.1109/ISPDC.2004.5.

[8] Ioana Banicescu and Zhijun Liu. Adaptive factoring: A dynamic scheduling method tuned to the rate of weight changes. In *Proc. of the High Performance Computing Symposium*, pages 122–129, 2000.

[9] Ioana Banicescu, Vijay Velusamy, and Johnny Devaprasad. On the Scalability of Dynamic Scheduling Scientific Applications with Adaptive Weighted Factoring. *Cluster Computing*, 6:215–226, July 2003. doi: 10.1023/A:1023588520138.

[10] Ioana Banicescu, Florina M. Ciorba, and Srishti Srivastava. Scalable Computing: Theory and Practice. Number Chapter 22, pages 437–466. John Wiley&Sons, Inc., 2013. Section: Performance Optimization of Scientific Applications using an Autonomic Computing Approach.

[11] Ricolindo L. Carino and Ioana Banicescu. A tool for a two-level dynamic load balancing strategy in scientific applications. *Scalable Computing: Practice and Experience*, 8(3), 2007. ISSN 1895-1767. URL https://www.scpe.org/index.php/scpe/article/view/417.

[12] Ricolindo Cariño and Ioana Banicescu. Dynamic load balancing with adaptive factoring methods in scientific applications. *The Journal of Supercomputing*, 44:41–63, April 2008. doi: 10.1007/s11227-007-0148-y.

[13] A.T. Chronopoulos, R. Andonie, M. Benche, and D. Grosu. A class of loop self-scheduling for heterogeneous clusters. In *Proceedings 2001 IEEE International Conference on Cluster Computing*, pages 282–291, October 2001. doi: 10.1109/CLUSTR.2001.959989.

[14] A.T. Chronopoulos, S. Penmatsa, and Ning Yu. Scalable loop self-scheduling schemes for heterogeneous clusters. In *Proceedings. IEEE International Conference on Cluster Computing*, pages 353–359, September 2002. doi: 10.1109/CLUSTR.2002.1137767.

[15] Ahmed Eleliemy and Florina M. Ciorba. A Distributed Chunk Calculation Approach for Self-scheduling of Parallel Applications on Distributed-memory Systems. *arXiv:2101.07050 [cs]*, January 2021. URL http://arxiv.org/abs/2101.07050. arXiv: 2101.07050.

[16] Ahmed Hamdy Mohamed Eleliemy. *Multilevel Scheduling of Computations on Parallel Large-scale Systems*. Thesis, University_of_Basel, 2021. URL https://edoc.unibas.ch/82695/.

[17] Z. Fang, P. Tang, P.-C. Yew, and C.-Q. Zhu. Dynamic processor self-scheduling for general parallel nested loops. *IEEE Transactions on Computers*, 39(7):919–929, July 1990. ISSN 1557-9956. doi: 10.1109/12.55693. Conference Name: IEEE Transactions on Computers.

[18] Susan Flynn Hummel, Edith Schonberg, and Lawrence E. Flynn. Factoring: a method for scheduling parallel loops. *Communications of the ACM*, 35(8):90–101, August 1992. ISSN 0001-0782. doi: 10.1145/135226.135232. URL https://doi.org/10.1145/135226.135232.

[19] Susan Flynn Hummel, Jeanette Schmidt, R. N. Uma, and Joel Wein. Load-sharing in heterogeneous systems via weighted factoring. In *Proceedings of the eighth annual ACM symposium on Parallel Algorithms and Architectures*, SPAA '96, pages 318–328, New York, NY, USA, June 1996. Association for Computing Machinery. ISBN 978-0-89791-809-1. doi: 10.1145/237502.237576. URL https://doi.org/10.1145/237502.237576.

[20] Hui Li, Sudarsan Tandri, Michael Stumm, and Kenneth C. Sevcik. Locality and Loop Scheduling on NUMA Multiprocessors. In *1993 International Conference on Parallel Processing - ICPP'93*, volume 2, pages 140–147, August 1993. doi: 10.1109/ICPP.1993.112. ISSN: 0190-3918.

[21] Jie Liu, Vikram A. Saletore, and Ted G. Lewis. Safe self-scheduling: A parallel loop scheduling scheme for shared-memory multiprocessors. *International Journal of Parallel Programming*, 22(6):589–616, December 1994. ISSN 1573-7640. doi: 10.1007/BF02577870. URL https://doi.org/10.1007/BF02577870.

[22] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, 36(12):1425–1439, December 1987. ISSN 0018-9340. doi: 10.1109/TC.1987.5009495. URL https://doi.org/10.1109/TC.1987.5009495.

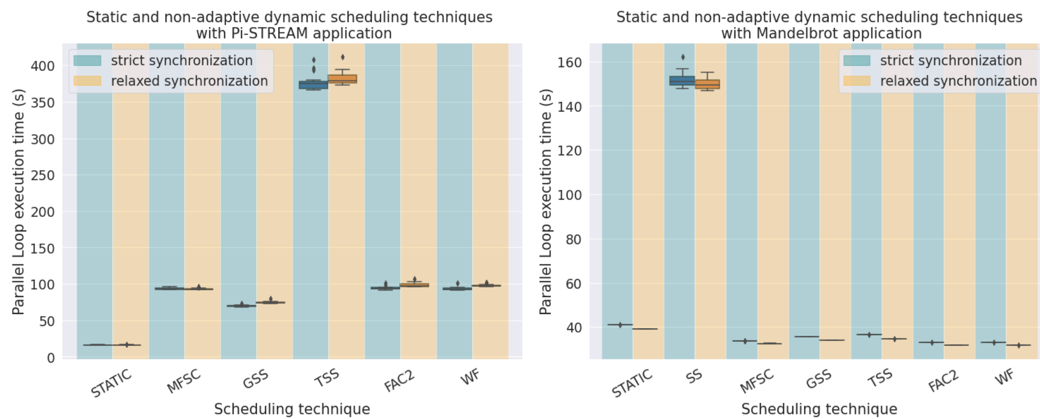
[23] P. Tang and P. C. Yew. Processor self-scheduling for multiple-nested parallel loops. Technical Report DOE/ER/25001-3, Illinois Univ., Urbana (USA). Center for Supercomputing Research and Development, January 1986. URL https://www.osti.gov/biblio/7258138-processor-self-scheduling-multiple-nested-parallel-loops.

[24] T.H. Tzen and L.M. Ni. Trapezoid self-scheduling: a practical scheduling scheme for parallel compilers. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):87–98, January 1993. ISSN 1558-2183. doi: 10.1109/71.205655. Conference Name: IEEE Transactions on Parallel and Distributed Systems.

[25] Eric W. Weisstein. Mandelbrot Set. URL https://mathworld.wolfram.com/. Publisher: Wolfram Research, Inc.

A
Figures

A.1 Performance Results



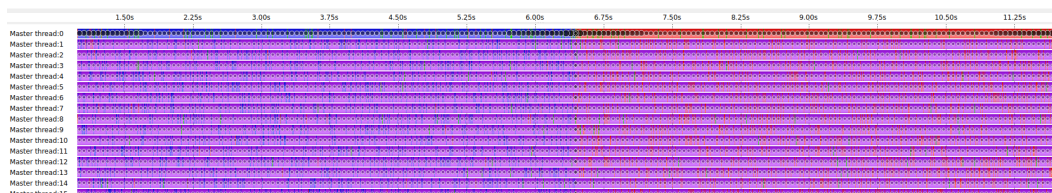
(a) Pi-STREAM, Static and non-adaptive dynamic scheduling techniques, including TSS

(b) Mandelbrot, Static and non-adaptive dynamic scheduling techniques, including SS

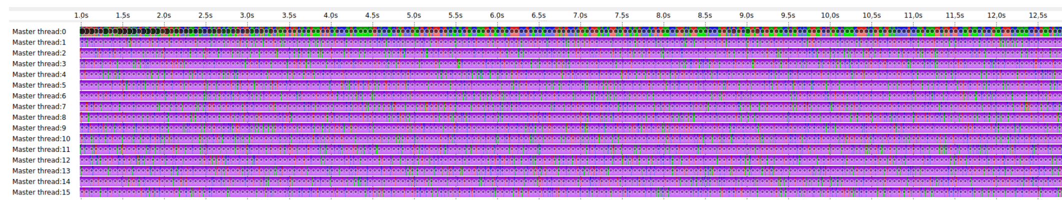
Figure A.1: Performance results including outliers. Blue background corresponds to synchronization among loops and light orange without.

A.2 Scheduling Visualization of mandelbrot application

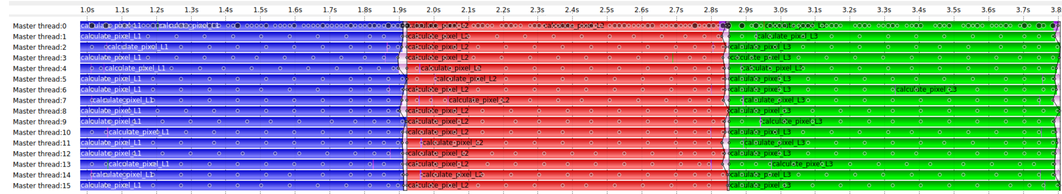
Figure A.2: Scheduling Visualizations



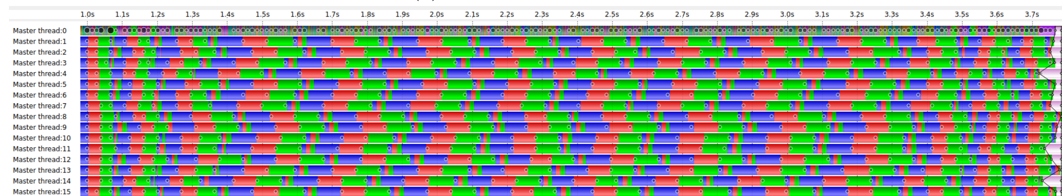
(a) SS, synchronized (only part of time step shown)



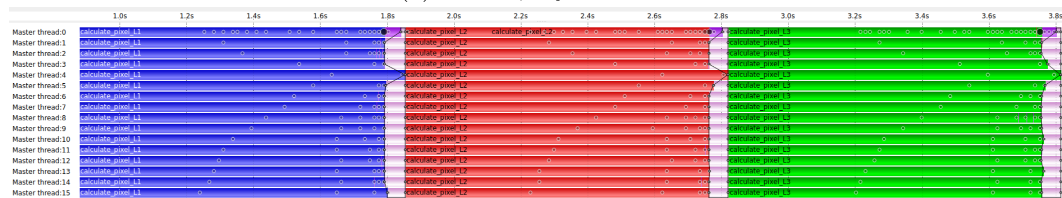
(b) SS, asynchronous (only part of time step shown)



(c) MFSC, synchronous

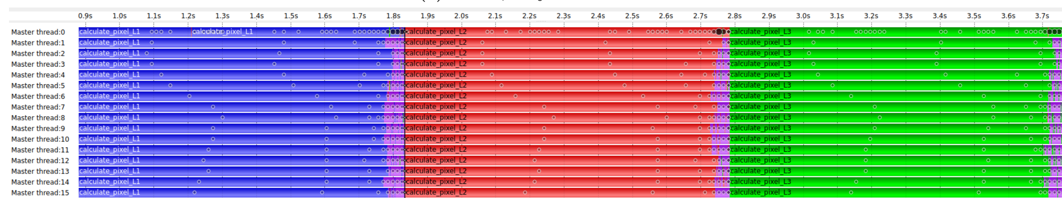


(d) MFSC , asynchronous

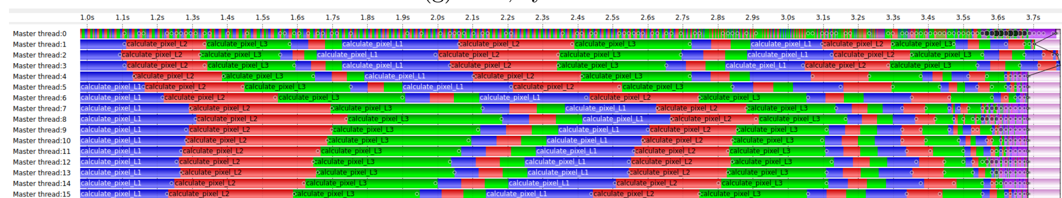


(e) GSS, synchronous

(f) GSS, asynchronous



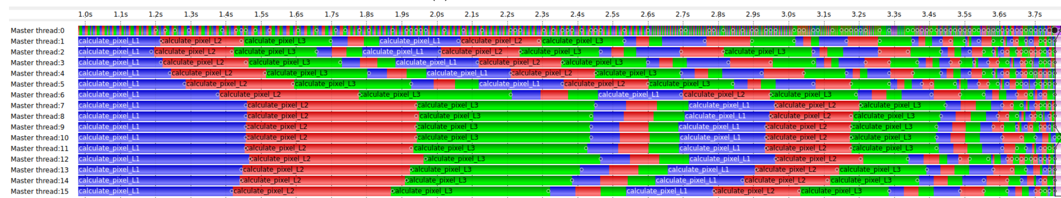
(g) TSS, synchronous



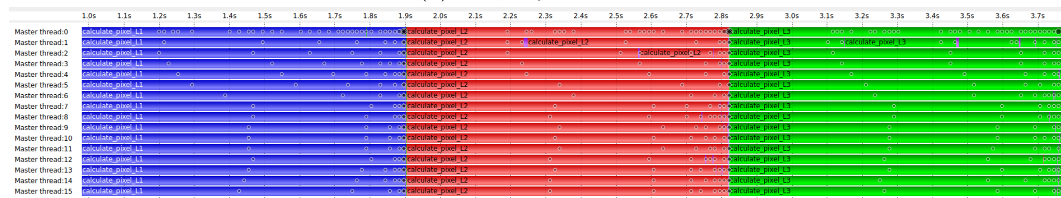
(h) TSS, asynchronous

(i) FAC2, synchronous

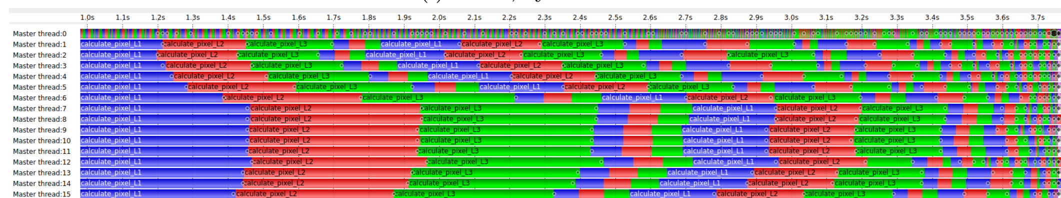
(j) WF, synchronous



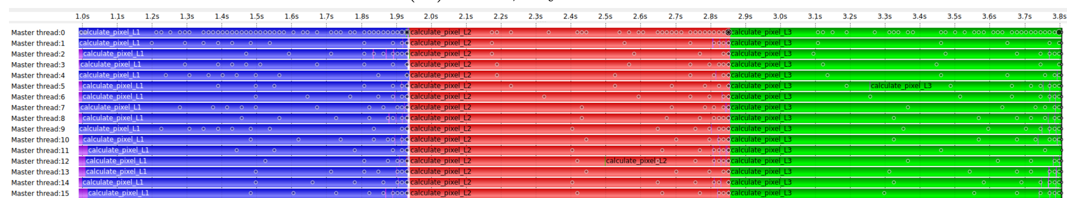
(k) WF, asynchronous



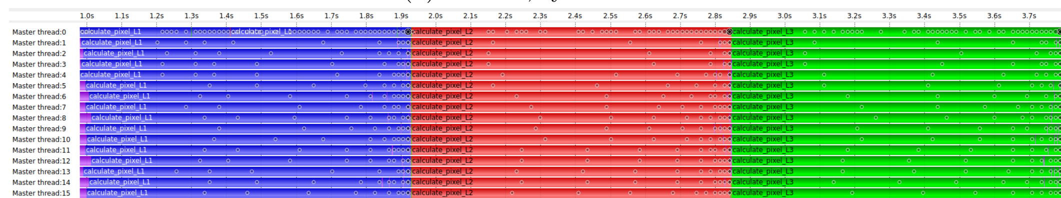
(l) AWF, synchronous



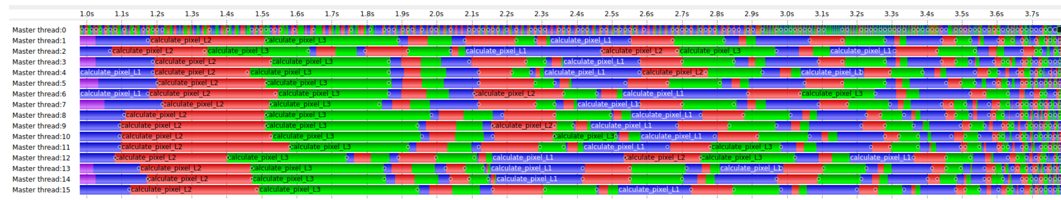
(m) AWF, asynchronous



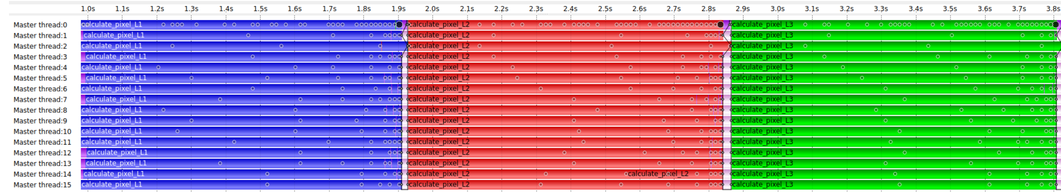
(n) AWF-B, synchronous



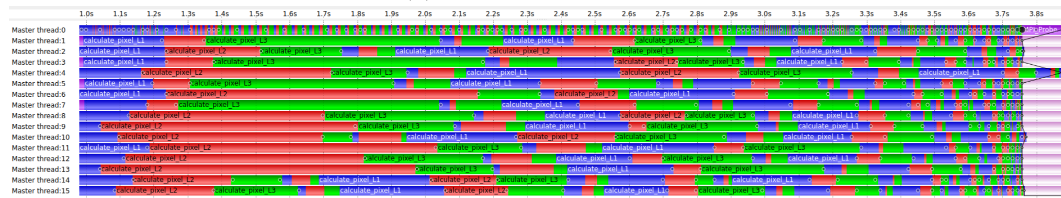
(o) AWF-C, synchronous



(p) AWF-C, asynchronous

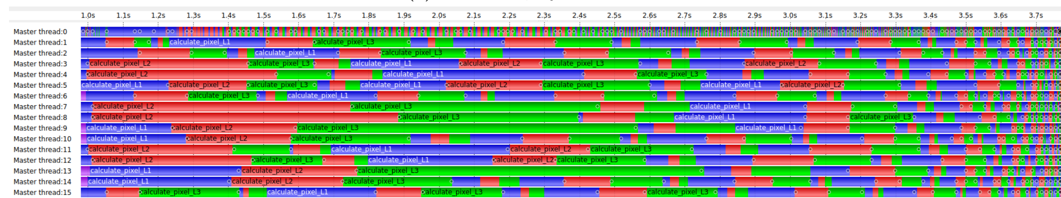


(q) AWF-D, synchronous



(r) AWF-D, asynchronous

(s) AWF-E, synchronous



(t) AWF-E, asynchronous

(u) AF, synchronous

(v) AF, asynchronous

B
Code

B.1 LB4MPI

```
void DLS_NumLoops(infoDLS *info, int n)
{
    Initialized = 1;
    info->numLoops = n;
}
```

Listing B.1: DLS_NumLoops function, used to set the number of loops to execute asynchronously

```
void DLS_Parameters_Setup( MPI_Comm icomm, infoDLS *info, int numProcs,
    int requestWhen, int breakAfter, int minChunk, double h_overhead,
    double sigma, int nKNL, double Xeon_speed, double KNL_speed )
{
    int tP;
    double total_sum = 0.0;
    double core_speed = 0.0;
    int i,j;
    // check if info->numLoops was initalized, if not we set numLoops to default (1)
    // otherwise it is already initialized by DLS_numLoop
    if (Initialized == 0)
    {
        info->numLoops = 1;
    }
    MPI_Comm_size(icomm, &tP);
    MPI_Comm_rank(icomm, &(info->myRank));
    info->comm = icomm;
    info->crew = MPI_COMM_NULL;
    info->commSize = tP;
    info->firstRank = 0;
    info->lastRank = tP-1;
    info->foreman = 0;
    info->breakAfter = breakAfter;
    info->requestWhen= requestWhen;
    info->minChunk = minChunk;
    // allocate memory for possibly multiple loops
    allocate_mem(info, info->numLoops, numProcs);

    // initialize infoDLS members
    for (i=0;i<info->numLoops;i++)
    {
        info->probeFreq[i] = -1;
        info->timeStep[i] = 0;
    }
    if (info->comm==MPI_COMM_NULL) return;

    // h and sigma for FSC orginial equation
    info->h_overhead = h_overhead;
```

```
39      info->sigma = sigma;
40      // calculate weights for WF .. assume two types of processors, Xeon and KNL
41      total_sum = nKNL * KNL_speed + (numProcs - nKNL) * Xeon_speed ;
42      for (i = 0; i < numProcs; i++)
43      {
44          if(i<(numProcs - nKNL))
45          {core_speed = Xeon_speed;}
46          else
47          {core_speed = KNL_speed; }
48          for (j = 0; j < info->numLoops; j++)
49          {
50              //initialize mu, sigma, and performance data count
51              info->stats[3*i+j*info->numLoops]   = -1; //mu
52              info->stats[3*i+1+j*info->numLoops] = -1; //sigma
53              info->stats[3*i+2+j*info->numLoops] = 0; //performance data count
54              // initialize weights
55              info->weights[i+j*info->numLoops] =  core_speed/total_sum * numProcs;
56          }
57
58
59      }
60  }
```

Listing B.2: DLS_Parameters_Setup function, modified to initialize members of for multiple loops.

```
1   void DLS_StartMLoops(infoDLS *info, int *firstIter, int *lastIter, int *imeths)
2   {
3       int tSize, worker;
4       int endedLoop;
5       double K;
6       int i;
7       double awap, trw;
8       int numLoops = info->numLoops;
9
10
11      memcpy(info->firstIter, firstIter, numLoops*sizeof(int));
12      memcpy(info->lastIter, lastIter, numLoops*sizeof(int));
13      memcpy(info->method, imeths, numLoops*sizeof(int));
14
15
16      for (int l = 0; l < info->numLoops; l++)
17      {
18          info->tExclude[l] = 0.0;
19          info->wSize[l] = 0;  /* remaining iterates in current chunk */
20          info->gotWork[l] = 1; /*.true.; */
21          info->workTime[l] = 0.0;
22          info->myIters[l] = 0;
23          info->N[l] = lastIter[l] - firstIter[l] + 1;
24          info->timeStep[l] = info->timeStep[l] + 1;
25          // check if range of iterations is valid and MPI comm exists,
26          // otherwise continue with next loop
27          if ( (info->comm==MPI_COMM_NULL) || (info->N[l]<=0) ) continue;
28
29          if ( (info->method[l]>DLS_MethodCount-1) || (info->method[l]<0) )
30              info->method[l] = 0;
31
32
33
34          info->TSSchunk[l] = ceil((double) info->N[l] / ((double) 2*info->commSize));
35          int n = ceil(2*info->N[l]/(info->TSSchunk[l]+1)); //n=2N/f+l
36          info->TSSdelta[l] = (double) (info->TSSchunk[l] - 1)/(double) (n-1);
37          //calculate AWF weights
38          if ( (info->method[l] == AWF) && (info->myRank == info->foreman))
39          {
40              if (info->timeStep[l] == 1) //first timeStep
41              {
42                  for(i = info->firstRank; i <= info->lastRank; i++)
43                  {
44                      info->weights[l*info->numLoops+i] = 1.0;
45                  }
46              }
```

```
47             else // all ranks have wap
48             {
49                 awap = 0.0;  // average weighted performance
50
51                 for(i = info->firstRank; i <= info->lastRank; i++)
52                 {
53                     //printf("rank %d: %lf", i, info->stat[3*i]);
54                     awap = awap + info->stats[l*numLoops+3*i];
55                 }
56                 awap = awap/info->commSize;
57                 trw = 0.0;  // total ref weight (refwt(i) = awap/info%stats(3*i)
58
59                 for(i = info->firstRank; i <= info->lastRank; i++)
60                 {
61                     trw = trw + awap/info->stats[l*numLoops+3*i];
62                 }
63
64                 for(i = info->firstRank; i <= info->lastRank; i++)
65                 {
66                     info->weights[l*numLoops+i] = ((awap/info->stats[l*numLoops+3*i])*
67                         info->commSize)/trw;
68                 }
69             }
70         }
71         // numChunks doesn't seem to be used anywhere
72         info->numChunks = 0;
73         info->myExecs[l] = 0;
74         info->mySumTimes[l] = 0.0;
75         info->mySumSizes[l] = 0.0;
76         tSize = (info->N[l]+info->commSize-1)/info->commSize;
77         info->chunkMFSC[l] = (0.55+tSize*log(2.0)/log( (1.0*tSize) ) );
78         info->kopt0[l] = sqrt(2.0)*info->N[l]/
79             ( info->commSize*sqrt(log(1.0*info->commSize)) );
80
81         // calculate FSC chunk
82         K=(sqrt(2)*info->N[l]*info->h_overhead)/
83             (info->sigma*info->commSize*sqrt(log(info->commSize)));
84         K=pow(K, 2.0/3.0);
85
86
87
88         info->chunkFSC[l] = (int) ceil(K);
89         info->nextWRKrcvd[l] = 0;
90         info->req4WRKsent[l] = 0;
91         info->finishedOne[l] = 0;
92
93         info->probeFreq[l] = max(1, info->breakAfter);
94         info->sendRequest[l] = max(1, info->requestWhen);
95         if (info->myRank == info->foreman)
96         {
97             info->chunkStart[l] = firstIter[l];
98             info->itersScheduled[l] = 0;
99             info->batchSize[l] = 0;
100            info->batchRem[l] = 0;
101            info->numENDed[l] = 0;
102
103            if (info->minChunk>0)
104                info->minChunkSize[l] = info->minChunk;
105            else
106                /* default min chunk size */
107                info->minChunkSize[l] = max(1,info->chunkMFSC[l]/2);
108            info->maxChunkSize[l] = (info->N[l]+2*info->commSize-1)/(2*info->commSize);
109
110            // set curLoop such that the chunks for the right loop are sent
111            info->curLoop = l;
112            /* send initial work to each processor */
113            for (worker = info->firstRank; worker <= info->lastRank; worker++)
114            {
115                if (info->chunkStart[l] < info->lastIter[l])
116                {
117                    SendChunk (info, worker);
118                }
119                else
```

```
120                 {
121                     endedLoop = 1;
122                     //end worker
123                     MPI_Send (&endedLoop, 1, MPI_INT, worker, END_TAG, info->comm);
124                     info->numENDed[l]++; // increment ended workers
125                 }
126             }
127         }
128     }
129     // reset curLoop to first loop by default
130     info->curLoop = 0;
131 }
```

Listing B.3: DLS_StartMLoops function, initialized loop-specific members for one or more loops. Sends out first chunk to every worker.

```
1  int DLS_MTerminated(infoDLS *info)
2  {
3      int i, done;
4      MPI_Status tStatus;
5      double t0;
6      int numNoIterates = 0;
7      int sumGotWork = 0;
8      int sumWSize = 0;
9      for (i=0; i<info->numLoops; i++)
10     {
11         if (info->N[i] <= 0) {
12             numNoIterates++;
13         }
14         sumGotWork += info->gotWork[i];
15         sumWSize += info->wSize[i];
16     }
17     if (numNoIterates == info->numLoops)
18     {
19         // all loops have no iterates!
20         done=1;
21     }
22     else
23     {
24         if ( (info->comm==MPI_COMM_NULL) && (info->crew!=MPI_COMM_NULL) )
25             MPI_Recv (&done, 1, MPI_INT, 0, TRM_TAG, info->crew, &tStatus);
26         else if ( (info->comm!=MPI_COMM_NULL) && (info->crew!=MPI_COMM_NULL) )
27         {
28             done = (sumGotWork==0) && (sumWSize==0);
29             for (i=1;i<info->crewSize;i++)
30                 MPI_Send (&done, 1, MPI_INT, i, TRM_TAG, info->crew);
31         }
32         else done = (sumGotWork==0) && (sumWSize==0);
33     }
34     return (done);
35 }
```

Listing B.4: DLS_MTerminated function, checks whether all loops have been completly scheduled.

```
1  void DLS_TargetLoop(infoDLS *info, int l)
2  {
3      info->curLoop = l;
4  }
```

Listing B.5: DLS_TargetLoop function, used to set certain loops as the next target loop.

```
1  void DLS_StartChunk (infoDLS *info, int *chunkStart, int *chunkSize)
2  {
3      int tSize, tStart, worker,reqLoop;
4      int MsgInQueue;              /* message came in */
5      int loc, maxRemaining;       /* source of chunk to be migrated */
6      int i, j, endedLoop, chunkInfo[3];
7      double perfInfo[4];
```

```
8       MPI_Status mStatus, tStatus;
9       int target = info->curLoop;
10      int recvLoop;
11
12      if (info->comm==MPI_COMM_NULL) { /* I'm just a simple worker */
13          MPI_Recv (chunkInfo, 2, MPI_INT, 0, WRK_TAG, info->crew, &tStatus);
14          *chunkStart = chunkInfo[0];
15          *chunkSize = chunkInfo[1];
16      }
17      else { /* I'm the coordinator, or a foreman */
18          if (info->wSize[target] == 0 && info->gotWork[target]) {
19              MPI_Probe (MPI_ANY_SOURCE, MPI_ANY_TAG, info->comm, &mStatus);
20              MsgInQueue = 1; /*.true. */
21          }
22          else
23              MPI_Iprobe (MPI_ANY_SOURCE, MPI_ANY_TAG, info->comm, &MsgInQueue, &mStatus);
24
25          while (MsgInQueue) {
26
27              switch ( mStatus.MPI_TAG ) {
28
29              case (WRK_TAG):
30                  MPI_Recv (chunkInfo, 3, MPI_INT, mStatus.MPI_SOURCE, WRK_TAG, info->comm, &tStatus);
31                  // What loop did I receive work for?
32                  recvLoop = chunkInfo[2];
33
34                  if (info->wSize[recvLoop] == 0) { /* no pending chunk */
35                      info->t0[recvLoop] = MPI_Wtime(); /* elapsed time for chunk starts here */
36                      info->tExclude[recvLoop] = 0.0;
37                      info->wStart[recvLoop] = chunkInfo[0];
38                      info->wSize[recvLoop] = chunkInfo[1];
39                      info->rStart[recvLoop] = info->wStart[recvLoop];
40                      info->rSize[recvLoop] = info->wSize[recvLoop];
41                      info->req4WRKsent[recvLoop] = 0; /* cancel request for work */
42
43                      //set info->curLoop to recvLoop to set breaks correctly
44                      info->curLoop = recvLoop;
45                      SetBreaks (info);
46                      //reset info->curLoop to original state
47                      info->curLoop = target;
48                      info->sumt1[recvLoop] = 0.0; /* for mu/wap */
49                      info->sumt2[recvLoop] = 0.0; /* for sigma */
50                  }
51                  else { /* current chunk is not finished  save as next chunk */
52                      info->nextStart[recvLoop] = chunkInfo[0];
53                      info->nextSize[recvLoop] = chunkInfo[1];
54                      info->nextWRKrcvd[recvLoop] = 1; /*.true. */
55                  }
56                  break;
57
58              case (REQ_TAG): /* received by foreman only */
59                  worker = mStatus.MPI_SOURCE;
60                  MPI_Recv (perfInfo, 4, MPI_DOUBLE, worker, REQ_TAG, info->comm, &tStatus);
61                  // What loop did I receive a request for?
62                  recvLoop = (int) perfInfo[3];
63
64
65                  if ( (info->method[recvLoop]==AF) || (info->method[recvLoop]==AWF_B) ||
66                  (info->method[recvLoop]==AWF_C) ||
67                      (info->method[recvLoop]==AWF_D) || (info->method[recvLoop]==AWF_E) ) {
68                      loc = perfInfo[2];
69                      info->stats[3*loc+2+recvLoop*info->numLoops] =
70                      info->stats[3*loc+2+recvLoop*info->numLoops]+1.0;
71                      /* adaptive methods */
72                      info->stats[3*loc+recvLoop*info->numLoops] = perfInfo[0];
73                      info->stats[3*loc+1+recvLoop*info->numLoops] = perfInfo[1];
74
75                      if (info->finishedOne[recvLoop] != info->commSize) {
76                          /* workers that have not finished a first chunk */
77                          /*  assume the lowest performance */
78                          j = loc;
79
80                          for (i=info->firstRank;i<=info->lastRank;i++)
```

```
81                                  if ( (info->stats[3*i+2+recvLoop*info->numLoops] > 0.0) &&
82                                      (info->stats[3*i+recvLoop*info->numLoops] <
83                                          info->stats[3*j+recvLoop*info->numLoops]) ) j = i;
84                              info->finishedOne[recvLoop] = 0;
85                              for (i=info->firstRank;i<=info->lastRank;i++)
86                                  if (info->stats[3*i+2+recvLoop*info->numLoops] == 0.0) {
87
88                                      //set my stats to those stats to minimum
89                                      info->stats[3*i+recvLoop*info->numLoops] =
90                                          info->stats[3*j+recvLoop*info->numLoops];
91                                      info->stats[3*i+1+recvLoop*info->numLoops] =
92                                          info->stats[3*j+1+recvLoop*info->numLoops];
93
94                                  }
95                                  else
96                                      info->finishedOne[recvLoop] = info->finishedOne[recvLoop] + 1;
97                          }
98
99                      } /* if (AWF methods) */
100
101                     /* any remaining unscheduled iterates ? */
102                     if (info->chunkStart[recvLoop] <= info->lastIter[recvLoop]){
103
104                         //set curLoop to recvLoop
105                         info->curLoop = recvLoop;
106                         SendChunk (info, worker);
107                         // reset curLoop
108                         info->curLoop = target;
109                     }
110                     else { /* all iterates scheduled */
111                         info->numENDed[recvLoop] = info->numENDed[recvLoop] + 1;
112                         if (worker != info->myRank) {
113                             endedLoop = recvLoop;
114                             MPI_Send (&endedLoop, 1, MPI_INT, worker, END_TAG, info->comm);
115                         }
116                         /* foreman exits? */
117                         info->gotWork[recvLoop] = (info->numENDed[recvLoop]!=info->commSize);
118                     }
119                     break;
120
121                 case (END_TAG): /* received by workers only */
122
123                     MPI_Recv (&endedLoop, 1, MPI_INT, mStatus.MPI_SOURCE,
124                         mStatus.MPI_TAG, info->comm, &tStatus);
125                     info->gotWork[endedLoop] = 0;
126                     break;
127
128                 } /* switch */
129                 MPI_Iprobe (MPI_ANY_SOURCE, MPI_ANY_TAG, info->comm, &MsgInQueue, &mStatus);
130             } /* while (MsgInQueue) */
131
132             *chunkStart = info->wStart[target];
133             *chunkSize = min (info->wSize[target], info->probeFreq[target]);
134             if (info->method[target] == AF) *chunkSize = min(1, *chunkSize);
135             info->subChunkSize[target] = *chunkSize;
136             if (info->subChunkSize[target]!=0) info->t1[target] = MPI_Wtime();
137
138             /* relay chunkStart, chunkSize to icomm */
139             if (info->crew!=MPI_COMM_NULL) {
140                 chunkInfo[0] = *chunkStart;
141                 chunkInfo[1] = *chunkSize;
142                 chunkInfo[2] = target;
143                 for (i=1;i<info->crewSize;i++)
144                     MPI_Send (chunkInfo, 3, MPI_INT, i, WRK_TAG, info->crew);
145             }
146         } /* (info->comm!=MPI_COMM_NULL) { */
147     }
```

Listing B.6: DLS_StartChunk function, messages are received and responded to in this function. Chunk start and size are written into chunkStart and chunkSize.

```
1   void DLS_EndChunk (infoDLS *info)
```

```
2  {
3      double tk, perfInfo[4];
4      int loc;
5      int i, j;
6      int target = info->curLoop;
7
8      if (info->comm==MPI_COMM_NULL) return;
9
10     if (info->subChunkSize[target]==0) return;
11
12     tk = MPI_Wtime();
13     info->t1[target] = tk - info->t1[target];
14     info->wStart[target] = info->wStart[target] + info->subChunkSize[target];
15     info->wSize[target] = info->wSize[target] - info->subChunkSize[target];
16     // add t1 to all elements of tExclude except for target loop
17     // later used to exclude this accumulated time for AWF-D and AWF-E methods
18     for (i=0; i<info->numLoops; i++) {
19         if (i != target) {
20             info->tExclude[i] += info->t1[target];
21         }
22     }
23
24     info->sumt1[target] = info->sumt1[target] + info->t1[target];
25     info->workTime[target] = info->workTime[target] + info->t1[target];
26     if (info->method[target] == AF)
27         info->sumt2[target] = info->sumt2[target] + info->t1[target]*info->t1[target];
28
29     if (info->wSize[target] == 0) { /* chunk finished */
30
31         if ( (info->method[target]==AWF_B) || (info->method[target]==AWF_C) ) {
32             /* adaptive weighted factoring, work time */
33             info->mySumTimes[target] = info->mySumTimes[target] +
34                 (1+info->myExecs[target])*info->sumt1[target];
35             info->mySumSizes[target] = info->mySumSizes[target] +
36                 (1.0+info->myExecs[target])*info->rSize[target];
37         }
38         else if ( (info->method[target]==AWF_D) || (info->method[target]==AWF_E) ) {
39             /* adaptive weighted factoring, elapsed time */
40             info->mySumTimes[target] = info->mySumTimes[target] +
41                 (1+info->myExecs[target])*(tk-info->t0[target]-info->tExclude[target]);
42             info->mySumSizes[target] = info->mySumSizes[target] +
43                 (1.0+info->myExecs[target])*info->rSize[target];
44         }
45
46         if(info->method[target] !=AF)
47         {
48             /* reset accumulators */
49             info->myIters[target] = info->myIters[target] + info->rSize[target];
50             info->myExecs[target] = info->myExecs[target] + 1;
51             info->sumt1[target] = 0.0; /* for mu */
52             info->sumt2[target] = 0.0; /* for sigma */
53             info->rSize[target] = 0;
54         }
55
56         if (info->nextWRKrcvd[target]) { /* foreman already responded to advance request */
57             info->t0[target] = MPI_Wtime(); /* elapsed time for chunk starts here */
58             info->tExclude[target] = 0.0;
59             info->wStart[target] = info->nextStart[target];
60             info->wSize[target] = info->nextSize[target];
61             info->rStart[target] = info->wStart[target];
62             info->rSize[target] = info->wSize[target];
63
64             SetBreaks (info);
65
66             info->nextSize[target] = 0;
67             info->nextWRKrcvd[target] = 0;
68             info->req4WRKsent[target] = 0;
69         }
70     } /* if (info->wSize == 0) */
71
72
73
74
```

```
75      /* send request ? */
76      if ( (info->wSize[target]<=info->sendRequest[target]) && (info->req4WRKsent[target]==0) ) {
77          perfInfo[3] = 1.0*info->curLoop;
78          switch (info->method[target]) {
79              case STATIC:
80              case SS:
81              case FSC:
82              case mFSC:
83              case GSS:
84              case TSS:
85              case FAC:
86              case WF:
87              case AWF:
88                  perfInfo[0] = 0.0;
89                  perfInfo[1] = 0.0;
90                  perfInfo[2] = 1.0*info->myRank;
91                  perfInfo[3] = 1.0*info->curLoop;
92                  break;
93
94              case AWF_B:
95              case AWF_C: /* mu = (chunk work time)/(chunk size) */
96                  perfInfo[0] = ( info->mySumTimes[target] + (info->myExecs[target]+1)*info->sumt1[target] )/
97                      ( info->mySumSizes[target] + 1.0*
98                          (info->myExecs[target]+1)*(info->rSize[target]-info->wSize[target]) );
99                  perfInfo[1] = 0.0;
100                 perfInfo[2] = 1.0*info->myRank;
101                 perfInfo[3] = 1.0*target;
102                 break;
103
104             case AF:
105                 perfInfo[0] = info->sumt1[target]/(info->rSize[target]-info->wSize[target]);
/* mu */
106                 if ((info->rSize[target]-info->wSize[target]) > 1) {
107                     perfInfo[1] = (info->sumt2[target] -
108                         perfInfo[0]*perfInfo[0]*(info->rSize[target]-info->wSize[target])) /
109                     (info->rSize[target]-info->wSize[target]-1); /* sigma */
110                     if (perfInfo[1] < 0.0) perfInfo[1] = 0.0;
111                     perfInfo[1] = sqrt(perfInfo[1]);
112                 }
113                 else perfInfo[1] = 0.0;
114                 perfInfo[2] = 1.0*info->myRank;
115                 perfInfo[3] = 1.0*target;
116
117             if (info->wSize[target] == 0)
118             {  // reset accumulators
119                 info->myIters[target] = info->myIters[target] + info->rSize[target];
120                 info->myExecs[target] = info->myExecs[target] + 1;
121                 info->sumt1[target] = 0.0;  // for mu
122                 info->sumt2[target] = 0.0;  // for sigma
123                 info->rSize[target] = 0;
124             }
125             break;
126
127             case AWF_D:
128             case AWF_E: /* mu = (chunk elapsed time)/(chunk size) */
129                 perfInfo[0] = ( info->mySumTimes[target] + (info->myExecs[target]+1)*(tk-info->t0[target]-
130                     info->tExclude[target])) /
131                     ( info->mySumSizes[target] + 1.0*
132                         (info->myExecs[target]+1)*(info->rSize[target]-info->wSize[target]) );
133                 perfInfo[1] = 0.0;
134                 perfInfo[2] = 1.0*info->myRank;
135                 perfInfo[3] = 1.0*target;
136                 break;
137         }
138
139         if(info->myRank == info->foreman)
140         {   //update performance data
141             if (info->method[target]==AWF_B || info->method[target]==AWF_C ||
142                 info->method[target]==AF || info->method[target]==AWF_D ||
143                 info->method[target]==AWF_E)
144             {
145                 loc = (int) perfInfo[2];
146                 info->stats[3*loc+2+target*info->numLoops] =
```

```
147                         info->stats[3*loc+2+target*info->numLoops]+1.0;
148                     //adaptive methods
149                     info->stats[3*loc+target*info->numLoops] = perfInfo[0];
150                     info->stats[3*loc+1+target*info->numLoops] = perfInfo[1];
151 
152                     if (info->finishedOne[target] != info->commSize)
153                     {
154                         // workers that have not finished a first chunk
155                         // assume the lowest performance
156                         j = loc;
157                         for(i=info->firstRank; i<=info->lastRank; i++)
158                         {
159                             if ( (info->stats[3*i+2+target*info->numLoops] > 0.0) &&
160                                 (info->stats[3*i+target*info->numLoops] <
161                                 info->stats[3*j+target*info->numLoops]) )
162                                 j = i;
163                         }
164                         info->finishedOne[target] = 0;
165                         for( i=info->firstRank; i<=info->lastRank;i++)
166                         {
167                             if (info->stats[3*i+2+target*info->numLoops] == 0.0)
168                             {
169                                 info->stats[3*i+target*info->numLoops] =
170                                     info->stats[3*j+target*info->numLoops];
171                                 info->stats[3*i+1+target*info->numLoops] =
172                                     info->stats[3*j+1+target*info->numLoops];
173                             }
174                             else
175                             {
176                                 info->finishedOne[target] = info->finishedOne[target] + 1;
177                             }
178                         }
179                     }
180                 }
181 
182                 // get more work to myself
183                 if (info->chunkStart[target] < info->lastIter[target])
184                 {
185                     info->req4WRKsent[target] = 1;
186                     info->nextWRKrcvd[target] = 0;
187                     SendChunk (info, info->myRank);
188                 }
189                 else if (info->wSize[target] == 0 && info->chunkStart[target] >= info->lastIter[target])
190                 {
191                     // all iterates scheduled
192                     info->numENDed[target] = info->numENDed[target] + 1;
193                     // foreman exits?
194                     info->gotWork[target] = info->numENDed[target]!=info->commSize;
195                 }
196         } /* if(info->myRank == info->foreman) */
197         else
198         {
199          MPI_Send (perfInfo, 4, MPI_DOUBLE, info->foreman, REQ_TAG, info->comm);
200          info->req4WRKsent[target] = 1; /*.true. */
201          info->nextWRKrcvd[target] = 0;
202         }
203     } /* if (...info->sendRequest...) */
204 }
```

Listing B.7: DLS_EndChunk, collects statistics about chunk execution and new work requests are made in this function.

```
1   void DLS_EndMLoops(infoDLS *info, int *niters, double *worktime)
2   {
3       double perfInfo[info->numLoops*3];
4       memset( perfInfo, 0, info->numLoops*3*sizeof(double) );
5       int shouldGather = 0;
6   
7       if (info->comm==MPI_COMM_NULL) return;
8   
9       memcpy(niters, info->myIters, info->numLoops*sizeof(int));
10      memcpy(worktime, info->workTime, info->numLoops*sizeof(double));
```

```
11
12      for (int l = 0; l < info->numLoops; l++)
13      {
14          // communicate time-step performance data for AWF
15          if (info->method[l]==AWF)
16          {
17              // timestepping adaptive weighted factoring
18              // mu = (chunk work time)/(chunk size)
19              shouldGather = 1;
20              perfInfo[l*3] = ( info->mySumTimes[l] + (info->timeStep[l])*info->workTime[l] ) /
21                  ( info->mySumSizes[l] + 1.0*(info->timeStep[l])*(info->myIters[l]));
22              perfInfo[l*3+1] = 0.0;
23              perfInfo[l*3+2] = 1.0*info->timeStep[l];
24          }
25      }
26
27      if (shouldGather == 1)
28      {
29          MPI_Gather(perfInfo, 3*info->numLoops, MPI_DOUBLE_PRECISION, info->stats,
30              3*info->numLoops, MPI_DOUBLE_PRECISION, info->foreman, info->comm);
31      }
32      // was commented out since otherwise result may be
33      // incorrect when using two loops and you do not explicitely synchronize the mpi ranks
34      MPI_Barrier(info->comm); // was commented ~Oli
35      // reset initialized
36      Initialized = 0;
37  }
```

Listing B.8: DLS_EndMLoops function, point of synchronization before a next time step can be computed. Provides information how many iterations a worker has executed and the worktime.

```
1   void SendChunk ( infoDLS *info, int worker )
2   {
3       // chunkInfo size changed to 3 to also send to which loop the chunk belongs to
4       int chunkSize, chunkInfo[3];
5       int target = info->curLoop;
6       GetChunkSize (info, worker, &chunkSize);
7
8       chunkInfo[0] = info->chunkStart[target];
9       chunkInfo[1] = chunkSize;
10      chunkInfo[2] = target;
11
12      if(worker == info->foreman)
13      {
14         if (info->wSize[target] == 0)  // no pending chunk
15         {
16              info->t0[target] = MPI_Wtime(); // elapsed time for chunk starts here
17              info->tExclude[target] = 0.0;
18              info->wStart[target] = chunkInfo[0];
19              info->wSize[target] = chunkInfo[1];
20              info->rStart[target] = info->wStart[target];
21              info->rSize[target] = info->wSize[target];
22              info->req4WRKsent[target] = 0;  // cancel request for work
23
24              SetBreaks(info);
25
26              info->sumt1[target] = 0.0;  //for mu/wap
27              info->sumt2[target] = 0.0;  // for sigma
28          }
29          else  //current chunk is not finished  save as next chunk
30          {
31              info->nextStart[target] = chunkInfo[0];
32              info->nextSize[target] = chunkInfo[1];
33              info->nextWRKrcvd[target] = 1;
34          }
35        }
36        else
37        {
38            MPI_Send (chunkInfo, 3, MPI_INT, worker, WRK_TAG, info->comm);
39        }
```

```
40
41      info->chunkStart[target]      = info->chunkStart[target]      + chunkSize;
42      info->itersScheduled[target] = info->itersScheduled[target] + chunkSize;
43  }
```

Listing B.9: SendChunk function, only called by foreman to send new chunks to workers, also used to assign chunks to the foreman himself.

```
1   void GetChunkSize ( infoDLS *info, int rank, int *chunkSize )
2   {
3       int i, tChunk, rem;
4       double bigD, bigT, awap, trw, weight, K;
5       // target loop is loop for which chunk size is needed
6       int target = info->curLoop;
7       rem = info->N[target]-info->itersScheduled[target];
8       switch ( info->method[target] ) {
9
10      case STATIC:
11           tChunk = ceil((double) info->N[target]/ (double) info->commSize);
12           info->batchSize[target] = tChunk;
13           info->batchRem[target] = min( info->batchSize[target], rem);
14           break;
15      case SS:
16           tChunk = 1;
17           info->batchSize[target] = tChunk;
18           info->batchRem[target] = min( info->batchSize[target], rem);
19           break;
20      case FSC:
21          tChunk = min( info->chunkFSC[target], rem);
22          info->batchSize[target] = tChunk;
23          info->batchRem[target] = min( info->batchSize[target], rem);
24          break;
25
26      case mFSC:
27          tChunk = min( info->chunkMFSC[target], rem);
28          info->batchSize[target] = tChunk;
29          info->batchRem[target] = min( info->batchSize[target], rem);
30          break;
31
32      case GSS:
33          tChunk = max( (rem+info->commSize-1)/info->commSize, info->minChunkSize[target] );
34          tChunk = min ( rem, tChunk );
35          info->batchSize[target] = tChunk;
36          info->batchRem[target] = min( info->batchSize[target], rem);
37          break;
38
39      case TSS:
40      tChunk = info->TSSchunk[target];
41          tChunk = min(rem, tChunk);
42          tChunk = max( info->minChunkSize[target], tChunk);
43          info->TSSchunk[target] = tChunk - info->TSSdelta[target];
44          info->batchSize[target] = tChunk;
45          info->batchRem[target] = min( info->batchSize[target], rem);
46          break;
47
48      case FAC:
49          if (info->batchRem[target] == 0) {
50              tChunk = max ( info->minChunkSize[target], (rem+2*info->commSize-1)
51                  /(2*info->commSize) );
52              info->batchSize[target] = info->commSize*tChunk;
53              info->batchRem[target] = min (info->batchSize[target], rem);
54          }
55          /* else use current batchSize */
56          tChunk = max( info->minChunkSize[target], info->batchSize[target]/info->commSize );
57          tChunk = min ( rem, tChunk );
58          break;
59
60      case WF:
61      case AWF:
62           if (info->batchRem[target] == 0) {
63              tChunk = max ( info->minChunkSize[target], (rem+2*info->commSize-1)/(2*info->commSize) );
64              info->batchSize[target] = info->commSize*tChunk;
```

```
65                  info->batchRem[target] = min (info->batchSize[target], rem);
66              }
67              /* else use current batchSize */
68              tChunk = max( info->minChunkSize[target], info->batchSize[target]/
69                  info->commSize*(info->weights[target*(info->numLoops)+rank]));
70              tChunk = min ( rem, tChunk );
71              break;
72
73          case AWF_B:
74          case AWF_D:
75              if (info->stats[target*(info->numLoops)+3*rank] < 0.0) {
76                  tChunk = info->minChunkSize[target];
77                  info->batchSize[target] = min(rem, tChunk);
78                  info->batchRem[target] = info->batchSize[target];
79              }
80              else { /* all ranks have wap */
81                  awap = 0.0;  /* average weighted performance */
82                  for (i=info->firstRank;i<=info->lastRank;i++)
83                      awap = awap + info->stats[target*(info->numLoops)+3*i];
84                  awap = awap/info->commSize;
85
86                  trw = 0.0;  /* total ref weight (refwt(i) = awap/info->stats[3*i] */
87                  for (i=info->firstRank;i<=info->lastRank;i++)
88                      trw = trw + awap/info->stats[target*(info->numLoops)+3*i];
89
90                  /* normalized weight for rank */
91                  weight = ((awap/info->stats[target*(info->numLoops)+3*rank])*info->commSize)/trw;
92
93                  if (info->batchRem[target] == 0) {
94                      tChunk = max( info->minChunkSize[target], (rem+2*info->commSize-1)/(2*info->commSize) );
95                      info->batchSize[target] = info->commSize*tChunk;
96                      info->batchRem[target] = min (info->batchSize[target], rem);
97                  }
98                  /* else use current batchSize */
99                  tChunk = weight*(info->batchSize[target]/info->commSize) + 0.55;
100                 tChunk = max( info->minChunkSize[target], tChunk);
101                 tChunk = min ( rem, tChunk );
102             }
103             break;
104
105         case AWF_C:
106         case AWF_E:
107             if (info->stats[target*(info->numLoops)+3*rank] < 0.0)
108                 tChunk = info->minChunkSize[target];
109             else { /* all ranks have wap */
110                 awap = 0.0;  /* average weighted performance */
111                 for (i=info->firstRank;i<=info->lastRank;i++)
112                     awap = awap + info->stats[target*(info->numLoops)+3*i];
113                 awap = awap/info->commSize;
114
115                 trw = 0.0;  /* total ref weight (refwt(i) = awap/info->stats[3*i) */
116                 for (i=info->firstRank;i<=info->lastRank;i++)
117                     trw = trw + awap/info->stats[target*(info->numLoops)+3*i];
118
119                 /* normalized weight for rank */
120                 weight = ((awap/info->stats[target*(info->numLoops)+3*rank])*info->commSize)/trw;
121                 tChunk = weight*((rem+2*info->commSize-1)/(2*info->commSize)) + 0.55;
122             }
123             tChunk = max( info->minChunkSize[target], tChunk);
124             info->batchSize[target] = tChunk;
125             info->batchRem[target] = min(rem, tChunk);
126             break;
127
128         case AF:
129             if (info->stats[target*(info->numLoops)+3*rank] < 0.0)
130                 tChunk = info->minChunkSize[target];
131             else {
132                 bigD = 0.0;
133                 bigT = 0.0;
134                 for (i=info->firstRank;i<=info->lastRank;i++) {
135                     bigD = bigD + info->stats[target*(info->numLoops)+3*i+1]/
136                         info->stats[target*(info->numLoops)+3*i];
137                     bigT = bigT + 1.0/info->stats[target*(info->numLoops)+3*i];
```

```
138                 }
139                 bigT = 1.0/bigT;
140                 /* compute chunk size for rank */
141                 tChunk = 0.55 + (0.5*(bigD + 2.0*bigT*rem -
142                     sqrt(bigD*(bigD + 4.0*bigT*rem)))/info->stats[target*(info->numLoops)+3*rank]);
143                 tChunk = min( info->maxChunkSize[target], tChunk);
144             }
145             tChunk = max( info->minChunkSize[target], tChunk);
146             info->batchSize[target] = tChunk;
147             info->batchRem[target] = min( info->batchSize[target], rem);
148             break;
149 
150         default:
151             printf("Unsupported DLS technique, fall back to STATIC\n");
152             tChunk = (info->N[target]+info->commSize-1)/info->commSize;
153             i = info->N[target] % info->commSize;
154             if ((i>0) && (rank>=i) ) tChunk=tChunk-1;
155             tChunk = min( tChunk, rem);
156             info->batchSize[target] = tChunk;
157             info->batchRem[target] = min( info->batchSize[target], rem);
158 
159         }
160 
161         *chunkSize = min(info->batchRem[target], tChunk);
162 
163         /* adjust remaining in batch */
164         info->batchRem[target] = info->batchRem[target] - *chunkSize;
165         if ( (info->batchRem[target] > 0) && (info->batchRem[target] <= info->minChunkSize[target]) ) {
166             *chunkSize = *chunkSize + info->batchRem[target];
167             info->batchRem[target]= 0;
168         }
169 }
```

Listing B.10: GetChunkSize function, calculates the chunk sizes to assign according to the chosen scheduling technique for a certain loop.

```
1  void  SetBreaks ( infoDLS *info )
2  {
3      //assumption: curLoop has been set accordingly
4      if (info->myRank == info->foreman) {
5          /* when to check for messages */
6          if (info->breakAfter<0)
7              info->probeFreq[info->curLoop] = max( 1, (info->wSize[info->curLoop]+
8                  info->commSize-1)/info->commSize/4 );
9          else
10             info->probeFreq[info->curLoop] = max( 1, info->breakAfter);
11 
12         /* how many iterates left before requesting next chunk */
13         if (info->requestWhen<0)
14             info->sendRequest[info->curLoop] = info->probeFreq[info->curLoop];
15         else
16             info->sendRequest[info->curLoop] = info->requestWhen;
17     }
18     else { /* not the foreman */
19 
20         /* how many iterates left before requesting next chunk */
21         if (info->requestWhen<0)
22             info->sendRequest[info->curLoop] = max( 1, (15*info->wSize[info->curLoop])/100 );
23         else
24             info->sendRequest[info->curLoop] = info->requestWhen;
25 
26         /* when to check for messages */
27         info->probeFreq[info->curLoop] = max( 1, info->wSize[info->curLoop]-
28             info->sendRequest[info->curLoop]);
29 
30     }
31 }
```

Listing B.11: SetBreaks function, allows to "interrupt" executions of chunks to check for messages or send work requests.