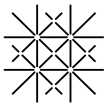


University of Basel

Dynamic Scheduling in HPC using a Distributed Data Approach

Master Thesis

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

Advisor: Prof. Dr. Florina M. Ciorba
Supervisor:
Jonas Henrique Müller Korndörfer

Gian-Andrea Wetten
gian-andrea.wetten@stud.unibas.ch
12-720-777

15^{th} of May 2022

Acknowledgments

I want to thank Prof. Dr. Florina M. Ciorba for motivating me to explore this field of study. Furthermore I also want to thank Jonas H. M. Korndorfer for supervising me. He was always available when I needed help and assisted me in a very kind manner during the period of my master's thesis. Additional thanks for consultation on the implementation of my work goes to Dr. Ahmed Hamdy Mohamed Eleliemy

Abstract

Scheduling has been an efficient way of dealing with load imbalance and therefore increasing the performance of parallel applications. However, the size of size of current parallel applications and systems introduces significant scalability issues. One way of addressing this problem is scheduling with distributed data. A lot of research has gone into this topic and work-stealing scheduling algorithms in particular have become increasingly popular over the years. In this work we investigate how scheduling with distributed data can increase the scalability and thus help mitigate these concerns, potentially leading to an overall increase in the performance of parallel applications.

Table of Contents

1 Introduction

It is a well known fact that nowadays, contrary to Moore's law[17], increasing the performance of modern computers is achieved by adding to the number of processors rather than increasing the number of transistors. This is due to the fact that we have hit the power wall resulting from the breakdown of Dennard scaling.
As a consequence the relevance of parallel programming skyrocketed and with it the interest in creating efficient parallel paradigms. A very important factor in reducing the overall execution time of parallel programs is to keep the idle time on processing elements (PEs) to a minimum. For this reason a lot of effort has been put in researching scheduling algorithms over the last three decades. Scheduling is a very powerful tool in the parallel programming world when dealing with irregular loops. It leads to a more even workload distribution and therefore to a reduction in load imbalance. It however also comes at a cost since it incurs an overhead. This trade-off between performance gain and optimal workload distribution is a crucial point to keep in mind.
The ever increasing demand on scalability in the high performance computing (HPC) world over the years means that we encounter problems which can not be covered using a centralized data approach. This led to searching for a scheduling solution which could deal with distributed data. A popular approach of dealing with this issue have been the work-stealing algorithms.
The goal of this master thesis is to implement several approaches for scheduling with distributed data in the LB4OMP[19] as well as the LB4MPI[1] library. Each library will be extended with 2 core scheduling algorithms which can be tuned further with environment variables. We will analyze the performance implications of using our solutions compared to existing solutions which work with replicated data and thus gain an insight on the scalability of our approach. In addition to that we will also compare our solutions between each other. This will be done using 6 different applications on our miniHPC cluster here at the University of Basel.

The remainder of this thesis is structured as follows. Chapter 2 explains the background knowledge required to understand the topics of our thesis. In chapter 3 we present what has already been achieved in our field and explain how our solution will differ from previous work. Chapter 4 and 5 explain how LB4OMP works and our methodology for extending the library with our own techniques. The next two chapters have the same structure but with a focus on LB4MPI. After that we look at the experimental setup in chapter 8 and follow up with a discussion of the results in chapter 9. The last chapter provides our drawn conclusions and opportunities for future work.

2
Background

2.1 APIs

When writing parallel programs we often make use of application programming interfaces (API). They help keeping our parallelized sections consistent and also facilitate the writing of parallel code in general. The two APIs used most commonly nowadays are OpenMP and MPI.

2.1.1 OpenMP

The OpenMP[20] API is widely considered the standard method of parallelism in shared-memory environments. It is based on the concept of multithreading, which means that we spawn additional sub-threads in the main thread and assign preferably evenly divided work to them. There are implementations in C,C++ and Fortran and it is supported by the most popular compilers, such as GCC, LLVM and the Intel compilers. The first version was introduced in 1987. Since then several new versions introducing new features have been released, such as the support for task parallelism in version 3.0.

OpenMP is comprised of compiler directives, library routines and environment variables. The core concepts of multithreading are controlled with pragmas. They allow for the creation of threads, work-sharing constructs and for controlling thread synchronization. User-level run-time routines are provided to check the number of threads, create timers and more. The environment variables are used to set the number of threads, the scheduling methods as well as other runtime parameters.

2.1.2 MPI

The Message-Passing Interface (MPI)[11] is a specification for interfaces of message-passing libraries. The goal is to provide an efficient and portable standard for message-passing programs. It is the most commonly used standard for parallel programming on a distributed-memory environment but can also be used on shared-memory environment.

Each process has its own call stack and address space. The coordination between them has to happen explicitly by sending and receiving messages. The standard defines point-to-point communication as well as collective communication operations. Later versions introduced

advanced concepts like remote-memory access operations and parallel I/O.
The implementations of MPI (Open-MPI, MPICH,...) provide functions which are directly callable from C,C++ and Fortran. In addition to that there are a number of language bindings for other programming languages.

2.2 Scheduling

Scheduling is crucial to control the amount of load balance introduced in our systems by the parallelization of an application. It helps reducing idle time on processes or threads and therefore also reduces the execution time of our programs.

2.2.1 Static scheduling

During static scheduling we divide the workload into evenly sized chunks. The chunks of work are then assigned to PEs statically before the parallel section is executed. This means that we have a very small scheduling overhead. However, often the loops that we want to parallellize are irregular loops. This means that not each iteration takes the same amount of time. Hence we end up with PEs that are finished before others, therefore leading to a load imbalance and idle time on those PEs. As a consequence we usually want to avoid using static scheduling when dealing with irregular loops.

2.2.2 Dynamic loop self-scheduling

Dynamic scheduling techniques aim to improve application performance by reducing load imbalance. Instead of dividing the workload a priori we assign work during the execution of the parallel section. When a PE is finished it will be assigned the next chunk of work. The chunk size can vary during the execution. A small chunk size means less load imbalance but a higher scheduling overhead due to having to assign work more often. Many algorithms have been implemented to balance the trade-off of load imbalance and scheduling overhead. An example would be the guided self-scheduling (GSS) algorithm which decreases the chunk size based on the number of remaining loop iterations.
Dynamic scheduling techniques can significantly improve the execution time of applications but are not always the correct choice. When dealing with highly regular loops we want to avoid using them because of the scheduling overhead induced by them.

2.2.3 Work-sharing

In work-sharing policies overloaded PEs push tasks from their local task queue to idle PEs. This can happen in a centralized approach using a global centralized task queue. Busy PEs push tasks to the global queue and idle PEs pull tasks from that same queue. As a consequence we introduce a synchronization overhead for access to the queue. Work-sharing can also be implemented in a decentralized manner. Either the busy PEs have to probe for idle ones which incurs a performance cost or they push to a random PE which leads to an increased total number of task migrations.

2.2.4 Work-stealing

In contrast to work-sharing policies the idle PEs are responsible for finding and stealing tasks from busy PEs. The PE stealing the task(s) is denoted as the thief in this policy while the target PE is denoted as the victim. As a consequence we have a significantly lower overhead on the busy PEs. No messages are exchanged if all PEs are busy, making this a stable kind of scheduling [9]. As mentioned by Eager et al. [10] we have a lower total task migration cost at heavy load than with work-sharing. However it might still be worse overall since we need to account for the cost of transferring executing tasks, whereas we only consider newly created tasks with work-sharing policies. Victim selection can be made randomly or with probing. Probing incurs a significant overhead and is often not worth it but it has the advantage of potentially reducing the total number of task migrations.

2.2.5 Affinity Scheduling

Affinity scheduling was first introduced in 1994 by Markatos and LeBlanc[16]. In addition to the aspect of load imbalance and scheduling overhead it also takes performance degradation due to access to non-local data into account. This non-uniform memory access (NUMA) cost plays a very important role on modern shared-memory multiprocessors. Therefore affinity scheduling is very suitable for NUMA machines that take data locality into account. The disadvantage is a high overhead incurred by task migrations when using a lot of processors.

2.2.6 LB4OMP

LB4OMP[19] is an extension of the OpenMP API. The motivation behind this extension was to address a lack of scheduling options in the OpenMP standard. In addition to the three techniques implemented in the OpenMP standard it introduces 14 new dynamic scheduling techniques to the system as well as 5 automated scheduling algorithm selection techniques. These automated techniques use run-time metrics to determine the best possible scheduling algorithm across time-steps and loop iterations.

2.2.7 LB4MPI

LIB4MPI[1] is a library that implements several dynamic scheduling techniques for MPI. It is an extension of the DLS4LB tool which in turn is based on the DLB_tool[6].

2.3 Divide & Conquer

Divide & Conquer (D&C) parallel programs work by partitioning the main problem in sub-problems until we can trivially solve the sub-problem. Afterwards all results are merged to attain the final result. The execution time of D&C programs is heavily dependant on if there are enough processors to execute all task of a certain tree depth. Another characteristic of these programs is that the degree of parallelism in the parallellization of them can be controlled. This is due to the fact that the recursive nature of the subdivision allows for the sub-problems to be performed sequentially or by a separate PE.

2.4 Non-uniform memory access (NUMA)

In a NUMA architecture each process is provided separate (local) memory. This has the advantage that, depending on circumstances, multiple processors can access the computer's memory at a time, which is a big missing feature on multi-processor systems without NUMA. Therefore NUMA helps avoiding processor starvation. An important thing keep in mind with NUMA is that remote memory access is a lot slower than local memory access. This means, for example, that we have to think about where we initialize the data in our multithreaded applications if we want to optimize performance.

2.5 First touch policy

The first touch policy dictates that a data page is allocated in the memory closest to the thread accessing this page for the first time. This can be a problem in multithreaded applications if we initialize the data on the first thread. If all the data resides in the memory of a single NUMA node we end up with congestion at the memory controller and thus processor starvation. One solution for this is to initialize the data in a parallel environment.

3
Related Work

A lot of research has been conducted on scheduling with a work-stealing concept. The topic has first been proposed by Burton and Sleep[4] in 1981. They introduced the idea of partitioning the processes into the three states *pending*, *active* and *blocked*. Processors maintain a list of pending processors. That information is also periodically shared with neighbouring processors. Every time a processor becomes available it marks a pending process as active and then chooses an active process to run. In case of no pending processes a processor can ask one of its neighbours to transfer one of theirs to its active set of processes, effectively stealing a pending task from them. In addition to that they reinforced a "single steal" rule which dictates that a process may be stolen at most once. This ensures locality of parent-child communication.

The work by Burton and Sleep was the basis for the "first provably good work-stealing scheduler for multithreaded computations" presented by Blumofe and Leiferson[2] in 1999. They showed that their work-stealing approach delivers better upper bounds both in terms of space and communication requirements than work-sharing schedulers. In their work they describe a randomized work-stealing algorithm where each processor maintains a ready double ended queue (deque) of threads. An idle processor removes the bottom of its deque and starts working on that thread. Should the deque be empty it steals the topmost thread from a randomly chosen processor. Newly spawned threads are added to the bottom of the ready queue.

Another interesting work-stealing approach has been brought up by Pezzi et al.[21] in 2007 with hierarchical work-stealing. It is an on-line scheduling algorithm for D&Q MPI programs. It uses a "tree-like hierarchy of manager processes" to route stealing requests. The leaf nodes represent workers in their system while the inner nodes act as managers. The managers handle the work-stealing request and their routing. This approach is taken to resolve the problem of MPI processes needing a shared communicator for one-to-one communication.

Wang et al.[23] came up with the idea of using the work-stealing and the work-sharing approach in conjunction as part of their hierarchical task scheduling scheme (AHS). A global scheduler (GS) is used to make the initial partitioning and maintain task counter for each worker node. Local schedulers (LS) maintain task queues for each worker threads on

a node. This intra-node scheduling is balanced using work-stealing. An LS can also send inter-node work-stealing requests to the GS which then in turn uses the task counter to determine the node with the heaviest load as the victim node. If the task counter is higher than a certain threshold then it informs the victim node to initiate a task transfer to the thief. Furthermore the inter-node work-sharing is implemented in a similar way but instead the node with the lowest task count is chosen as the victim node, given a task count lower than the set threshold. In our work we are going to use this concept of combining a LS with a GS for implementing a loop scheduling algorithm which can deal with work-stealing requests in a more informed manner.

A problem with traditional work-stealing models with modern multisocket CPU architectures is that we often cannot access the data from the fast local memory. Furthermore the shared cache utilization is often sub-par. For these reasons Chen and Guo[7] decided to implement a locality-aware work-stealing (LAWS) algorithm for task scheduling in 2015. LAWS is a Cilk extension and its main feature is the even dataset distribution across the memory nodes and the allocation of tasks to the corresponding socket. They reached an impressive performance increase of up to 54.2%, compared with Cilk, on their AMD-based platforms for heavily memory-bound D&C applications. This was achieved using a three step process. At first tasks are allocated to sockets using the load-balanced task allocator. Afterwards it improves cache efficiency by packing the tree-shaped task execution graph of the D&C programs into cache friendly(CF) subtrees. In the last step they account for load unbalances in the task allocator by using a triple-level work-stealing scheduler where a socket steals a CF subtree from another randomly chosen socket if it is done with all their subtrees. For CPU-bounded application they implemented a fallback to traditional work-stealing based on the measured cache miss intensity of the tasks. Although the overhead of their algorithm is quite small, there is no benefit of using the LAWS scheduler with CPU-bounded applications. Our work is going to use the idea of socket-local queues for a locality-aware work-stealing that applies to loop scheduling.

In the same year an article was published by Muddukrishna et al.[18] proposing locality-aware taks scheduling for OpenMP. They bind a task queue to each NUMA node and allow work-stealing from other queues if there is no work in the local queue. The victim queue is chosen based on on NUMA node distances which can be obtained from operating system tables. Furthermore stealing form nearly empty queues is not possible to limit the number of multiple consecutive steals from the same victim. We use a similar approach in our implementation but focus on loop scheduling rather than task scheduling.

LB4OMP[19] was introduced to address a lack of scheduling options in the OpenMP standard. Korndörfer et al. implemented 14 dynamic scheduling algorithms in their work and performed an analysis of the performance gain possible with each option. They showed that the presented techniques outperform the ones from the OpenMP standard on multiple application-systems pairs. This work is using their research infrastructure LB4OMP to perform thread-level scheduling.
LB4MPI [1] is used for process-level scheduling. The library was introduced by Ciorba and Eleliemy and adds multiple dynamic scheduling algorithms for MPI. This work aims to extend that library with work stealing algorithms for scheduling with distributed data.

4

Loop Scheduling in LB4OMP

Loops in OpenMP can be parallelized by adding a simple compiler directive like *pragma omp parallel for*. The application developers have a plethora of additional parameters at their disposal to tune this parallelization procedure to their preferences, such as the control over shared and private variables or how to handle nested loops. The *schedule* clause can be used to choose which strategy should be used to perform the scheduling of the loop iterations. Alternatively we can also decide to choose the strategy at the runtime of the application if we set this parameter to *schedule(runtime)* and provide a valid option to the library via an environment variable called *OMP_SCHEDULE*.

4.1 Extending LB4OMP

Extending LB4OMP with an additional technique comprises modifications in a few source code files of the library. First of all we want our algorithm to be recognized as a valid parameter for the scheduling directives. For this we have to add the algorithm identifier to the *kmp_sched* and *sched_type* enums in *kmp.h*. Furthermore we also have to complete the mapping procedure in the *__kmp_get_schedule* function of *kmp_runtime.cpp*, as well as ensure the correct parsing (*__kmp_parse_single_omp_schedule*) and printing (*__kmp_stg_print_omp_schedule*) of the scheduling technique in *kmp_settings.cpp*.
Auxiliary environment variables can be added in *kmp_global.h*, should they be needed. In that case we again have to provide a method for parsing and printing them in *kmp_settings.cpp* and declare them as an *extern* variable in *kmp.h* for global access purposes.
The main scheduling logic happens in *kmp_dispatch.cpp*. This file contains two very important functions *__kmp_dispatch_init_algorithm* and *__kmp_dispatch_next_algorithm*. Both of those functions have to be extended with a case branch for the next algorithm that we want to use. The first function, *__kmp_dispatch_init_algorithm*, is called once on all threads at the beginning and is used to set up the variables used by the scheduling technique. The library provides a templates for structs containing thread-private (*dispatch_private_info_template*) and shared (*dispatch_shared_info_template*) variables. These templates can be extended if there is a need for more parameters. The actual calculation for the chunk sizes happens in *__kmp_dispatch_next_algorithm* and is called until there

are no more iterations left to perform. Proper locking of shared variables has to be ensured to avoid common pitfalls like deadlocks or race conditions. For some configurations there is an additional cleanup step in *__kmp_dispatch_finish* or *__kmp_dispatch_finish_chunk* where some counters are reset.

5
Implementation in LB4OMP

5.1 RWS

Random work stealing (RWS) can be chosen with the parameters *rws_static,[chunkparam]* as a value for the *OMP_SCHEDULE* environment variable.
RWS is implemented by partitioning the iterations evenly to each thread at the initialization step. This is done by keeping track of the number of iterations using a lower and upper bound for each thread. Since we are storing all of the information in thread-private variables and only need locking when we attempt to steal chunks from other threads we operate with a **thread-local** queue. The actual stealing happens in the *__kmp_dispatch_next_algorithm* method. We first compare our own lower and upper bound and if there are no more iterations left we proceed with the stealing procedure.
The victim is chosen randomly based on random number generator function *rnd(x)* which takes a set of queues as input and produces the output queue using an integer calculated according to a weighted discrete distribution

$$P(i|w_0, w_1, ..., w_{N-1}) = \frac{w_i}{\sum_{k=0}^{N-1} w_k} \tag{5.1}$$

The previous equation has N, denoting the number of threads, and weights $w_i, w_k \in 0, 1$ as variables. On each thread we initially set the weight for their own queue to 0 and the weight of all other queues to 1. A probing is performed to ensure that we do not attempt to steal from a thread that has no iterations left. If the probing returns an empty queue we set the weight for that queue to 0 and choose another victim.
After acquiring a lock on the victim thread, the amount of iterations to be stolen needs to be determined. The ratio of iterations to steal can be configured via an environment variable called ***KMP_STEAL_RATIO***. The variable takes an integer number which statically denotes the percentage of iterations left on the victim thread we want to steal. By default we set a steal ratio of 25%. Furthermore there is a possibility of using the algorithm for the *fac2a* technique to determine the amount we want to steal dynamically. This can be achieved by choosing *rws_fac2a,[chunkparam]* as the schedule.

5.2 LAWS

Locality aware work stealing (LAWS) can be chosen with the parameters *laws_static,[chunkparam]* respectively *laws_fac2a,[chunkparam]* either as an option to the schedule directive or as a value for the *OMP_SCHEDULE* environment variable.
The implementation of LAWS follows RWS pretty closely with the exception of keeping track of the NUMA node id for each thread. In the initialization step each thread determines the NUMA node it belongs to and stores the corresponding id in a shared map which has NUMA node ids as keys and thread ids as values. For the victim search we first try to choose a random thread with the same NUMA node id. If all threads on the same NUMA node have no iterations left we use a fallback option of randomly choosing a thread on a different NUMA node as the victim thread.
The calculation for the amount of iterations to steal has the same options and makes use of the same environment variables available as RWS.

5.3 Comparison

Table 5.1 shows a comparison between our newly implemented scheduling techniques in LB4OMP-WS and the existing technique *static_steal* which was already implemented in LB4OMP. The main difference between each algorithm is the victim selection, as well as the number of iterations that are being stolen.

Table 5.1: Comparison of Work-Stealing Techniques in LB4OMP-WS

	static_steal	**rws_fixed**	**rws_fac2a**	**laws_fixed**	**laws_fac2a**
Nr. of queues	1 per thread				
Victim selection	$v^i = (v^{i-1}+1)\%N$	$v^i = rnd(Q)$		$v^i = rnd(Q^*)$	
Nr. of iterations to steal	25%	fixed	fac2a	fixed	fac2a
Fallback victim selection	$v^i = (v^{i-1}+1)\%N$	$v^i = rnd(Q \setminus \{v^{i-1}\})$		$v^i = \begin{cases} rnd(Q^* \setminus \{v^{i-1}\}) & \lvert Q^*\rvert > 0 \\ rnd(Q \setminus \{v^{i-1}\}) & else \end{cases}$	

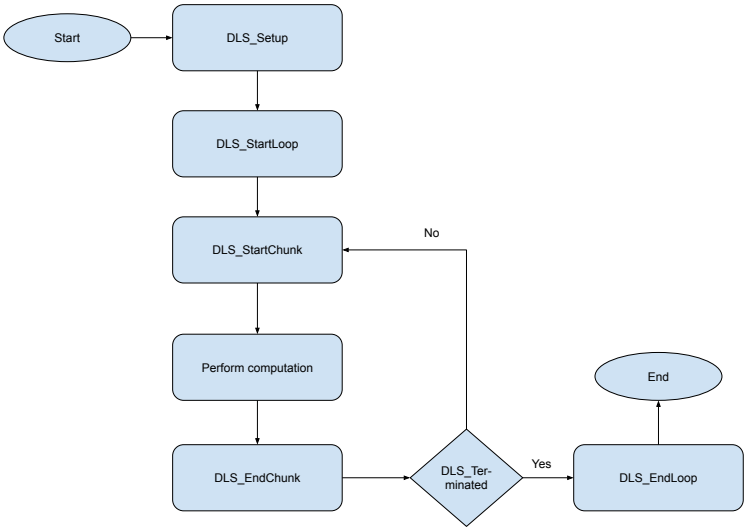
- N: Number of threads
- v^i: Thread id of the victim i-th victim thread
- Q: Set of all thread ids
- Q^*: Set of thread ids from threads on the same NUMA node
- Probabilities for $rnd(x)$: $P(i|w_0, w_1, ..., w_{N-1}) = \frac{w_i}{\sum_{k=0}^{N-1} w_k}$ with $0 \leq i < N$ and $w_i, w_k \in \{0, 1\}$

6

Loop Scheduling in LB4MPI

The scheduling of loops in MPI applications can be done using the LB4MPI library. There are three modifications to the application code that need to be implemented for this. First of all we need to setup all the necessary parameters using the *DLS_Parameters_Setup* method. With this procedure we communicate to LB4MPI what kind of resources and how many ranks we are using. This information is crucial for the library as some of the scheduling techniques rely on performance metrics. In a second step we need to call *DLS_StartLoop* and *DLS_EndLoop* before and after the loop we want to schedule. These functions are executed exactly once in each rank and are responsible for the initialization and cleanup of the scheduling techniques. In *DLS_StartLoop* we specify which scheduling technique we want to use in addition to the total number of iterations that need to be performed. This is where the queue of iterations is being created and where the first chunk of work is distributed. The last modification pertains to the actual scheduling of the loop (i.e. chunk calculation) and distribution of work. The loop to be scheduled is encased in a while loop which checks if there are any iterations left to perform or if the execution of the loop is finished. This loop runs until the check from the *DLS_Terminated* method returns true. Inside that loop we calculate the starting iteration and chunksize using *DLS_StartChunk* and use those variables to execute our loop for the respective amount of iterations. We can do this by using a function which calls our loop and takes start and end iteration as an argument. After the execution of the loop iterations we call *DLS_EndChunk* to update performance and status metrics for our scheduling algorithms. Figure 6.1 shows the workflow of the LB4MPI library and in 6.1 we can see an example code.

Figure 6.1: LB4MPI workflow



Listing 6.1: Example Code LB4MPI

```
1  int itersDone=0, requestWhen = 0, breakAfter = -1, minChunk = 1, nKNL =0;
2  int start=0,end=0, chunk_size=0, nprocs=nProcesses;
3  double workTime=0.0, Xeon_speed =0.0, KNL_speed =0.0, h_overhead=0.0, sigma=0.0;
4  DLS_Parameters_Setup(MPI_COMM_WORLD, &iInfo, nprocs, requestWhen, breakAfter, minChunk,
       h_overhead, sigma, nKNL, Xeon_speed, KNL_speed);
5  DLS_StartLoop ( &iInfo, 0,len, 0, exponentialLoop);
6    while(!DLS_Terminated ( &iInfo )){
7        DLS_StartChunk ( &iInfo, &start, &chunk_size );
8        end = start + chunk_size;
9        if(start<end)
10         {
11           exponentialLB(exponentialLoop, len, nProcesses, myrankid, start, end);
12         }
13       DLS_EndChunk ( &iInfo );
14   }
15 DLS_EndLoop( &iInfo, &itersDone, &workTime );
```

7 Implementation in LB4MPI

7.1 Limitations of LB4MPI

The current version of LB4MPI can only be integrated into applications that work with replicated data. It uses a work-sharing approach where a coordinator distributes work to MPI ranks from a centralized queue. The amount of work that is being shared is based on the scheduling algorithm chosen by the user. While this is a very reasonable approach, it is not generic enough for a lot of MPI applications that are being developed and used nowadays. Developers of scientific applications increasingly use a distributed data approach due to scalability reasons.

7.2 LB4MPI-WS

The limitations mentioned in the previous section mean that LB4MPI can not be integrated in a lot of modern scientific applications that use MPI. Our goal was to provide a solution for this problem by adding the functionality of dealing with distributed data to LB4MPI. In the following two sections we introduce LB4MPI-WS, an LB4MPI extension that uses a work-stealing algorithm to add this feature to LB4MPI.

7.2.1 Random Work-Stealing (RWS) in LB4MPI-WS

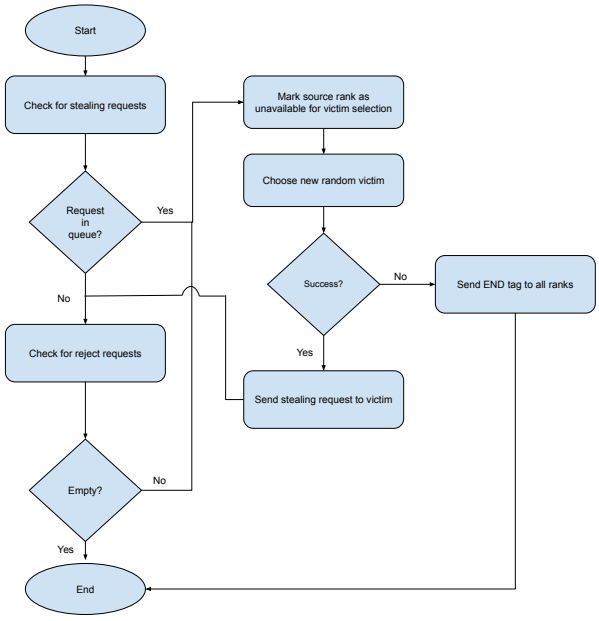
In RWS one of the ranks acts as a coordinator while all the other ranks are referred to as workers. When a rank notices that it has no more iterations left to work on it sends a stealing request to the coordinator. The coordinator keeps track of potential victim ranks. A rank can be a victim if both of these conditions apply:

1. It has not been a thief before.
2. It is not the coordinator.

The first conditions mitigates ping-pong style stealing while the second condition makes sure that we do not end up in a deadlock situation on the coordinator. The situation could occur if we send out a stealing request from the coordinator while another rank is waiting for a response on their own stealing request.

If there are possible candidates left the coordinator chooses a victim randomly amongst them and relays the stealing request to that victim. Otherwise we can assume that no more stealing can be performed for this loop and therefore the coordinator sends out an *END* tag to all ranks, as depicted on the coordinator workflow in Figure 7.2. Upon receiving a stealing request a worker either sends a certain amount of iterations and the corresponding data to the original requester or, in case it has no iterations left, a rejection to the coordinator. The stealing procedure from a worker's point of view can be observed in Figure 7.1. The coordinator starts the whole process of victim selection anew should he receive such a reject message from a worker.

Figure 7.1: Work-Stealing procedure from the coordinator

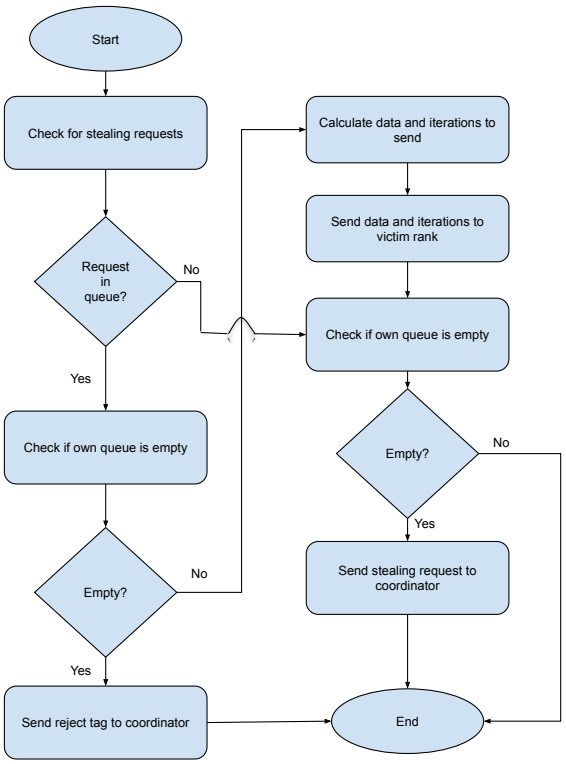


7.2.2 Describing Data in LB4MPI-WS

Adding work stealing scheduling to an application needs a few modifications on the application side. A very important part of said modifications is describing the data that is being used in our modified loop. The goal is to communicate to the library which data needs to be sent from the victim to the thief and how it needs to be packed on the sender side and unpacked on the receiver side. The way that this works in our implementation is that you need call an additional setup function from the library on the application side, called *DLS_DataSetup*. This function takes a void pointer to the data structure that holds the data which needs to be stolen. Additionally it also requires the MPI Datatype of our data elements. As a last parameter it takes a pointer to a mapping function which describes the relation between an iteration of our loop and the data. Said mapping function is called by the victim rank before sending the data to the thief and needs to take the following parameters:

- The first iteration of the chunk that needs to be stolen

Figure 7.2: Work-Stealing procedure from a worker



- The last iteration of the chunk that needs to be stolen
- A void pointer to the data
- The MPI datatype of one data element
- A pointer to a derived MPI datatype which is where the mapping function stores its output

The user of the library has to consider which MPI Datatype is appropriate for the mapping of his data to iterations. In the following section we introduce the main three categories of describing data through this function with a varying degree of complexity. The descriptions are accompanied by listings which demonstrate a concrete use-case for each category.

7.2.3 One to One Mapping

The most straight forward case is when we have a simple one-dimensional array of primitive types with a 1:1 mapping of data elements to iteration. This means that we use exactly one data element for each iteration and there is no displacement or stride factor. An example of such a loop and the corresponding *DLS_DataSetup* function call can be seen in 7.1. In this case we don't need to pass a mapping function at all because the library assumes 1:1 mapping by default. Therefore we can pass a nullpointer as an argument for the mapping function in our call to the *DLS_DataSetup* function.

Listing 7.1: Loop with 1:1 relation between data and iteration

```
int *myData,*result;
for (int i = start; i<end; i++) { // start and end is determined by LB4MPI
    result[i] = 5 * myData[i];
}

DLS_DataSetup(&iInfo, myData, MPI_INT, NULL);
```

7.2.4 N to One Mapping with Primitive Datatypes

If the data we are using has no 1:1 mapping we can create an instance of a MPI derived datatype. The listing 7.2 shows a scenario where our data is comprised of a three-dimensional matrix of the type double. For each iteration of the the stolen chunk we need to send all data elements of the matrix where the coordinates are described as

$$\{(x, y, z) \mid i - 1 <= x <= i, 0 <= y < |Y| - 1, 1 <= z < |Z|\} \tag{7.1}$$

where i corresponds to the iteration counter. Figure 7.3 shows a visualization of an example for the loop in our listing. Marked in red are the blocks which need to be send to the thief if he were to steal the second iteration of our scheduled loop. The MPI_Type_create_subarray function can be used to construct a n-dimensional subarray of an n-dimensional array. It takes the dimension sizes of the original array, the dimension sizes of the subarray and the starting coordinates of the subarray in each dimension as parameters. With this information we create an instance of a derived datatype that describes the mapping of our data to (chunks of) iterations. The advantage of this is that the receiver side can use this datatype to unpack the stolen data correctly without the need of having to explicitly define the serialization and deserialization procedure.

Listing 7.2: Loop with a N:1 relation between data and iteration

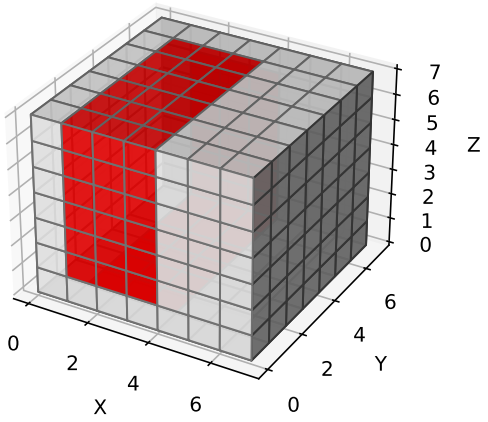
```
for (i = start; i < x_block_size; i++) // start and end is determined by LB4MPI
    for (j = 1; j <= y_block_size; j++)
      for (k = 1; k <= z_block_size; k++)
          result[i][j][k] = 5 * (myData[i-1][j  ][k  ] +
                                 myData[i  ][j-1][k] +
                                 myData[i+1][j][k+1]);

void dMap(void* in, int start, int end, MPI_Datatype* out, int* len) {
  MPI_Datatype arrayslice;
  int arraysliceSize;

  const int mSizes[] = {x_block_size + 2,y_block_size + 2,z_block_size + 2}; // Input
       array sizes
  const int mSubSizes[] = {end-start + 1,y_block_size + 1,z_block_size + 1}; // Subarray
       sizes
  const int mStarts[] = {start-1,0,0}; // Start indices for subarray
  MPI_Type_create_subarray(3,mSizes,mSubSizes,mStarts,MPI_ORDER_C,MPI_DOUBLE,&arrayslice);
  MPI_Type_commit(&arrayslice);
  MPI_Type_size(arrayslice,&arraysliceSize);
  *out = arrayslice;
  *len = arraysliceSize;
}

void (*dMap_ptr)(void*, int, int, MPI_Datatype*, void*, int*) = &dMap;
DLS_DataSetup (&iInfo, myData, MPI_DOUBLE, NULL, dMap_ptr);
```

Figure 7.3: N to one example for a matrix



7.2.5 Explicit (De-)Serialization

The third example demonstrates the situation where our data is of a complex datatype like a C++ class. In this case it is necessary for us to serialize and deserialize the data explicitly. The listing 7.3 below depicts objects of the class Node that are being serialized to a byte stream on the victim rank. The thief rank takes this stream of bytes and deserializes it to objects of the class again. For this we need to provide an extra function to our library which calls the deserialization methods on the objects. The serialization procedure can be either implemented manually or by the means of a serialization library like *Boost::Serialization*[3]. This method can also be used if each iteration uses multiple data structures for the computation of our result.

Listing 7.3: Loop which uses complex datatypes for our data

```
1  std::vector<Node> nodeList;
2  for (int i = start; i<end; i++) { // start and end is determined by LB4MPI
3      result[i] = findNeighbors(nodeList[i]);
4  }
5
6  std::string serializeNode(const Node &n) {
7    std::stringstream ss;
8    boost::archive::binary_oarchive oa{ss};
9
10   oa << n;
11
12   return ss.str();
13 }
14
15 Node deserializeNode(const std::string &in) {
16   std::stringstream ss(in);
17   boost::archive::binary_iarchive ia(ss);
18
19   Node obj;
20   ia >> obj;
21
22   return obj;
23 }
24
25 void dMap(void* in, int start, int end, MPI_Datatype* out_type, void* out, int* len) {
26   Node *t = static_cast<Node *>(in);
```

```
27   std::string ts= serializeNode(*t);
28   *out_type = MPI_BYTE;
29   *out = ts.c_str();
30   *len = ts.size();
31 }
32
33 void (*fun_pointer)(void*, int, int, MPI_Datatype*, void*, int*) = &dMap;
34 DLS_DataSetup ( &iInfo, static_cast<void*>(nodeList.data()), MPI_BYTE, fun_pointer);
```

8 Experiments

8.1 System

All experiments have been performed on the miniHPC[8] cluster of the University of Basel. This cluster consists of 22 Xeon nodes, 4 KNL nodes and 1 Cascade Lake node. The specifications of each node type are displayed in table 8.1

Table 8.1: miniHPC system specifications

	miniHPC-Xeon	**miniHPC-KNL**	**miniHPC-Cascade Lake**
Processor Name	Intel Broadwell E5-2640 v4	Intel(R) Xeon Phi(TM) CPU 7210	Intel Xeon Gold 6258R
Nodes	22	4	1
Sockets	2	1	2
CPU speed (GHz)	2.4	1.3	2.7
Cores (per CPU)	10	64	28
Threads (per CPU)	20	256	56
Cache (MB)	L2: 25	L3: 32	L3: 38.5

LB4OMP-WS, Mandelbrot[15], SPHYNX[5] and miniVite[12] have been compiled with the Intel Compiler[14] 19.0. For LB4MPI-WS, SPH-EXA[13] and Dist we have used a more recent release, namely Intel Compiler 2021.4.0. For our resource configuration system we used Slurm version 21.08.5[25].

8.2 Design of Experiments

Table 8.2 shows the factorial design of experiments in detail. The applications were chosen based because they have been shown to be good candidates for these kinds of experiments in previous work [19]. Choosing the correct loops for our measurements is very crucial since the overhead induced by the scheduling algorithms might be higher than the performance gain if we modify loops which only produce very little load imbalance. For this choice we once again rely on prior knowledge from the sources cited above.

With 5 repetitions, 4 applications, 21 scheduling technique configurations and 3 different types of computing nodes we end up with a total of 1360 jobs for the LB4OMP-WS experiments.

For the LB4MPI-WS experiments we use 5 repetitions, 3 applications and 7 scheduling technique configurations which amounts to a total of 105 jobs. This means we end up with

a total of 1465 individual jobs for our experiments. These jobs have been submitted to Slurm using batch scripts which group all jobs using the same scheduling technique configuration for one repetition.

Table 8.2: Design of the experiments

Factors		Values	Properties
Applications	Thread-level parallelism	Mandelbrot (Timestepping)	N = 262,144 | T = 200 | Total loops = 1 | Modified loops = 1
		SPHYNX Evrard Collapse	N = 1,000,000 | T = 20 |Total loops = 37 | Modified loops = 2
		SPH-EXA Sedov	N = 125,000 | T = 100 | Total loops = 16 | Modified loops = 3
		miniVite	N = 16,777,216 | T = 518 | Total loops = 39 | Modified loops = 7
	Process-level parallelism	SPH-EXA Sedov	N = 125000 | T = 100 | Total loops = 16 | Modified loops = 1
Microbenchmarks	Process-level parallelism	Dist-D	N = 400,000 | T = 20 | Total loops = 5 | Modified loops = 5 L0 (constant): 2.3 × 108 FLOP per iteration, L1 (uniform): ⌊103 , 7 × 108⌋ FLOP per iteration, L2 (normal): μ = 9.5 × 108 FLOP, σ = 7 × 107 FLOP, ⌊6 × 108, 1.3 × 109⌋ FLOP per iteration, L3 (exponential): λ = 1/3 × 108 FLOP, ⌊948, 4.5 × 109⌋ FLOP per iteration, L4 (gamma): k = 2, θ = 108 FLOP, ⌊4.1 × 106, 2.7 × 109⌋ FLOP per iteration
Thread-level Scheduling	OpenMP Standard	static	Straightforward parallelization
		guided (GSS), dynamic,1 (SS)	Dynamic and non-adaptive self-scheduling techniques
	LB4OMP	mFAC	
		mAF	Dynamic and adaptive self-scheduling techniques
		static_steal	Extension of static scheduling with a steal ratio of 25%
		RandomSel,ExhaustiveSel,BinarySel,ExpertSel	Automated DLS algorithm selection
	LB4OMP-WS	rws_static	Randomized work stealing with a static steal ratio
		laws_static	Locality aware work stealing with a static steal ratio
		rws_fac2a	Randomized work stealing with stealing based on the fac2a chunk calculation
		laws_fac2a	Locality aware work stealing with stealing based on the fac2a chunk calculation
	Chunk parameter	Default	No expert chunk size calculation
		Expert Chunk	Use the expert chunk size
	Steal Ratio	15%,25%,35%,45%	Percentage to steal from the remaining iterations on the victim thread for rws_static and laws_static
Process-level Scheduling	LB4MPI	static	Straightforward parallelization
		guided	Dynamic and non-adaptive self-scheduling technique
	LB4MPI-WS	rws_static	Randomized work stealing with a static steal ratio
		rws_guided	Randomized work stealing with stealing based on the guided chunk calculation
	Steal Ratio	15%,25%,35%,45%	Percentage to steal from the remaining iterations on the victim thread for rws_static
Computing nodes		miniHPC-KNL	Intel(R) Xeon Phi(TM) CPU 7210 (1 socket, 64 cores) P=64 cores without hyperthreading, Pinning: OMP_PLACES=cores OMP_PROC_BIND=close
		miniHPC-Xeon	Intel Broadwell E5-2640 v4 (2 sockets, 10 cores each) P=20 cores without hyperthreading, Pinning: OMP_PLACES=cores OMP_PROC_BIND=close
		miniHPC-Cascade Lake	Intel Xeon Gold 6258R (2 sockets, 28 cores each) P=56 cores without hyperthreading, Pinning: OMP_PLACES=cores OMP_PROC_BIND=close
Metrics		T_{Par}	Parallel execution time of the loops
		$c.o.v.$	Coefficient of variation
		N_{Steal}	Number of successful stealing operations

8.3 LB4OMP-WS Experiments

These experiments have been performed by setting the thread number using *OMP_NUM_THREADS* to the total number of cores. Therefore we use 64 cores on the KNL nodes, 20 on the Xeon nodes and 56 on the Cascade Lake node. The threads have been pinned to the cores by setting *OMP_PLACES* to 'cores' and *OMP_PROC_BIND* to 'close'. The main aim of these experiments is to compare the performance of our techniques to the ones already implemented in LB4OMP.

8.3.1 SPHYNX Evrard Collapse

SPHYNX simulates an Evrard collapse. It has been shown by Korndörfer et. al. [19] that this application is a suitable candidate for experiments with thread-level scheduling. The two loops that have been modified are *findNeighbors* and *treewalk*.

In these experiments our two techniques did not perform very well. On the Xeon nodes 8.1 and Cascade Lake node 8.3 we got worse results than with static steal. On the KNL nodes 8.2 we they performed about the same but are still behind factoring methods and some of the automatic algorithm selection techniques.

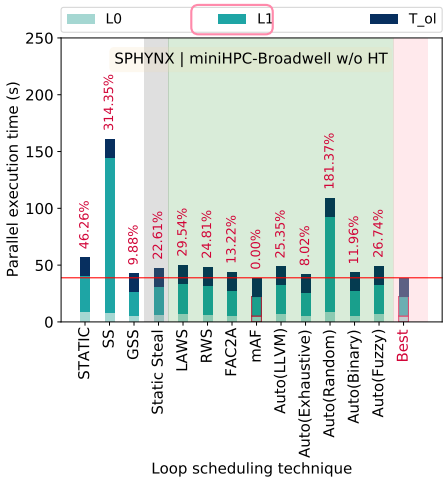


Figure 8.1: T_{par} for SPHYNX Evrard collapse on miniHPC-Xeon

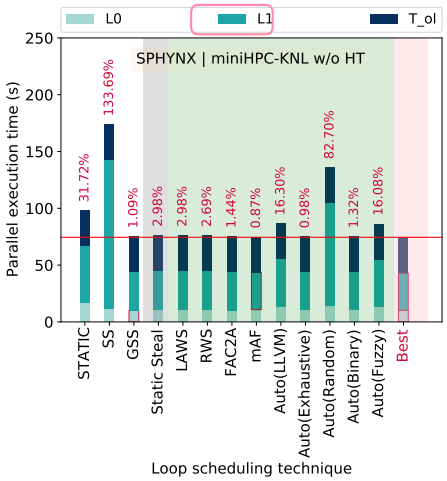


Figure 8.2: T_{par} for SPHYNX Evrard collapse on miniHPC-KNL

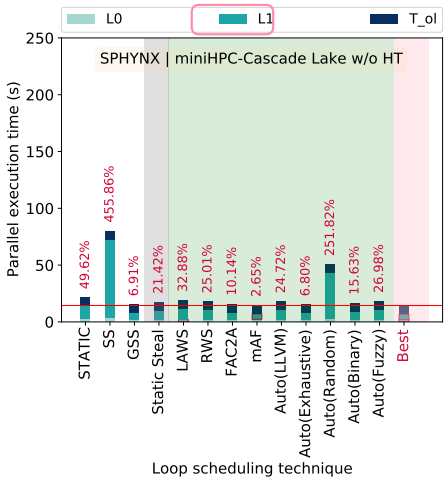


Figure 8.3: T_{par} for SPHYNX Evrard collapse on miniHPC-Cascade Lake

8.3.2 miniVite

MiniVite is a proxy app that implements a single phase of Louvain method in distributed memory for graph community detection. As input we are using the "DIMACS10/rgg_n_2_24_s0" matrix from the SuiteSparse Matrix Collection[22]. Due to the prior experience with this application we know that its main loop *distLouvainMethod* is irregular. The results in Figures 8.4 and 8.5 show the that the exhaustive selection from the auto methods achieves the best performance on miniHPC-Xeon and miniHPC-KNL. On miniHPC-Cascade Lake 8.6 it still has a very good result coming in as a close second. In these experiments RWS and LAWS on average perform slightly better than the static steal technique. However even on this data-intensive application we do not see major improvement by using the LAWS method. This might be due to the first touch policy and how the data is initialized in miniVite. A further investigation is necessary to determine if this was the cause for the performance results of LAWS.

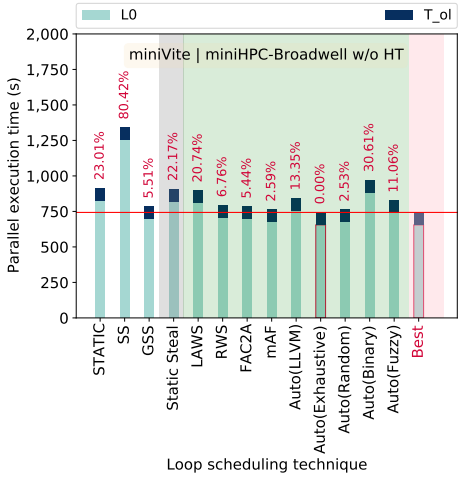


Figure 8.4: T_{par} for miniVite on miniHPC-Xeon

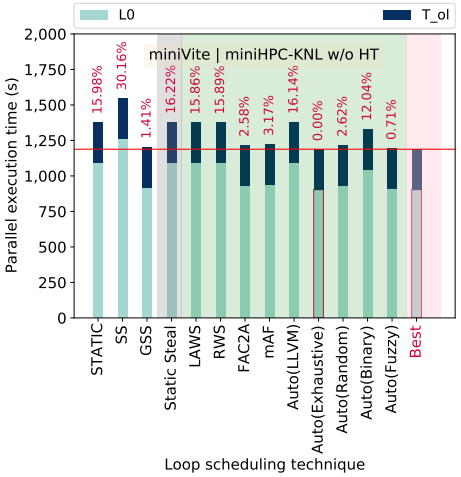


Figure 8.5: T_{par} for miniVite on miniHPC-KNL

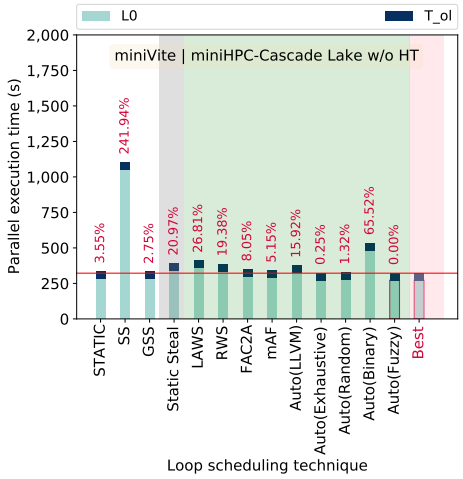


Figure 8.6: T_{par} for miniVite on miniHPC-Cascade Lake

8.3.3 Mandelbrot

Mandelbrot is due to the irregular nature of the problem of generating Mandelbrot sets a good candidate for experiments focusing on thread-level scheduling. In these experiments we run a timestepping version of Mandelbrot for 200 timesteps and use our scheduling algorithms on the only parallelized loop in the code. The trend we saw in the results from the other applications continues. RWS and LAWS again show very similar results to static steal ranking them slightly worse in performance when compared to the factoring and automatic selection techniques.

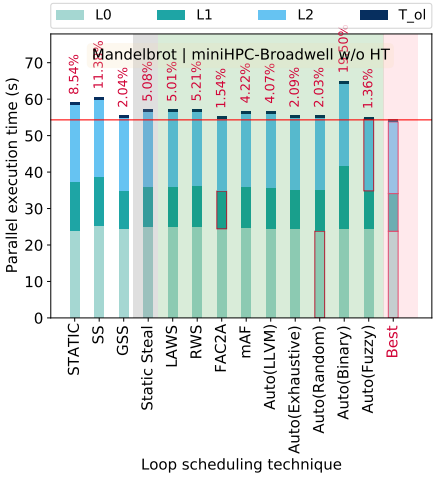


Figure 8.7: Parallel Execution time of Mandelbrot on miniHPC-Xeon

8.3.4 SPH-EXA Sedov

8.4 LB4MPI-WS Experiments

We chose to perform these experiments all on the miniHPC-Xeon nodes. We can use a higher number of total ranks this way because that type of computing node has the most instances on our cluster. The number of ranks varies between the applications and has been chosen in a way that it makes sense in relation to the input size of the experiments. We will elaborate on the detailed rank configuration further in each of the following three sections.

8.4.1 Dist

Dist is a synthetic benchmark. These experiments have been performed on 8 nodes with 20 ranks per node. The variable N in table 8.2 corresponds to the number of lines we read from a file. This file contains a number x in each row. We then perform a simple arithmetic operation x times. There are 5 loops each using data from a different file. The numbers in the files belong to a distribution and therefore we named the loop after the corresponding distribution.These experiments have been performed on 8 nodes with 20 ranks per node.

An important fact when looking at the results is that the two techniques *STATIC* and *GSS* have been performed on Dist benchmark that works with replicated data, while we use the distributed data version Dist-D for RWS. This is due to the fact that the only technique that currently works with distributed data in LB4MPI-WS is RWS. Likewise we can not use RWS when dealing with replicated data. The only difference between Dist and Dist-D is that in Dist-D we read only part of the numbers in each file into an array and therefore distribute the data across the ranks.

When looking at the results from Figure 8.11 we can clearly see that the RWS techniques perform very similarly across the board. This is to be expected. The reason for that is that even though we have a vastly different number of successful steals with the different steal ratios 8.12 the data that is being transmitted is very small (one integer per iteration stolen). Thus we have very little overhead from the additional steals.

A second observation to be made is that the c.o.v. which is a load imbalance metric also

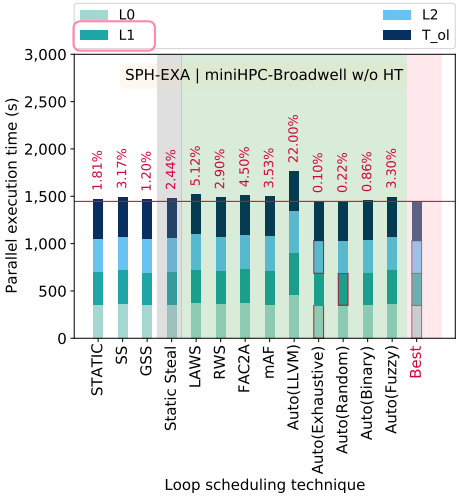


Figure 8.8: T_{par} for SPH-EXA Sedov on miniHPC-Xeon

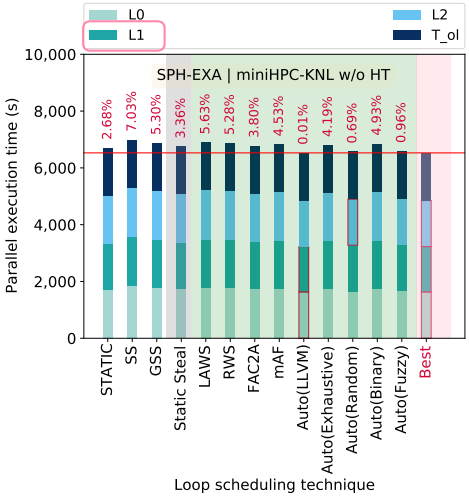


Figure 8.9: T_{par} for SPH-EXA Sedov on miniHPC-KNL

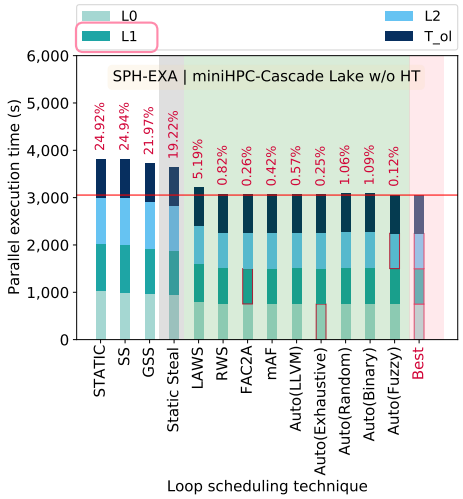


Figure 8.10: T_{par} for SPH-EXA Sedov on miniHPC-Cascade Lake

seems to behave as expected. This can be seen in Figure 8.13. With a static scheduling we are going to have a higher c.o.v. if the loop is irregular because it is just a static distribution of work. GSS should have a low c.o.v. and a relatively high overhead which is also true in our case. Out of the steal ratios 25% performs the best in terms of c.o.v. and parallel execution time in the exponential and the uniform loop. The default of 25% might be a viable choice which is also what has been chosen by the authors of static_steal in OpenMP. Lastly we can also say that the RWS implementation performs quite well considering that it has a lower parallel execution time across the board when compared to static scheduling with the replicated version. This has to be taken with a grain of salt as the transmitted data is very small. We can however state that this observation indicates that the overhead from the algorithm itself (i.e. ignoring the cost of communication between thief and victim) is rather small.

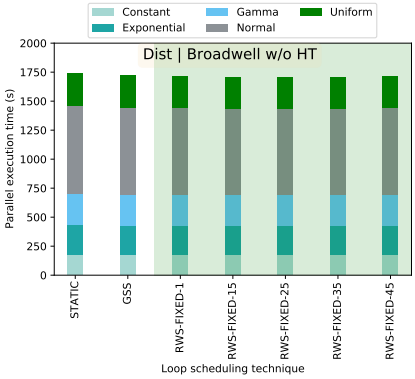


Figure 8.11: T_{Par} for Dist/Dist-D

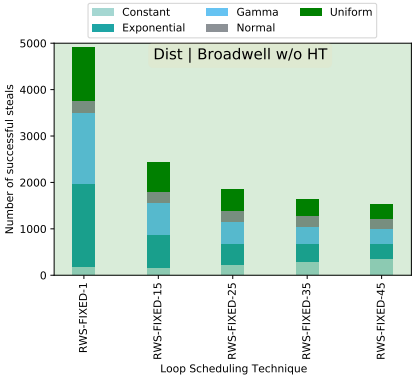


Figure 8.12: Number of successful steals for Dist

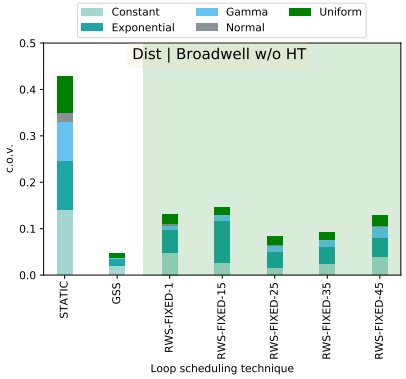


Figure 8.13: C.O.V. for Dist

8.4.2 SPH-EXA Sedov

We chose to modify the computeIAD loop as it has a high arithmetic intensity. The experiments have been performed on 4 nodes with 4 ranks each. As we are only using distributed data it is not possible to evaluate results for the static and GSS techniques.

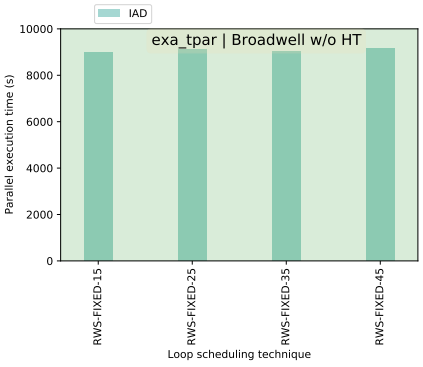


Figure 8.14: T_{Par} for SPH-EXA Sedov

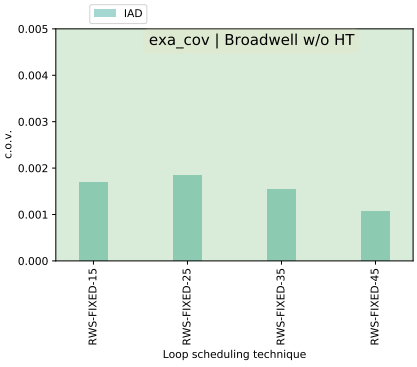


Figure 8.15: C.O.V. for SPH-EXA Sedov

This time around we get the best results with a steal ratio of 15% as opposed to the 25% in Dist. The parallel execution time is however still fairly close to each other for each steal ratio. This indicates that it might sometimes be worth it to tune *LB4MPI_STEAL_RATIO* when trying to optimize the scheduling of a loop.

9

Conclusions & Future Work

We introduced LB4OMP-WS, an extension of LB4OMP, which adds a random work-stealing algorithm as well as a locality aware work-stealing algorithm to the repertoire. We showed that the two algorithms are on average up to par with the static steal algorithm from LLVM OpenMP. Furthermore we demonstrated that the choice of steal ratio does not play a very significant role in terms of performance of our algorithms.
In addition to that we also presented LB4MPI-WS, which is an extension to LB4MPI. The very limited number of experiments produced promising results. For the distributed version of Dist microbenchmark we achieved results that can rival the work-sharing algorithms that were used in the replicated version. This work acts as a small step of introducing LB4MPI into the world of scheduling multiprocessing applications that work with distributed data.
Both LB4OMP-WS and LB4MPI-WS provide many opportunities for future work. First of all the number of repetitions used was very low in our experiments. This was mainly due to time constraints. A further analysis with additional applications and more repetitions would be a very good first step. For LB4OMP-WS a good idea might be to dive deeper into the code of the applications and have a closer look at how and where data is initialized. This would help gauging the impact of the first-touch policy on our techniques.
Furthermore it would be interesting to see more scheduling techniques that can deal with distributed data in LB4MPI-WS. Adding additional counters such as the number of victim probings could be very interesting. Last but not least performing more experiments with applications that have both a version for replicated and distributed data (similar to Dist and Dist-D) should provide a lot of insight on scalability and performance of the library.

Bibliography

[1] A. Mohammed and F. M. Ciorba. Research Report - University of Basel, Switzerland, 2018. URL https://drive.switch.ch/index.php/s/aanqAdp3X2Fxsoe.

[2] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[3] Boost. Boost C++ Libraries. http://www.boost.org/, 2022. Last accessed 2022-05-10.

[4] F Warren Burton and M Ronan Sleep. Executing functional programs on a virtual tree of processors. In *Proceedings of the 1981 conference on Functional programming languages and computer architecture*, pages 187–194, 1981.

[5] Rubén M Cabezón, Domingo Garcia-Senz, and Joana Figueira. Sphynx: an accurate density-based sph method for astrophysical applications. *Astronomy & Astrophysics*, 606:A78, 2017.

[6] 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.

[7] Quan Chen and Minyi Guo. Locality-aware work stealing based on online profiling and auto-tuning for multisocket multicore architectures. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(2):1–24, 2015.

[8] Ciorba, Florina M. minihpc: Small but modern hpc. https://hpc.dmi.unibas.ch/en/research/minihpc/. Accessed:2022-07-15.

[9] James Dinan, D Brian Larkins, Ponnuswamy Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11. IEEE, 2009.

[10] Derek L Eager, Edward D Lazowska, and John Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load sharing. *Performance evaluation*, 6(1): 53–68, 1986.

[11] Message P Forum. Mpi: A message-passing interface standard. Technical report, USA, 1994.

[12] Sayan Ghosh, Mahantesh Halappanavar, Antonino Tumeo, Ananth Kalyanaraman, and Assefaw H. Gebremedhin. Minivite: A graph analytics benchmarking tool for massively parallel systems. In *2018 IEEE/ACM Performance Modeling, Benchmarking*

and Simulation of High Performance Computer Systems (PMBS), pages 51–56, 2018. doi: 10.1109/PMBS.2018.8641631.

[13] Danilo Guerrera, Aurélien Cavelan, Rubén M Cabezón, David Imbert, Jean-Guillaume Piccinali, Ali Mohammed, Lucio Mayer, Darren Reed, and Florina M Ciorba. Sphexa: Enhancing the scalability of sph codes via an exascale-ready sph mini-app. *arXiv preprint arXiv:1905.03344*, 2019.

[14] Intel Corporation. Intel® oneapi dpc++/c++ compiler. https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compiler.html. Accessed:2022-05-15.

[15] Benoit B Mandelbrot. Fractal aspects of the iteration of z→ λz (1-z) for complex λ and z. *Annals of the New York Academy of Sciences*, 357(1):249–259, 1980.

[16] Evangelos P. Markatos and Thomas J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed systems*, 5(4):379–400, 1994.

[17] Gordon E Moore et al. Cramming more components onto integrated circuits, 1965.

[18] Ananya Muddukrishna, Peter A Jonsson, and Mats Brorsson. Locality-aware task scheduling and data distribution for openmp programs on numa systems and manycore processors. *Scientific Programming*, 2015, 2015.

[19] Jonas H. Müller Korndörfer, Ahmed Eleliemy, Ali Mohammed, and Florina M. Monica Ciorba. Lb4omp: A dynamic load balancing library for multithreaded applications. *IEEE Transactions on Parallel and Distributed Systems*, pages 1–1, 2021. doi: 10.1109/TPDS.2021.3107775.

[20] OpenMP Architecture Review Board. OpenMP application program interface version 5.1, November 2020. URL https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf.

[21] Guilherme P Pezzi, Márcia C Cera, Elton Mathias, Nicolas Maillard, and Philippe OA Navaux. On-line scheduling of mpi-2 programs with hierarchical work stealing. In *19th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'07)*, pages 247–254. IEEE, 2007.

[22] Texas A&M University. Suitesparse matrix collection. https://sparse.tamu.edu/DIMACS10/rgg_n_2_24_s0. Accessed:2022-05-07.

[23] Yizhuo Wang, Yang Zhang, Yan Su, Xiaojun Wang, Xu Chen, Weixing Ji, and Feng Shi. An adaptive and hierarchical task scheduling scheme for multi-core clusters. *Parallel computing*, 40(10):611–627, 2014.

[24] Jixiang Yang and Qingbi He. Scheduling parallel computations by work stealing: A survey. *International Journal of Parallel Programming*, 46(2):173–197, 2018.

[25] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Workshop on job scheduling strategies for parallel processing*, pages 44–60. Springer, 2003.

Universität Basel

Philosophisch-Naturwissenschaftliche Fakultät

Erklärung zur wissenschaftlichen Redlichkeit und Veröffentlichung der Arbeit (beinhaltet Erklärung zu Plagiat und Betrug)

Titel der Arbeit:

Name Beurteiler*in: Prof. Dr. Florina M. Ciorba

Name Student*in Gian-Andrea Wetten

Matrikelnummer: 12-720-777

Mit meiner Unterschrift erkläre ich, dass mir bei der Abfassung dieser Arbeit nur die darin angegebene Hilfe zuteil wurde und dass ich sie nur mit den in der Arbeit angegebenen Hilfsmitteln verfasst habe. Ich habe sämtliche verwendeten Quellen erwähnt und gemäss anerkannten wissenschaftlichen Regeln zitiert.

Ort, Datum: Basel, 26.05.2022 Student*in: Wetten Gian-a.

Wird diese Arbeit veröffentlicht?

○ Nein

◉ Ja. Mit meiner Unterschrift bestätige ich, dass ich mit einer Veröffentlichung der Arbeit (print/digital) in der Bibliothek, auf der Forschungsdatenbank der Universität Basel und/oder auf dem Dokumentenserver des Departements / des Fachbereichs einverstanden bin. Ebenso bin ich mit dem bibliographischen Nachweis im Katalog SLSP (Swiss Library Service Platform) einverstanden. (nicht Zutreffendes streichen)

Veröffentlichung ab: _____

Ort, Datum: Basel, 26.05.2022 Student*in: Wetten Gian-a.

Ort, Datum: _____ Beurteiler*in: _____

Diese Erklärung ist in die Bachelor-, resp. Masterarbeit einzufügen.