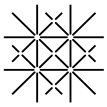


University of Basel

Task Scheduling and Work Stealing in the DAPHNE Runtime System

Master's 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

Jonathan Giger
j.giger@stud.unibas.ch
19-067-511

Acknowledgments

First and foremost, I would like to express my gratitude to Prof. Dr. Florina Ciorba for giving me the opportunity to conduct my Master's Thesis in the High Performance Computing research group. I would like to sincerely thank my supervisors Dr. Ahmed Eleliemy and Gabrielle Poerwawinata for the continuous support and valuable feedback throughout my Thesis. I would also like to thank Marc Hennemann for the grammatical and structural suggestions that helped form this Thesis. Finally, I would like to thank Patrick Damme and the entire DAPHNE team for their support and insights in response to my questions. I greatly appreciate all that I have learned and discovered while completing this Thesis.

Abstract

Modern research relies on the processing of large datasets using High Performance Computing, Big Data, and Machine Learning operations. DAPHNE is a system infrastructure for such integrated data analysis pipelines that provides language abstractions, compilation and runtime techniques, and built-in parallelization features for researchers to process such datasets seamlessly. In order to offer parallelization features that are nearly transparent to the researcher while still delivering high performance, versatile scheduling techniques with proven track records must be evaluated and implemented. As computing hardware is constantly evolving, often becoming more heterogeneous, scheduling heuristics are becoming more complex and new factors must be taken into account. This Thesis surveys task scheduling techniques and work-stealing mechanisms from previous research on runtime systems in the High Performance Computing field. The advantages of each technique are interpreted with respect to various application types and hardware systems that may be employed by a researcher using DAPHNE to process data. Locality-aware task scheduling techniques for integrated data analysis pipelines are then implemented in DAPHNE and the performance is evaluated using publicly available datasets.

Table of Contents

1 Introduction

The scientific community relies on data processing now more than ever before. The demand for data processing comes in many shapes and sizes. There are Big Data applications which rely heavily on disk I/O while having relatively simple computational steps, there are Machine Learning applications which have heavy computational requirements but often result in small output datasets, and there are also High Performance Computing applications which usually rely on the applications being parallelizable in nature. These computational tasks, which when combined with the data to be processed are referred to as pipelines and are similar in all of these scenarios, yet still differ in their requirements enough that there has not yet been a full convergence on the software infrastructure to handle them. While all three of these domains are heavily researched in their own fields, a single unified system combine all three into a single system infrastructure for data analysis is an open research topic.

While datasets are becoming larger and computing clusters are becoming more powerful, research and development communities are striving to create an system infrastructure that can handle integrated data analysis pipelines with a wide range of input data, performance, and computational requirements. Input data requirements often restrict the data types that software frameworks support. Performance requirements are usually met by both programming solutions that bring the code closer to the bare-metal, and also supporting the distributed execution of code across multiple computing nodes. These computing nodes are in turn often becoming more heterogeneous, especially in large clusters that contain specialized hardware such as FPGAs and other hardware accelerators. The hardware itself is also becoming increasingly more complex, with shared-memory systems that do not have uniform access to memory becoming more common.

There has always been a convergence on the hardware level between Machine Learning, Big Data, and High Performance Computing applications, since all of these use cases are applied using Data Centers with similar computing units. However, an integrated system infrastructure that creates a convergence on the software level is still in the early stages of research and development. One such project that aims to create this convergence is DAPHNE, an integrated **D**ata **A**nalysis **P**ipelines for large-scale data management, **H**igh-

performance computing, and machi**NE** learning [1].

The DAPHNE project is a system architecture built from scratch in C++ to process workloads that contain integrated data analysis pipelines. It is designed to be open and extensible which allows users to implement their own use cases through use of the Domain Specific Language, DaphneDSL. The user's code is then passed through the DAPHNE compiler and is optimized to improve utilization of the hardware cluster's resources. This process takes advantage of techniques such as reordering, reducing redundancy, and taking advantage of matrix sparsity. The modular use of kernels in the DAPHNE backend also allows for easy extensibility by adding new kernels or adjusting scheduling knobs.

Use cases such as earth observation, semiconductor manufacturing, and automotive vehicle development have shown the potential of this convergence on the software level in real-world applications. These real-world use cases consist of a combination of Big Data, Machine Learning, and High Performance Computing characteristics that have since not been solved on a single system in an efficient manner. Consolidating these tasks into a single integrated system not only simplifies the development of these applications for the programmer, but also allows for clever optimizations to be made when consolidating certain operations. In the context of DAPHNE, the vectorized execution engine splits datasets into chunks which can be independently executed, while combining adjacent operations when applicable in order to optimize the transport of data.

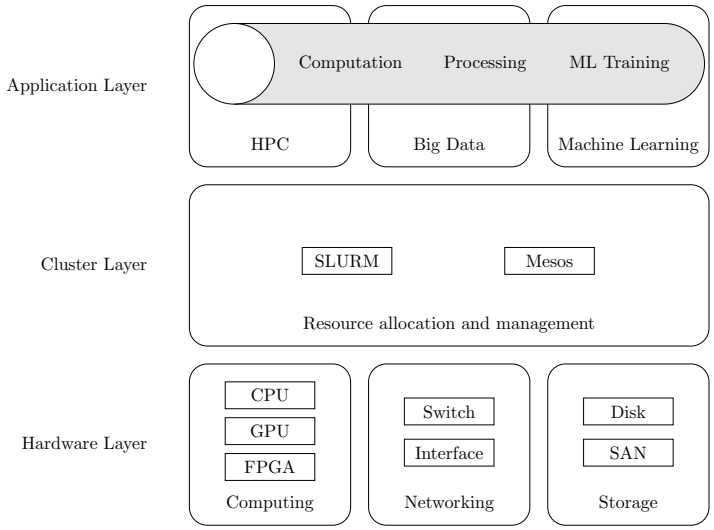


Figure 1.1: Ecosystem for integrated data analysis pipelines, adapted from [14]

The goal of a integrated system infrastructure for data analysis pipelines is not just to be able to output the correct result, but also to minimize the execution time needed on the computing cluster, freeing up computing resources for other jobs and saving energy. One factor that affects a programs execution time is the scheduling of the program's instructions. When scheduling a program's instructions, the order, size, and distribution of the instructions must be carefully optimized. This can be done by taking into account heterogeneous

[1] https://daphne-eu.eu/

hardware, natural variance in code execution time, and sparseness properties of the input dataset. When developing a scheduler for such a versatile and extensible system it is important to weigh many factors such as data locality, load imbalance, and overhead in order to achieve an optimal balance that is efficient for the given use case.

Another dimension of complexity emerges when taking into account systems with Non-Uniform Memory Access (NUMA) which brings another layer of possible optimizations when designing a scheduler for such a system infrastructure. This scenario opens the possibility of being able to access an address in memory, but the delay in accessing the memory can vary depending on where it is being accessed from. This requires not only being aware of where data is stored in memory, but also anticipating when in the program's execution it will be called and adjusting the scheduling behavior accordingly. Research in this area exists and different approaches to handling this complexity already exist in High Performance Computing libraries such as OpenMP and in implementations of this library such as the LLVM libomp runtime library.

1.1 Motivation

While scheduling is a comprehensive field of research, the methods used are tightly connected to their context. A scheduling technique that is effective for small shared-memory system will not necessarily be effective for a large distributed-memory computing cluster with heterogeneous hardware. This makes the case for a highly customizable system infrastructure that is versatile enough to be used in all distributed computing contexts. The dimensions in the design of a scheduler include the partitioning of the load, order of execution, design of the queues, work stealing, and additional optimizations which can be made during runtime. In order to design an effective scheduler, the runtime system must take into account factors such as the size of the individual computation chunks, which end of the queue to dequeue from, if the layout of the queues should reflect the NUMA architecture, how nodes should behave at the end of their respective queue, and what strategies to use to split the input data. All of these factors can be corroborated to maximize data locality and minimize scheduling overhead to the best extent possible for each use case. This thesis will explore the use of state of the art scheduling techniques and identify which of these techniques are best suited for scheduling integrated data analysis pipelines on large scale heterogeneous systems.

The varying sparsity of data, complex interconnections of dependencies in operations, and heterogeneous hardware used in the processing of data analysis pipelines demands a comprehensive scheduling solution that takes all of these factoring into account. Highly sparse input data requires larger chunk sizes to account for the sections of data that gets processed quickly because they are empty. A complex series of operations can often be executed by various combinations of kernels, some of which can be parallelized more effective than others. The homogeneity of the hardware used to process this data can also cause scheduling abnormalities that can often be accounted for beforehand. This Thesis will analyse these aspects of scheduling and possible in the context of DAPHNE.

1.2 Contribution

The contribution of this Thesis is a modification to the load partitioner and the allocation of queues in the DAPHNE work scheduler. In order to implement a work-stealing scheduler, we first allow for multiple queues to be created to hold work planned for execution. Then we modify the work allocation function to distribute work to these queues using block, cyclic, and a combination technique. Finally, a hierarchical implementation in which restricts work stealing to only certain appointed foreman workers is applied and evaluated.

1.3 Outline

The following chapters of this Thesis will provide a summary of the background, implementation, and evaluation of the implementations for this Thesis. In Chapter 2 the Terminology used in this Thesis is introduced and background on the test system, software environment, and evaluation applications are provided. Then in Chapter 3 work related to task scheduling and work stealing is summarized and the interesting concepts in the context of this Thesis are highlighted. Chapter 4 provides the methodology of the implementation and hypothesizes what the result might be. In Chapter 5 the results are presented and interpreted. Chapter 6 then summarized what can be learned from this Thesis. Finally, Chapter 7 speculates on possible future extensions of this work and provides insight to what could be achieved.

2

Terminology and Background

In the context of scheduling many algorithms from other work are used and many concepts are referred to by abbreviations. This list encompasses most abbreviations and technical terms that are used in this Thesis. Some terms can have multiple definitions depending on the context they are used in. Since the concept of work-stealing schedulers is an evolving field of research, the definitions of these terms may also become more loosely interpreted over time.

2.1 Terminology

Self-scheduling - A work assignment principle in which a worker obtains a task to execute once it completed the previous task.

Work-sharing - A scheduling approach following the self-scheduling principle with a centralized work queue from which workers obtain tasks.

Work-stealing - Another scheduling approach which uses distributed work queues in which workers dequeue tasks from following the self-scheduling principle.

Worker - Refers to individual software processing units that execute tasks.

Foreman - A type of worker that executes tasks and also coordinates the scheduling for other workers. (Also referred to as a Shepherd [23])

Cluster - Multiple nodes connected by a fast interconnect network.

Node - One physical server with shared-memory.

Thread - Can refer to software threads in a programming context or hardware threads in a system architecture context.

Split - A unit representing the smallest chunk that work can be divided into, in DAPHNE either rows, columns, or scalars. Somewhat interchangeable with iterations in other scheduling contexts.

Work Partitioning - The scheduling step in which input work is grouped into tasks of various size, often exploiting data and/or functional parallelism. [7]

Work Assignment - Refers to both the mapping of tasks to workers on the software level, and also the mapping of workers to hardware execution units (CPUs, GPUs, FPGAs) on the hardware level.

DSL - Domain Specific Language

DAPHNE - System infrastructure for large-scale integrated data analysis pipelines.

IR - Intermediate Representation

DAPHNE IR - DAPHNE Intermediate Representation (Dialect of MLIR)

Runtime system - An engine that translates a DSL into machine code for execution. [8]

Operator - A mapping or function in mathematics that on elements to produce other elements.

Workflow - A series of repeatable steps performed on a dataset.

Static Scheduling - Scheduling algorithms in which the size of the chunks are known before the program is executed.

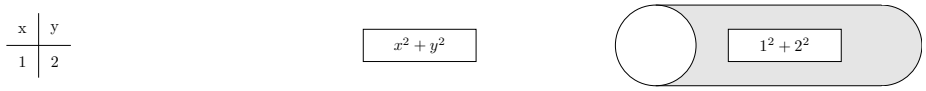
Dynamic Scheduling - Scheduling algorithms where the size of the chunks are known once the size of the input data is known.

Adaptive dynamic scheduling - Scheduling algorithms where the size of the chunks are decided during the program's runtime.

DLS - Dynamic Loop Self-Scheduling

Task - An object containing the functions to be executed, packaged in way that can be scheduled.

Pipeline - A term referring to a task being executed on input data.



(a) Input Data (b) Task (c) Pipeline

Figure 2.1: Visual representation of a Data Analysis Pipeline made up of a task and data

Vectorized pipeline - Refers to multiple pipelines executed in parallel.

Sparsity - The number of non-zero elements divided by the number of elements in a matrix (also referred to as density)

I/O - Input / Output (Usually from or to a Disk)

2.2 Background

This Thesis analyses the effectiveness of works stealing schedulers in the context of Data Analysis Pipelines. The experiments will be performed on the DAPHNE infrastructure in order to provide a realistic estimation of the performance of work-stealing scheduling concepts for real data analysis tasks.

2.2.1 DAPHNE Infrastructure and Applications

DAPHNE is designed to be open and extensible. Since mathematical operations are implemented as kernels, new operations can be added or modified by simply adding or modifying the relevant kernel file. When a kernel is implemented, the vectorization of the pipeline operation is handled by the DAPHNE IR using another kernel named vectorized pipeline. This vectorizing kernel acts as an operation which can be used with other operations, and sends its input to the multi-threaded wrapper whose implementation will be discussed in

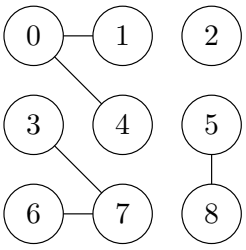
more detail in Chapter 4.

2.2.1.1 Built-in Kernels

One of the key building blocks of DAPHNE is that operations in the IR (Intermediate Representation) can be implemented by kernels, which in themselves are highly extensible and versatile. The DAPHNE IR has a set of common built-in kernels, however the user can also add their own kernels as needed. These kernels are highly type flexible and allow for great functionality while minimizing code duplication. Each kernel has input sand outputs, and can have multiple multiple implementations to handle different input and output combinations that the kernel will support.

2.2.1.2 Dense and Sparse Matrix Representation

Since some datasets are made up of matrices that are excessively large, but have very few non-zero values, it can become feasible to represent these matrices using a separate notation. Matrices that have many non-zero values which contain any given data format can be represented by a combination of three columns, one for the row location of a value, one for the column location of a value, and one for the value itself. This is referred to as the COO matrix format. While file input in DAPHNE uses the COO matrix format, the representation inside the DAPHNE IR for sparse matrices is the CSR format, which is similar to COO format, but the row indexes are compressed.



(a) Verteces and Edges

0	1	0	0	1	0	0	0	0
1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0
1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	1	0
0	0	0	1	0	0	1	0	0
0	0	0	0	0	1	0	0	0

(b) Dense Representation

5	8
6	7
3	7
0	4
0	1

(c) Sparse Representation

Figure 2.2: Dense and Sparse representations of an undirected graph

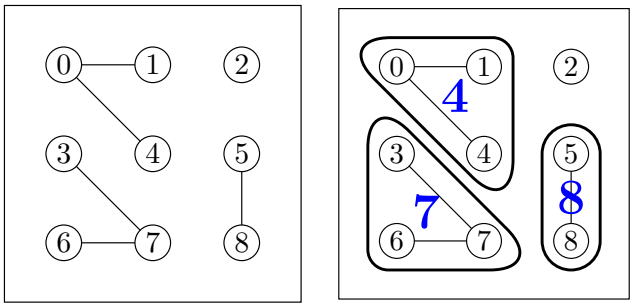
Using sparse matrix representations allows for computations with much larger matrices than would otherwise be possible with just a dense representation. Since only the elements that contain non-zero values have to be stored in memory the amount of memory needed to store a matrix using sparse representation in memory would scale linearly with the amount of non-zero values in the matrix. Currently, the use of sparse representations in DAPHNE can be enabled with the select-matrix-representation argument, which will automatically use the sparse representation when less than 10% of the values in a matrix contain non-zero values.

Sparse matrices also result in much fast computations than their dense counterparts. Rows that do not contains any non-zero values do not add any computational effect when performing the calculations instructed by the kernel. Due to this different in implementa-

tion, kernels must be implemented multiple times. Since some kernels can have more than one input, every possible combination of dense and sparse inputs to the kernel must be implemented separately.

2.2.1.3 Connected Components Algorithm

The connected components algorithm, also referred to as blob extraction or region labeling, is a graph theory application that finds the largest subsets of a given set are found and labels them uniquely. The input of this application is a list of connected vertexes of a graph, and the output will be a list of all vertexes with their respective subset label.



(a) Input Graph (b) Output Graph

Figure 2.3: Input and result of the connected components algorithm

This algorithm is an ideal test case for scheduling performance in DAPHNE as it includes a loop, multiple operators that can be used in a vectorized pipeline, and intermediate sparse outputs. The input to this algorithm is a CSV file that contains a list of all the edges in the graph, and a metadata file that contains the total number of nodes in the graph, which in this case will also be the number of rows and columns of the input matrix. This input matrix then undergoes a serious of transpose, element-wise multiplication, aggregation, maximum, comparison, and summation operations which will label the beginning and end point of each edge with the number of the largest connected component that the corresponding node is part of.

2.2.1.4 Slurm Integration

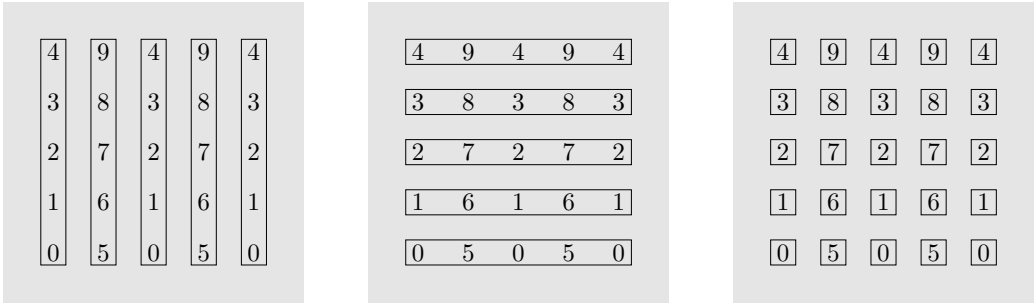
High performance computing clusters are usually shared with a number of users and run a variety of different computation jobs. Slurm is a workload manager that can coordinate jobs from several users on a cluster and relieve contention on computing resources by managing a queue of pending work. A cluster that uses Slurm to manage workloads can be configured in one of two ways, exclusive allocation and non-exclusive allocation. When using exclusive allocation an entire computing node is reserved for one job, regardless of how many CPU threads are requested or required by the job. However in a non-exclusive configuration, a computing node can be allocated to more than one job at a time. In this case Slurm employs thread pinning which locks processes to a specific set of CPU threads in order to divide the resources of that computing node in fair manner. DAPHNE applications can also be run on clusters using Slurm, however certain precautions must be taken to pre-

vent the non-exclusive provisioning of resources from affecting the results of the experiments in this Thesis.

When the Slurm Task/Affinity plug is enabled, multiple jobs can be executed on a single node concurrently. Each job is assigned a set number of CPU cores to utilize and Slurm then schedules jobs to nodes based on the number of available CPU cores on that node. This limiting of CPU resources is achieved through either the setaffinity function, UNIX cgroups, or both. How both of these systems exactly function may vary by operating system, however the definitions in this Thesis pertain to Linux.

2.2.1.5 Vectorized Execution Engine

Matrices can be split by rows, columns, or scalars. This versatile approach allows for efficient scheduling of "tall" and "wide" matrices, especially when they are sparse. In some applications such as linear regression there could be large datasets encoded into a matrix which only has a handful of columns, these would not result in balanced schedules if split by columns as there may not even be enough rows for the number of CPU cores in the system. Even if there are sufficient rows to create at least one task per CPU, the eleven other scheduling schemes described in this Thesis may not have enough input splits to produce meaningful chunk sizes.



(a) Column Split (b) Row Split (c) Scalar Split

Figure 2.4: Vectorizable splits of a dense matrix

2.2.2 Scheduling Schemes

The simplest scheduling technique is a simple static partitioner. With the technique the total units of work to be done is divided into P tasks, where P is the number of processing units. This scheduler very common due to its ease of implementation and low overhead. Any variation in execution time of a single task will cause load imbalance when using this scheduler.

In order to prevent this load imbalance from taking place, a strategy with fine-grained tasks which allow the scheduler to allocate the tasks at runtime, referred to as self-scheduling (SS) was suggested. This technique results in the most fine-grained distribution of work among the processing units, as the work can be split and reallocated by the smallest possible chunk size. While this technique theoretically results in the lowest possible amount of load imbalance, the cost of the scheduling overhead can far outweigh the benefits. A minimum

chunk size parameter passed from the user could optimize this trade off.

In order to balance the low overhead of large chunk sizes and low load imbalance of small chunk sizes, there are a number of scheduling schemes that dynamically calculate chunk sizes mathematically based on the size of the input work, the number of workers, and some other factors that may even be specific to the system that the application will be running on. In Guided self-scheduling (GSS), the load partitioner aims to find a balance between load imbalance and scheduling overhead by creating tasks with a large chunk size for the beginning of the programs execution and then tasks with a small chunk size at the end of the execution. This technique still falls under the category of self-scheduling since the determination of which processing units each task is execution on still occurs at runtime. Similar to Guided self-scheduling, Trapezoid self-scheduling (TSS) also utilizes the advantages of small and large chunks by decreasing the chunk size at run time. In this case the chunk size is decreased linearly, as opposed to guided where the chunk size is decreased according to a division function. The main advantage of TSS is that the function to calculate the chunk size requires very little computational resources.

In contrast to the previous methods which simply calculate the chunk size of a given task using a continuous function, the chunk sizes can be computed in batches which complements the symmetrical nature of processing units on a computing node. The factoring scheme (FAC) has a "step" variable, which only gets increased once the chunk sizes for one batch of tasks have been computed, meaning one task per processing unit. The factoring technique can be further customized by dividing the chunk size by a variable x, a variable which can be set by the user and further reduces the chunk size. For practical use, a value of 2 is used for this variable, and this variation referred to as FAC2. A combination of factoring and trapezoid self-scheduling exists with the name Trapezoid Factoring Self-Scheduling (TFSS). According to this technique, chunk sizes are calculated in a linearly decreasing manner, however the the computations are done in batches.

While all of the previous technique that have variable chunk sizes up until now have had a decreasing chunk size as the program executes, there are also techniques that increase the chunk size. Fixed Increase Self-Scheduling (FISS) uses a calculation similar to factoring, however the chunk size is increase instead of decreasing [26]. Similar to FISS, Variable Increase self-scheduling (VISS) also increments the chunk size, however amount of the chunk size increase is reduced as the program is executing.

In addition to simply computing the chunk size using a mathematical formula, more advanced techniques exist which calculate the size of a chunk at run time using information from the previously executed chunks to influence the calculation. In Performance-based Loop self-scheduling (PLS) the size of a chunk is calculated based on a workload ratio that is computed during run time. A variation of the above adaptive scheduling technique is Probabilistic self-scheduling (PSS). With this technique the size of a chunk is calculated using the number of currently idle processors.

The last scheduling technique that is explored in this Thesis is Modified Fixed-Size Chunk self-scheduling (MFSC). This is a simple technique which is an implementation of self-scheduling that uses one chunk size for all tasks in the application. This single chunk size is calculated using a logarithmic function with the number of workers and the total

number of work units. The goal of the mathematical function to compute the chunk size, is to end up with the same resulting number of chunks as when using the factoring (FAC) technique.

3
Related Work

The book [6] provides a comprehensive overview of task scheduling on shared-memory architecture. The book separates parallel environments into two sections, task parallelism and data parallelism. In data parallelism, the focus is on distributing data across processing units, as opposed to in task parallelism where the focus is on distributing tasks among processing units which will be executing on the same data. The book then focused on task scheduling, outlines the major obstacles to performance in current task scheduling schemes, and offers solutions which for the most part involve different work stealing scheduling techniques.

When designing a scheduler one of the most important functions of the scheduler is the work partitioner. For any given length of work N, a mathematical function must compute the sizes of the individual chunks to schedule to the work queue. The most simple implementation of a scheduler is the static scheduler, this will simply divide the total work units evenly into the number of processing units and result in one task per processing unit. This results in perfect data locality as data stays exactly where it is at first assigned, however this method is not suited for unbalanced loads as there is no balancing mechanism when one processing unit completes a task before another. In the High Performance Computing field, many other techniques have been researched to provide balance between these three dimensions to create a versatile scheduler.

Research on work scheduling techniques can be found in the context of loop scheduling techniques and task scheduling techniques. The main difference being that in a task-based technique, any task can depend on any number of other tasks, while in a loop-based technique the current loop iterations can only depend on previous loop iterations. The taxonomy in [33] provides an overview of the current task-based scheduling techniques. Several task-based scheduling techniques are compared in various dimensions including the distribution of the data, fault tolerance, and the memory architecture of the hardware. What is noteworthy from this comparison is that all task-based technique surveyed support work stealing. The main differences between these task-based schedulers fall into the following categories listed below.

3.1 Data Locality

A simple work stealing scheduler allocates a separate queue for each processing unit. When a queue is empty, additional work is stolen from other queues. In order to optimize the data locality of the input data that this work is performed on, strategies are used to increase the likelihood that work is executed where the data is located. In [1] a mailbox concept is used, where each process has a FIFO queue of threads that have affinity to the current process. This mailbox concept may have had an influence on modern tasking schedulers, also use a FIFO queue for storing tasks, which in this context are effectively an abstraction of how software threads were used at the time. The mailbox in this case is similar to double-ended queue used in modern work stealing schedulers, which act as a standard FIFO queue from the perspective of the Worker, but also allow for other workers to steal tasks from the front of the queue.

The simplest way to improve data locality is it reuse tasks on the same core as much as possible on the same core. This idea was implemented in [17] with a new scheduler named constrained locality-sensitivity, or shortened cla. This scheduler supports multiple queues, still has one shared queue, and has an adjustable parameter which can be set from 0.0 to 1.0 which determines the probability that a task will be chosen according to it's origin domain. The adjustable parameter allows the user to tune the schedulers load balance and data locality.

Distributed-memory systems are understood to cause a memory access overhead whenever remote memory is access, however this cost also exists on shared-memory systems with non-uniform memory access. This cost can be measured and is reported in [9] as both latency and bandwidth measurements. To achieve this, the authors create a tool which reads OMPT trace files and creates a SimGrid simulation of the tasks being executed while taking into account task dependencies, data locality, and memory effects. SimGrid, which is a framework for simulating application running on a network, proved to be suited for the task as the estimation for the bandwidth available on over a Intel Ultra Path Interconnect system was estimated to be 45 GB/s, which is close to the 41.6 GB/s (or respectively 62.4 GB/s if using 3 UPI links) specification from Intel for this interconnect.

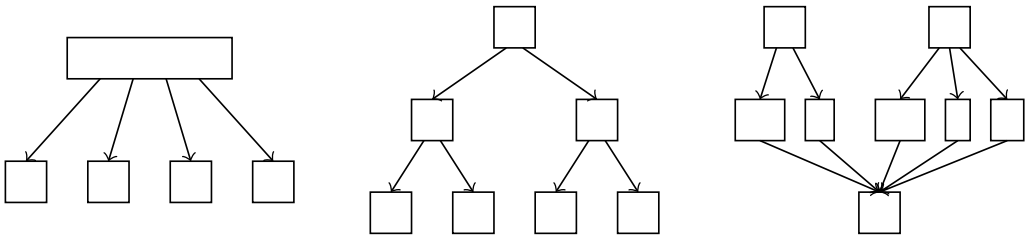
3.2 Task Granularity

When designing a task scheduler, determining the chunk size of a task is one of the most important decisions to make. Task granularity, also referred to as chunk size, is the deciding factor when optimizing between load imbalance and scheduling overhead. When the chunk size is too small, the overhead from all the additional context switches and lock contention rises. On the other hand, when the chunk size is too large, computing power may needlessly sit idle due to load imbalance near the end of a program's execution. In [2] comprehensive experiments on program execution time are conducted using several different task granularity values for several applications using the Cilk runtime library. They also propose a solution to this problem is proposed in which batches of tasks that will be executed sequentially anyway are aggregated into larger tasks when appropriate. The authors also make the case for using two schedulers, one local and another remote to handle work-sharing

requests. Another solution is presented in [34] where tasks are split into smaller tasks while the application is running. In [24] Olivier, Porterfield, and Wheeler implemented several of the above queue allocation and work placement schemes in Qthreads and compare the results quantitatively. The evaluation criteria includes serial and parallel execution times, the number of failed steals, number of successful steals, L3 cache misses, bytes read from memory, and the total number of L3 cache misses. After an analysis of the results, this paper makes the case for a Multi-Threaded Shepherd approach where each NUMA domain is assigned a specific worker to steal tasks, with the option to steal a fixed number of tasks at once. The experiments are repeated with chunk sizes varying from 1 to 64. The authors also stress the importance of using schedulers that take into account the system topology, especially when using server chips that have more than 12 cores. Finally, the overhead caused by locking and unlocking queues is emphasized and the potential for an array-based lock-free deque is hinted at.

3.3 Task Dependencies

While work stealing is viable for both shared-memory and distributed-memory systems, the advantages and disadvantages of using work stealing schedulers are most apparent on distributed memory systems. In [35] a hierarchical system is proposed that takes advantage of the improved data locality of work stealing on the intra-node and inter-node level, while using a global scheduler to handle the victim selection for inter-node task stealing. This removes relatively large cost of failed steals, which are particularly apparent on distributed memory systems. As a followup to this paper, [36] then adds six algorithms to determine how many tasks to steal for a given steal request. Both [35] and [36] also identify three patterns in task parallelism, which are flat parallelism (also referred to as iterative parallelism), recursive parallelism, and irregular parallelism.



(a) Iterative (b) Recursive (c) Irregular

Figure 3.1: Types of Parallelism, adapted from [36]

In contrast to the previous practical evaluation of task stealing, Sonenberg et al. evaluate task stealing mathematically in [32] and found that the decision whether to steal parent or child tasks first can have a large impact on performance. Both parent-stealing and child-stealing were simulated in applications that have various load and probe rates. Child job stealing was found to perform better when there is high load and a low probe rate, while parent job stealing was better with low loads and high probe rates.

When designing a work stealing scheduler, it may seem logical to have idle processing units steal work from other queues in a greedy fashion. In [12], Halpern points out "In

applications where processor affinity is important, a non-greedy scheduler can perform better than a greedy scheduler." This is due to data locality effects on NUMA architectures. Since the memory access times can vary based on which processing unit the code is executed on, and the queue that a task is on is generally the best place for it to be executed, there are situations where the program execution would be faster if said task is not stolen, but rather executed locally, even if it would result in another processing unit being idle. Given that a task trace profile shows load imbalance but does not show additional execution time due to cache misses or suboptimal memory access, this may not seem logical at first glance.

Since the addition of promises and futures in C++11, there has been research on incorporating futures into tasking schedulers. Both [29, 31] provide examples of where wrapping tasks into promises and futures can provide advantages of traditional tasking. An especially interesting use-case was found in the context of blocking system calls.

Earlier research on scheduling techniques refer to computation units as threads [3], in this case referring to software threads in a multi-threaded application. To avoid confusion between software threads and hardware threads, modern research in this area refers to units of work as tasks. A collection of software threads in a waiting list was often referred to as a thread pool, however when referring to task-based scheduling a queue is usually used. While software threads and tasks are conceptually different, the scheduling techniques can be transferred seamlessly. A similar concept is explained in [5] with a focus on tasking and work stealing.

The main compromise made between work-sharing (central queue) and work-stealing (multiple queue) techniques is the balance between data locality being preserved and load imbalance. A technique created to maximize both of these dimensions is Hierarchical Work Stealing. In hierarchical work stealing each task is assigned a "level" in a tree and a threshold is set to limit task stealing. Tasks with a level below the threshold are considered global tasks, while tasks with a level above the threshold are local and cannot be stolen [28].

When designing a work stealing scheduler another important aspect to reflect is the victim selection algorithm. When a worker is idle and executes a steal, there are usually multiple task queues that can be stolen from. When deciding which queue to execute the steal on, there are several factors that can be taken into account including the freshness of the task, the memory access time of where the queue (and the respective task data) is located, and the architecture of the system. In [25] benchmarks are performed using random victim selection and random victim selection with a skewed distribution. The impact of search time and failed steals, as well as the impact of chunk sizes are also discussed.

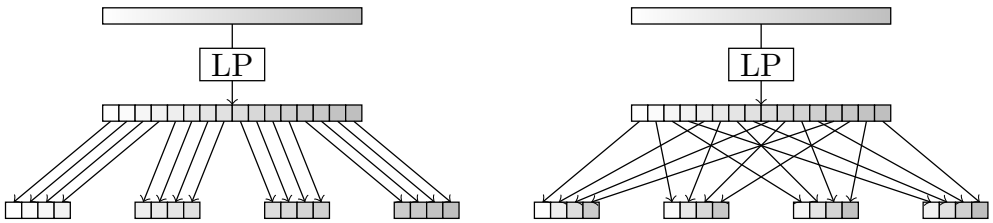
4

Methodology

In order to improve the performance of vectorized data analytics pipelines in DAPHNE, this Thesis hypothesises changes to the work partitioning, work assignment, and task execution order in order to balance scheduling overhead, data locality, and load imbalance. In the load partitioning section, the distribution of

4.1 Load Partitioning

Since the load partitioning object in DAPHNE is takes one input size and outputs chunk sizes for chunks that are intended to be enqueued into a single queue. The architecture must be adjusted in order to support multiple queues. One solution would be to assign each chunk to a respective queue in a block fashion given where the starting split would land relative to the total input. Another solution would be to generate chunks and distribute them into separate queues in a cyclic fashion.



(a) Block Distribution (b) Cyclic Distribution

Figure 4.1: Block and Cyclic distribution, adapted from [4]

In this case, a block distribution would result in a better data affinity as consecutive chunks of data would be accessed in order by the same worker. There is however a downside that arises when using a scheduler that results in changing chunk sizes. When a scheduler starts allocating large or small chunks at the beginning of the programs execution, those chunks would be unevenly allocated to the first workers in line, thus causing an asymmetrical distribution of work which can result in a load imbalance at the end of the execution. Especially when using schedulers with large chunk sizes, this load imbalance can have a significant impact on the applications performance. In order to counteract this, the input

work can first be divided into even chunks, then a separate load partitioning object can be created for each of these chunks separately. This results in chunks of different size while still having symmetrical queues.

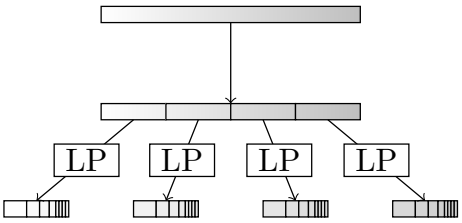


Figure 4.2: Tiled Partitioning

4.2 Scheduling Schemes

A crucial component of any scheduler is the scheme that calculates the chunk sizes that the work is partitioned into. There are varying strategies when it comes to work partitioning ranging from starting with small chunks, ending with small chunks, partitioning in batches, probabilistic, and performance based heuristics. In this Thesis 11 scheduling schemes that are implemented in DAPHNE will be analysed. Plots with visualizations of the below mentioned scheduling schemes are shown in Figure 4.3 and all equations are adapted from [10].

Table 4.1: Table of Symbols

Symbol	Description
K	Output chunk size
N	Total size of input data
P	Number of Workers
i	Scheduling iteration
R_i	Size of data that has not been partitioned at step i
T_{min}	Minimum task execution time
T_{max}	Maximum task execution time

4.2.1 Static

Using a static scheduler is the simplest way to distribute work in a multithreaded application. The total work to be done is simply divide evenly into among each worker into a single task for each worker. This technique results in optimal data locality, as the worker that executes each task is deterministic, assuming that the tasks are distributed to the same workers each time the program is executed. The downside of the static technique is that it results in the highest average load imbalance from all the techniques covered in this Thesis. Once a worker completes its task, is simply remains idle until all other workers complete their work and reach the barrier. Even if the runtime system supports work stealing, there would be no tasks to steal as all the tasks that are created would by either executing with another worker or already completed as well. The chunk size S is calculated below by simply dividing the input iterations N among the workers P.

$$K_i^{Static} = \frac{N}{P} \tag{4.1}$$

4.2.2 Self-Scheduling

A possible solution to the load imbalance problem that a static scheduler has is to create tasks with a size of one, meaning with the smallest indivisible unit of work that the input data allows. Throughout this Thesis, **S**elf-**S**cheduling refers to any method that results in more than one task per worker, as this allows for work stealing, which means that the works are effectively scheduling themselves by stealing remaining tasks. This technique would theoretically result in the best possible load balance, since upon any load imbalance a worker can steal very fine-grained units of work that have not been executed yet from other workers. The downside of this method is that tasks are not executed on the worker that the task was originally assigned to. Especially on NUMA systems, this can result in performance degradation since the increased memory access times result in the same task taking longer than it would have if it here executed by the originally assigned worker. Logically, one would assume that a task being executed on another worker is better than the task not being executed at all until the assigned worker becomes available again, however there are situations were the original worker would have quickly become available again and the stealing worker being idle would have been worth the time penalty of executing the task on a different worker. Since it is costly to determine how long a task will take to execute without executing it, this is very difficult to predict ahead of time. This work partitioning technique is implemented by simply settings the size of each chunk to 1.

$$K_i^{SS} = 1 \tag{4.2}$$

Another downside to self-scheduling is the higher scheduling overhead resulting from the multithreading wrapper having the create the task objects, the victim selection stage during task stealing, and the excess context switches from a worker switching between executing tasks and stealing tasks.

4.2.3 Guided Self-Scheduling

A reasonable compromise to the problems mentioned above would be the compromise and create multiple tasks with varying chunk sizes. **G**uided **S**elf-**S**cheduling is one of the oldest and most researched techniques to pick chunk sizes in the HPC industry. Since the issue of load imbalance only arises once a worker has completed their assigned tasks, it can be stated that there is no load imbalance until the first task in the application's execution is completed. On this basis larger tasks can be created at the beginning of the task queue in order to maximize data locality and minimize overhead, then later on in the queue the task can become more fine-grained to optimize load imbalance. The chunk size S is calculated by dividing the remaining iterations R_i by the numbers of workers P. [27]

$$K_i^{GSS} = \lceil \frac{R_i}{P} \rceil \tag{4.3}$$

4.2.4 Trapezoid Self-Scheduling

Similar to GSS, **T**rapezoid **S**elf-**S**cheduling (TSS) assigns decreasing chunk sizes however in this case the first chunk is half the size of that of GSS. The goal of TSS is similar to GSS however the computational cost of computing each iteration is lower which results in a lower scheduling overhead.

$$K_i^{TSS} = K_{i-1}^{TSS} - \lfloor \frac{K_0^{TSS} - K_{S-1}^{TSS}}{S-1} \rfloor \tag{4.4}$$

$$S = \frac{2 \times N}{K_0^{TSS} + K_{S-1}^{TSS}} \tag{4.5}$$

$$K_0^{TSS} = \lceil \frac{N}{2 \times P} \rceil, K_{S-1}^{TSS} = 1 \tag{4.6}$$

4.2.5 Factoring

While GSS does generate decreasing chunk sizes, the consistently decreasing chunk size results in different chunk sizes for different workers. In order to create more symmetrical tasks among the workers, the factoring [13] scheme generates chunk sizes in batches. This way if there are P workers, there will be P tasks generated with the same size every batch. this consistent chunk size within the same batch of tasks removes a large source of load imbalance between workers as opposed to a scheduling scheme that does not use batches. In this Thesis, the FAC2 scheme is used, which results in chunks with half the size of factoring.

$$K_i^{FAC2} = \lceil \frac{R_i}{2 \times P} \rceil if i mod P = 0 \tag{4.7}$$

$$K_0^{FAC2} = K_{i-1}^{FAC2} otherwise \tag{4.8}$$

$$R_i = N - \Sigma_{j=0}^{i-1} k_j^{FAC2} \tag{4.9}$$

4.2.6 Trapezoid Factoring Self-Scheduling

In order to benefit from calculating batches of equal sizes as seen in factoring, but at the same have a consistently decreasing chunk size, **T**rapezoid **F**actoring **S**elf-**S**cheduling (TFSS) combines both the concepts of factoring and trapezoidal self-scheduling to result in chunk sizes in batches that also decrease at a constant rate.

$$K_i^{TFSS} = \frac{\Sigma_{j=i}^{i+P} K_{j-1}^{TSS}}{P} if i mod P = 0 \tag{4.10}$$

$$K_i^{TFSS} = K_{i-1}^{TSS} otherwise \tag{4.11}$$

4.2.7 Fixed Increase Self-Scheduling

All of the scheduling schemes mentioned up to this point operate on the principle of decreasing chunk sizes. When using **F**ixed **I**ncrease **S**elf-**S**cheduling the opposite concept is applied. Chunk sizes start at a relatively small point and increase as the program executes. The goal of the increasing chunk size is to avoid the large scheduling overhead at the end of a programs execution due to the excessively small chunk sizes generated when using a

decreasing chunk size. This scheduling scheme is designed for distributed-memory systems which suffer from greater scheduling overhead due to the fact that task stealing happens over the network. Even though the scheme is designed with distributed-memory systems in mind, it is included in this Thesis in order uncover a possible benefit on NUMA systems.

$$K_i^{FISS} = K_{i-1}^{FISS} + \lceil \frac{2 \times N \times (1 - \frac{B}{2+B})}{P \times B \times (B-1)} \rceil, where \tag{4.12}$$

$$K_0^{FISS} = \frac{N}{(2+B) \times P} \tag{4.13}$$

4.2.8 Variable Increase Self-Scheduling

Similar to how factoring decreases the chunk size in batches, VISS is a scheduler with increasing chunk sizes that schedules chunks in batches. This combines the advantage of more symmetrical queues from factoring, with the potential idea of increasing chunk sizes from FISS.

$$K_i^{VISS} = K_{i-1}^{VISS} + \frac{K_{i-1}^{VISS}}{2} if i mod P = 0 \tag{4.14}$$

$$K_i^{VISS} = K_{i-1}^{VISS} otherwise \tag{4.15}$$

$$K_0^{VISS} = \frac{N}{(2+B) \times P} \tag{4.16}$$

4.2.9 Performance-Based Loop Self-Scheduling

Since DAPHNE runs on many different systems with different characteristics, it is beneficial to also offer scheduling techniques that can dynamically adjust to system and application characteristics. In **P**erformance-based **L**oop **S**elf-scheduling, the work is divided into two parts, with the first being scheduling with a static chunk size, and the second scheduled with GSS. The size of the first part relative to the second is based on a **S**tatic **W**orkload **R**atio, which is unique to each system and is computed beforehand in a separate execution of the application.

$$K_i^{PLS} = \frac{N \times SWR}{P} if R_i > N - (N \times SWR) \tag{4.17}$$

$$K_i^{PLS} = K_i^{GSS} otherwise \tag{4.18}$$

$$SWR = \frac{T_{min}}{T_{max}} \tag{4.19}$$

4.2.10 Probabilistic Self-Scheduling

While all of the previous scheduling schemes have precise input data, there may be utility to using approximations of data such as the number of idle workers. In **P**robabilistic **S**elf-**s**cheduling the average number of idle workers over a given time period is used to determine the chunk size. The calculation for the chunk size used in this case is similar to GSS, however in this case only the number of idle workers is counted, instead of the total workers.

$$K_i^{PSS} = \lceil \frac{R_i}{1.5 \times P_x} \rceil \tag{4.20}$$

4.2.11 Modified Fixed-Size Chunk

The **M**odified **F**ixed-**S**ized **C**hunk scheduler is very simple to implement, as the chunk size only needs to be calculated once and then chunks can easily be scheduled with said chunk size. In MFSC, the a chunk size is calculated where the number of chunks will be equal to the number of chunks generated when using factoring. In the DAPHNE load partitioner, this chunk size is calculated using the formula below, where N is the input size and P is the number of Workers.

$$S = \lceil \frac{ln(2) \times (N + P - 1)}{P \times ln(\frac{N+P-1}{P})} \rceil \tag{4.21}$$

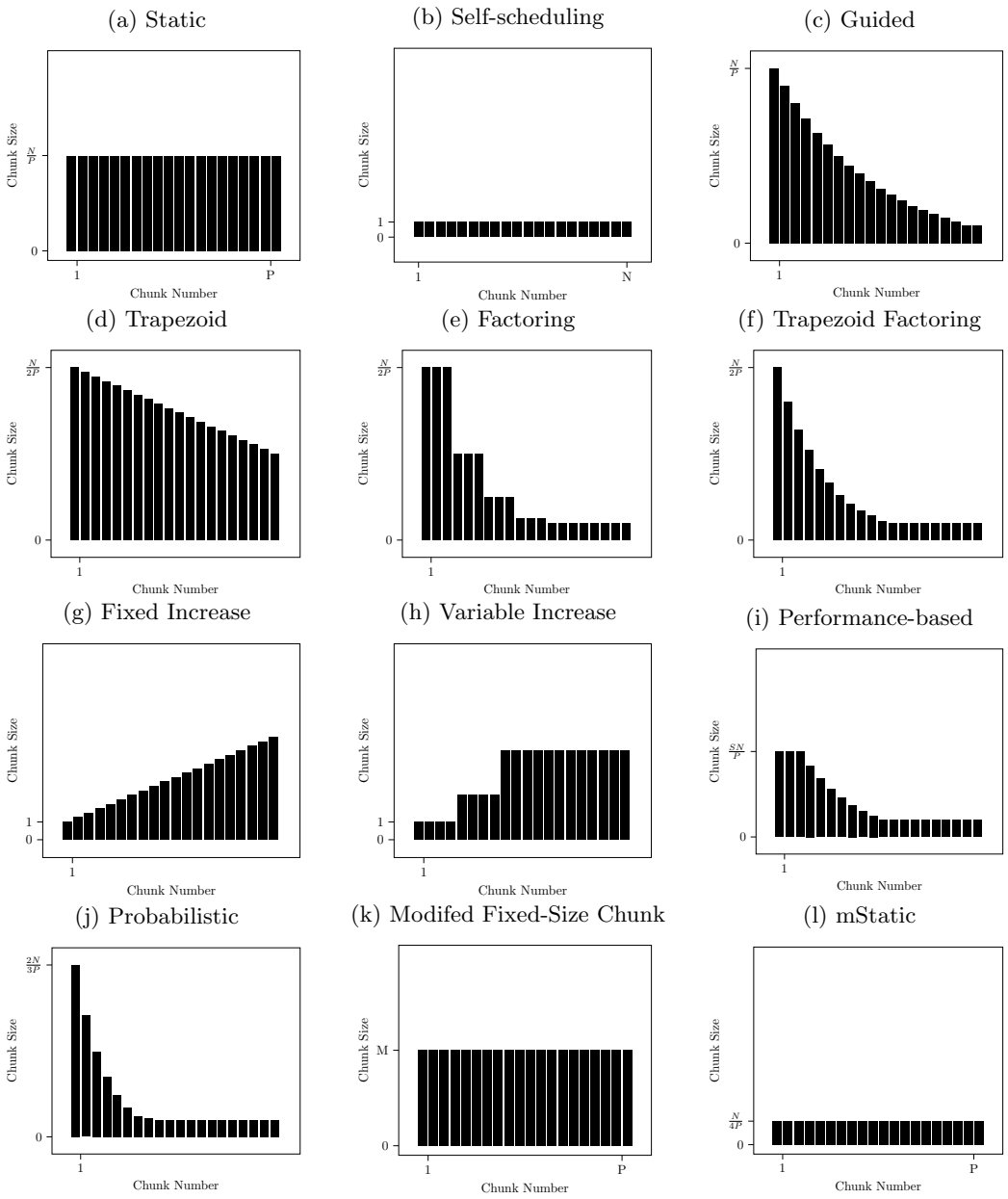


Figure 4.3: Visual overview of scheduling techniques and the respective chunk sizes (Chunk sizes approximated for visual effect)

4.3 Fused Kernels

In the DAPHNE vectorized execution engine, when the output of one operation is directly input into another operation, the two operations can be fused together and vectorized as if it were a single operation. This improves performance by reducing the overhead needed to accumulate the outputs of one operation and the setup needed for the second operation.

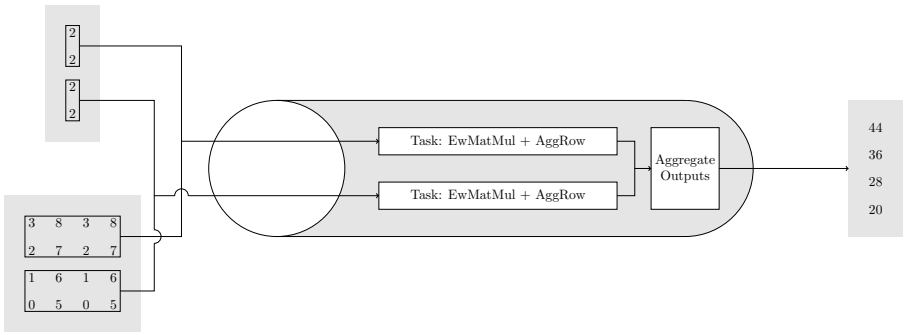


Figure 4.4: Multiple kernels fused and vectorized by row (Chunk size of 2)

4.4 Work-Sharing

A work-sharing scheduler is a scheduler in which a central entity assigns tasks to workers. In practice, this is generally implemented by a single-queue with a shared lock that all workers and dequeue tasks from. This is generally the simplest shared-memory scheduler to implement as it does not involve any work-stealing or victim selection logic. The downside of work-sharing scheduler is that it is more difficult to implement locality-aware schedulers.

4.5 System Architecture

Historically computing systems have been improving via an increase processor clock speeds. According to Moore's Law, these clock speed were expected to increase linearly with respect to time. Since the time it takes for an electric signal to reach it's destination is limited by the speed of light, there is a limit to how large a CPU can be built before delays are introduced in the internal connections. Due to the fact that higher clock rates generally require higher voltages, and there is a limit to how much thermal density can be packed into a limited space, the improvement in clock rate has largely plateaued. In order to improve performance, applications must now be parallelized and executed on shared-memory multi-processor systems and distributed-memory multi-node systems. While a multi-processor system has the advantage of offering more computing power without the need to coordinate the sending of data in memory, the fact that the memory is not longer accessed uniformly creates new challenges that scheduling systems must take into account. Systems that have more than one path to access memory are referred to as **N**on-**U**niform **M**emory **A**ccess (NUMA) systems. An example of a possible NUMA system architecture is depicted in Figure 4.5 below.

4.5.1 NUMA System Topology

Both test systems are configured with Intel CPUs from the Broadwell and Casecade Lake family, respectively. The system topologies are similar, apart from the difference in generation. While the Broadwell architecture uses a "ring" bus to access the L3 cache and memory interfaces, the Casecade Lake family has a "mesh" architecture which allows for a quicker route with less hops to get to the destination cache line or memory address. In addition, the Cascade Lake architecture has three interconnect links as opposed to two in the Broadwell architecture. These factors would theoretically result in a larger performance improvement in the Per-Core results for the Cascade lake test system over the Broadwell test system.

Even though these are shared-memory systems in which every memory page is available to every processor, the access times are not uniform. The latency of a memory access to a remote memory node is generally considered to be twice as high as a local memory node access. In interesting effect that occurs due to the fact that the interconnect bus is separate from the memory controller for local memory is that there are situations where accessing local memory and remote memory at the same time results in a higher memory bandwidth than only accessing local memory, since the bandwidth of the local memory controller and the interconnect link can both be fully saturated simultaneously. However, since all the tests performed in this Thesis utilize all of the available CPUs on the system, this effect would not become apparent in this Thesis.

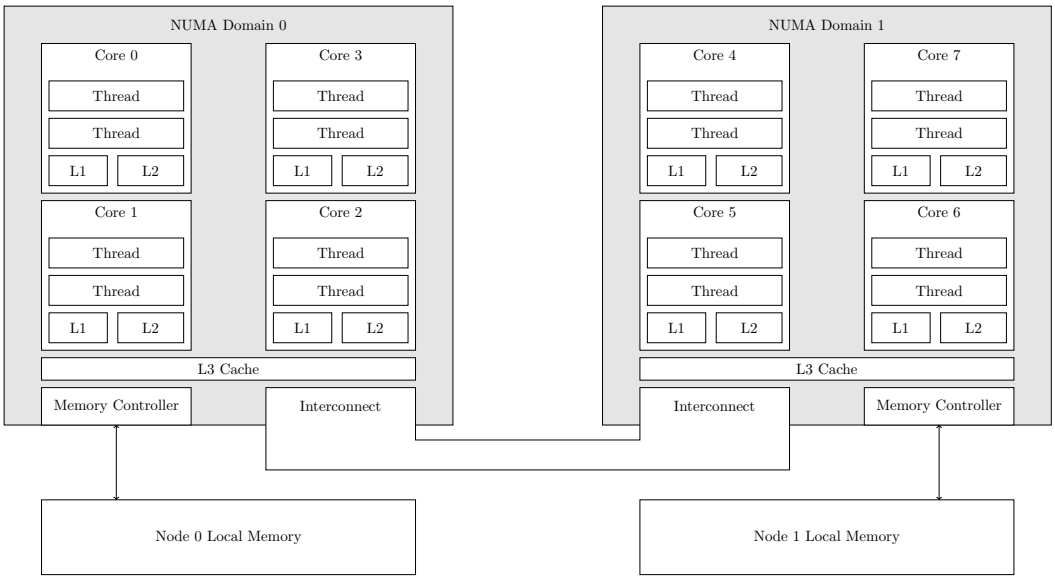


Figure 4.5: System Topology

Memory pages that are requested and reside on the local memory connected to the requesting processor are referred to as local requests. Memory pages requested from any other memory node are referred to as remote requests. Remote requests are more costly in terms of performance since the requests but first be directed to the respective CPU, and the other CPU must then handle that memory access request. Essentially, the request must be handling anyway, but if it is a remote request then the inter-socket communication

overhead is adds to the access time in addition to the standard memory access time. The exact amount of this additional overhead can be interred by the memory interconnects speed and latency. The two infrastructures used in this Thesis, Intel Broadwell and Intel Cascade Lake, which use Intel QuickPath [16] and Intel Ultra Path [21] interconnects, are listed in Table 4.1 with their respective bandwidth specifications.

Table 4.2: Test hardware memory interconnect specifications

Test System	Broadwell	Cascade Lake
Interconnect	Intel QuickPath Interconnect	Intel Ultra Path Interconnect
Clock Rate	4 GHz	5.8 GHz
Transfers/Hz	2	
Transfer Rate	8 GT/s [15]	10.4 GT/s [22]
Link Width in Bits	20 [16]	
Interconnect Links	2 [15]	
Payload/Link width	16/20 [16]	
Bits/Byte	1/8	
Total Bandwidth	32 GB/s	41.6 GB/s

These values should provide an upper-bound on the performance improvement that can be achieved by optimizing the data locality of tasks on the above mentioned shared-memory systems. With regards to the bandwidth specifications above, overhead caused by the "Home Snoop" procedure and the fact that an memory line with a minimum size of 64 bytes must be retrieved each time would further reduce the practical link bandwidth, so a lower performance improvement than this upper-bound is to be expected. More complex topologies such as the 2-socket/4-chip AMD Magny Cours [24] are also used in this context, but will not be analysed in this Thesis.

4.5.1.1 First-Touch policy

On a system with non-uniform memory access, the operating system makes decisions on how to best utilize the memory topology available. Possible strategies to maximize the effectiveness of non-uniformly accessed memory is to either ignore it's existence altogether and treat it as a normal uniform memory system, stripe the memory addresses between the NUMA domains so that consecutive data is forced to be distributed evenly among the NUMA nodes, or to manage the memory in a NUMA-aware manner. The Linux operating system uses the first-touch policy to decide where to allocate memory for a virtual address [18]. It is important to note that it is this policy only takes effect when the memory in question is "touched" or written to. Not all programming instructions that are intuitively thought to allocate memory actually touch the page of memory in question upon invocation.

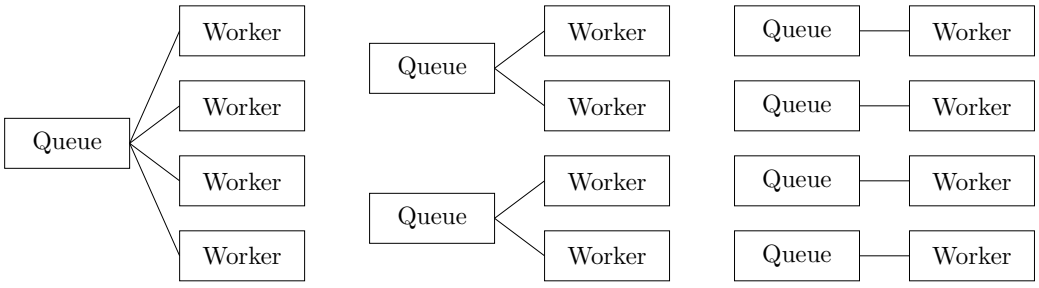
For a single-threaded application this concept is quite simple, as it is clear in which NUMA domain the memory address will be allocated in. However in an application where multiple threads located in different NUMA domains access the same memory address, ambiguity is introduced and it can often not be predicted where the memory will be allocated to before the program executes. In Linux 3.8 there was development on a framework that automatically migrates memory pages from one NUMA node to another when the appropriate opportunity arises [18].

4.5.2 Simultaneous Multithreading

Simultaneous Multithreading, also commonly referred to as hyperthreading by Intel, is a technique that some hardware supports in order to improve performance in certain situations. As clock speeds have reached physical limitations, this technique allows for multiple hardware contexts to be to run in parallel on a single core without having to duplicate all of the hardware on the core. The concept is to duplicate certain elements of processing units, namely the computing elements, while still sharing functional units of the processor cores. This is a clever method of increasing the logical throughput of computing hardware without having to duplicate entire processor cores.

Due to the additional functional units, many small operations can be executed in parallel and it can be beneficial to utilize these additional functional units, as they essentially result in a latency-hiding technique [30], however the increased usage of the L2 and L3 cache may outweigh these benefits and result in a performance regression. It can be difficult to determine if utilizing these threads is a net gain or loss in performance before a program is executed, which is why most computing resource providers recommend benchmarking the application in question with both configurations first [11]. Generally applications that suffer from memory stalls, wrong branch predictions, or unbuffered file I/O benefit from simultaneous multithreading. Since implementation of multiple task queues in this Thesis queries the system topology anyway to allow for the prioritized work stealing, the option to utilize simultaneous multithreading is exposed to the user.

4.6 Work-Stealing



(a) Central Queue (b) Per-Group Queues (c) Per-Core Queues

Figure 4.6: Queue Allocation Schemes

4.6.1 Serializing Work Stealing

Dequeue and enqueue from bottom of deque, steal from top of other deques.

Since a task that is added to a queue is fresh in in the cache of the worker that added it to the queue, it would be advantageous for that same worker to dequeue and execute it. For this reason, worker enqueue and deque to the bottom of the double-ended queues. Since this advantage does not apply to other workers due to their cache being separate, other workers will dequeue tasks from the top of the queue in order to leave more fresh tasks for the enqueueing worker that can claim an advantage from them.

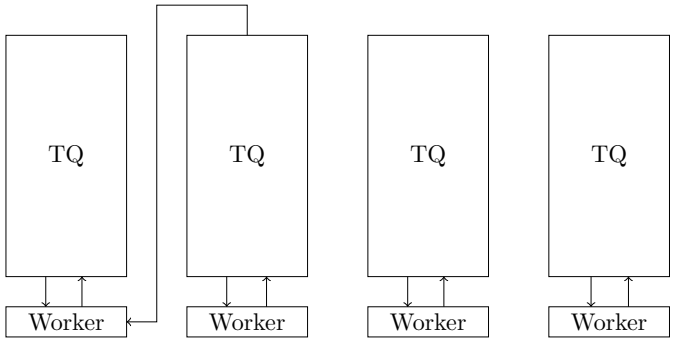


Figure 4.7: Serializing Work Stealing

4.6.2 Eager Binary Splitting

When partitioning work using a tasking scheduler, the load partitioner must determine the upper and lower bound indexes of the input data that each task will represent. In serializing work stealing, one large task with a upper and lower bound encompassing all of the input data is created, and workers "split" off a single iteration at a time when task stealing. Instead of splitting one unit of work at a time, Eager Binary Splitting splits the stolen task into two equal halves, with one half remaining as a task on the victim's deque, and the other half executed by the initiator as a new task.

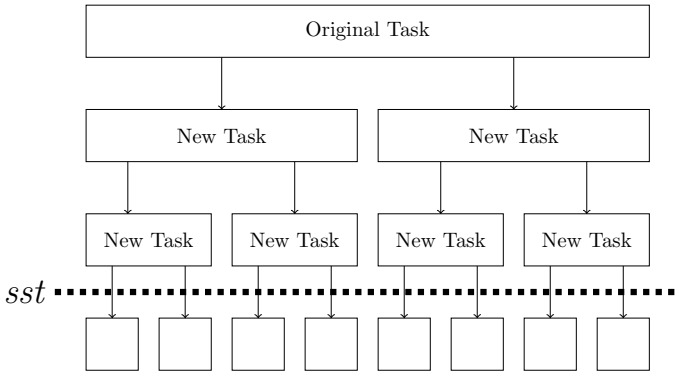


Figure 4.8: Eager Binary Splitting

In order to prevent task from being continuously split until the task size is 1, effectively resulting in serializing work stealing, a *stop splitting threashold* is set which sets a minimum task size after which tasks will no longer be split [34]. In addition to the static partitioner method with a set *sst* there is also an auto partitioner concept in which the *sst* is dynamic depending on the number of workers.

Eager Binary Splitting is an effective concept to quickly partition load among any number of queues in a work stealing system. However, since in DAPHNE the load partitioning is centralized by the Multi-threading wrapper, the input work can directly be split among the queues without the need for stealing. In addition, since the number of queues is known by the multi-threading wrapper at the time of load partitioning, the work can directly be split according to the scheduling scheme selected by the user in the program arguments. When using the pre-partitioning argument the segments can then be split again, all within the wrapper.

4.6.3 Hierarchical Work Stealing

While Serializing Work Stealing and Eager Binary Splitting determine how many tasks to steal, or respectively what to do with a task while it is stolen, it is also important to take into account the architecture of the system that the parallel application is running on. Whether it is the network structure on a distributed memory system, or the memory architecture of a NUMA system, the additional time that it takes a task to run due to the system hierarchy can be minimized when deciding victim queue to steal from. One implementation of a hierarchical system is Multi-Threaded Shepherds, where one workers from every domain is a selected to be the Shepherd who is responsible for stealing tasks for all workers in its domain.

4.6.4 Victim Selection

Work stealing schedulers have to make a decision on which victim queue to steal a task from. Research in this area generally points to two strategies, sequential and random victim selection. Although many papers make the case for sequential or random victim selection in distributed memory environments, the same arguments can also be made for a shared memory environments.

The logic that a worker follows to execute tasks without work stealing is very simple. The worker simply dequeues a task and executes it, if the task that is returned from the queue is blank (referred to as EOFTask in this Thesis and in the DAPHNE source code) then that worker is completed and can join the barrier. The logic workflows described in this Thesis are in part inspired by the task stealing logic from the LLVM OpenMP Runtime System [20].

Another decision that work-stealing schedulers make is whether to stick with the same victim after a successful steal, or to select a new victim each time, regardless of whether the previous steal was successful or not. In (citation) the implementation changed the victim each time in order in attempt to balance the load more evenly, however the authors noted there a potential where sticking to the same victim after a successful steal would improve performance.

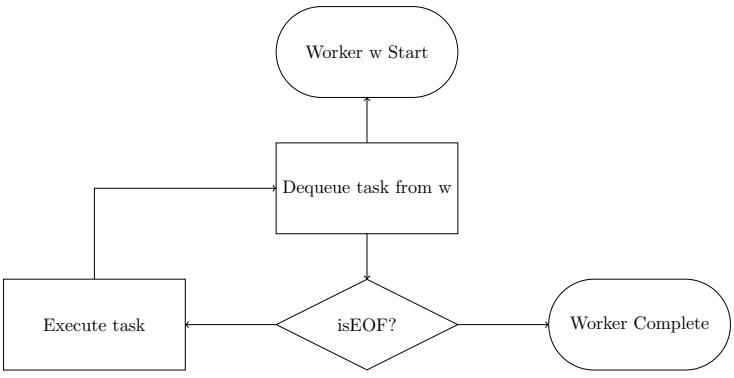


Figure 4.9: Worker task execution logic without work-stealing

4.6.4.1 Sequential

In sequential victim selection a worker searches for victim queues in a round-robin fashion starting at their own position in the system topology. An advantage of this method is that queues that are close to the initiator, or at least numerically above the initiator, will naturally have a priority to be chosen as a victim queue. In [25] victims are skipped regardless of whether the steal is successful or not in order to provide more scheduling deterministic behavior. In the implementation for this Thesis, victim's are only skipped after a failed steal, as it simplifies the implementation of the termination algorithm.

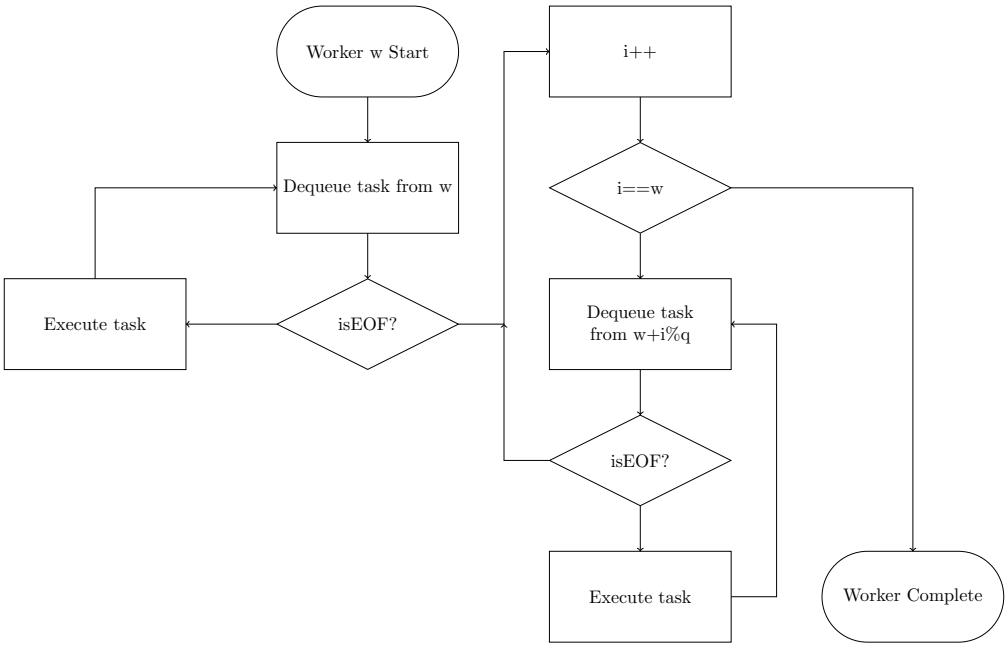


Figure 4.10: Worker task execution logic with sequential victim selection

In this implementation, the termination is simply after one pass has been made through all of the possible queues. This simple algorithm does not require any writes to memory to keep track of which queues are closed or still open, other than the incremented variable i.

4.6.4.2 Sequential Prioritized

In order to preserve data locality between NUMA domains when possible, and to prevent inter-socket communication, queues from the same domain can be prioritized [6]. This makes the work stealing logic slightly more complicated, as a comparison needs to be made to check whether a potential victim queue is in the same domain first, however this comparison does not require taking the lock of the potential victim's queue so that extra overhead added should be minimal. In Figure 4.11 this is done using a simple incrementing variable i. When i reached its original position again, a steal has been unsuccessful on every available queue, so the worker is complete and can join the barrier.

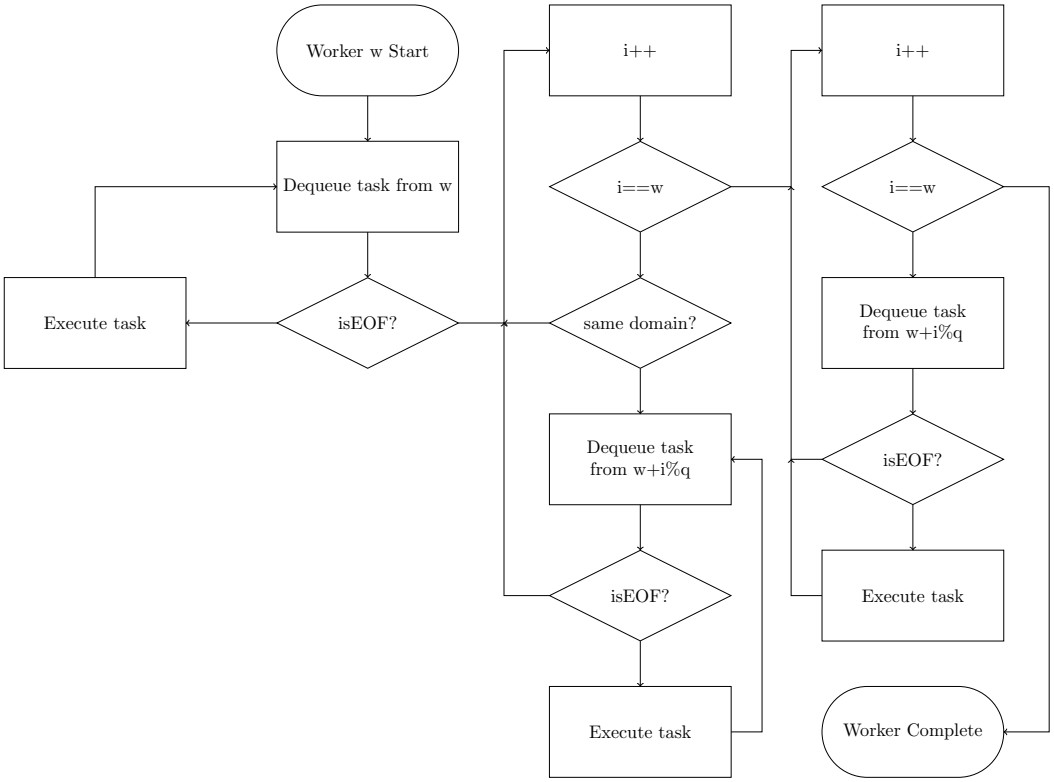


Figure 4.11: Worker task execution logic with prioritized sequential victim selection

4.6.4.3 Random

While a sequential victim selection algorithm is already very efficient, there is research that indicates that a random victim selection can improve performance even further. In [25] the random victim selection algorithm is compared to a sequential reference implementation and a large decrease in the number of failed steal attempts was observed, in some cases this also translated to an improvement in performance. This can be explained by the sequential algorithm resulting in a large number of failed steal attempts while traversing from the initiating worker to the victim that results in a successful steal. If the input data results in load imbalance that is clustered in regions, then a random victim selection will arrive at the a victim in the optimal region quicker.

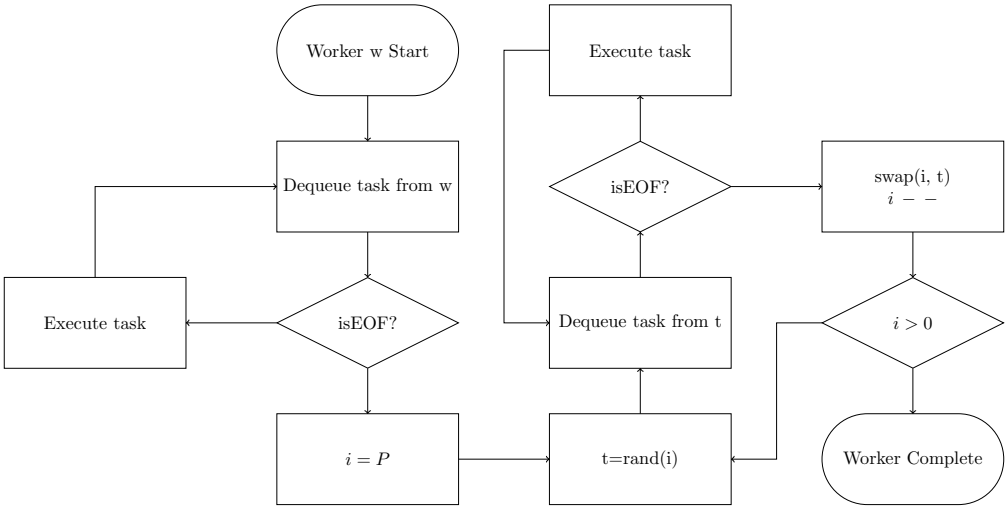


Figure 4.12: Worker task execution logic with random victim selection

As seen in Figure 4.12, the downside of using a random victim selection algorithm is the additional overhead caused by generating a random number each time a steal attempt is made, and also the requirement to keep a register of which queues have been completed. This latter overhead can be minimized by keeping the list local to that worker's memory, however it still results in additional reads and writes to memory.

4.6.4.4 Random Prioritized

Similar to the "Random" implementation, the Random Prioritized implementation also requires keeping a register of which CPU cores have empty and closed queues and which ones still have the possibility of having tasks in their queue. However, in this case two registered must be kept, so in the worst-case the overhead caused by this bookkeeping could double in comparison to the standard random implementation. In practice the cost of keeping these registers on a shared-memory system should be minimal.

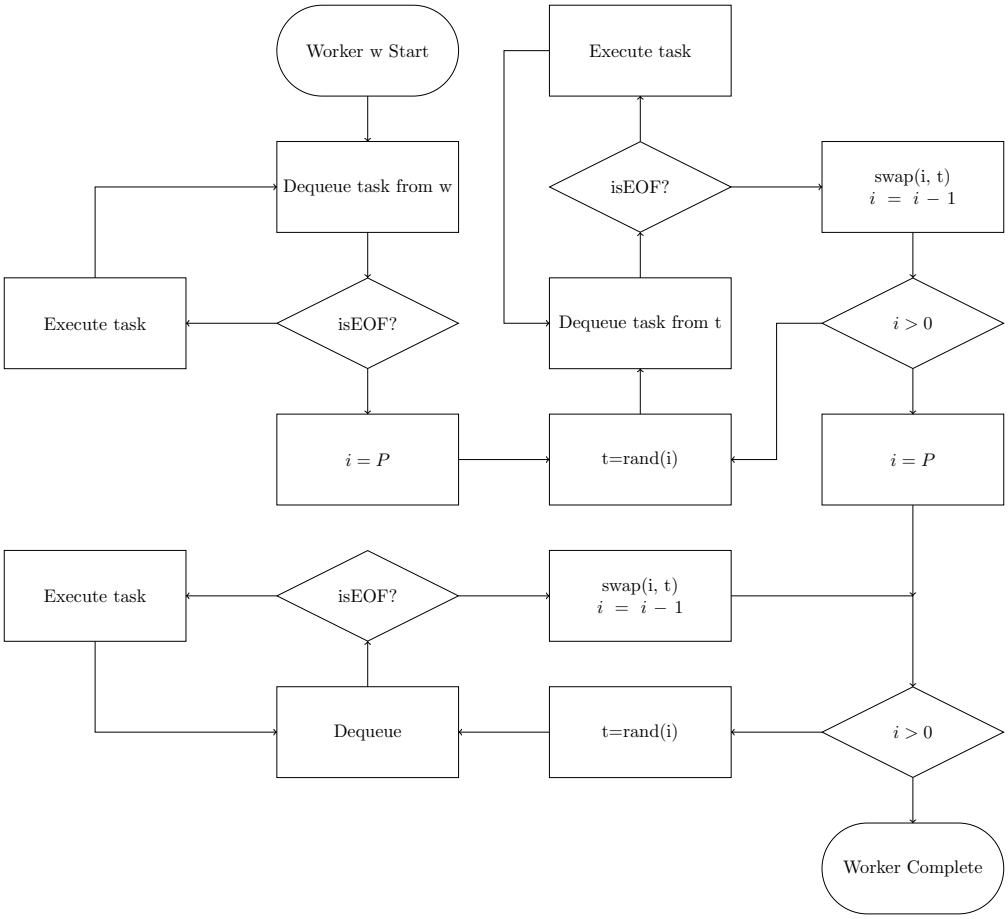


Figure 4.13: Worker task execution logic with prioritized random victim selection

4.6.5 Multi-Threaded Shepherds

While a simple work-sharing architecture with a single queue provides for optimal load balancing, and a work-stealing architecture with multiple queues provides for optimal data locality, a compromise can be made between the two by grouping workers utilizing work-sharing inside the groups while employing work-stealing between the groups. Previously in this Thesis there are examples of work-stealing with one queue per CPU socket, however in the the previous examples any worker was able to steal from any other worker. To optimize communication between the groups, one worker from each group can be assigned as the Foreman (Also referred to in the literature as Shepherds) of that group. The foreman is responsible for stealing tasks and enqueueing them onto the queue for that group. This limits the work stealing activities to only one worker per group and prevents unnecessary communication overhead.

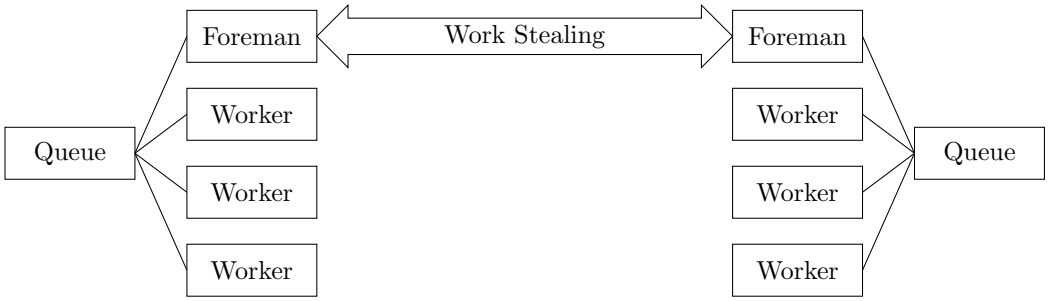


Figure 4.14: Multi-threaded Shepherds Architecture

The downside of this hybrid work-sharing and work-stealing implementation, which is a type of Hierarchical, is that it adds a new layer of overhead to the application, as tasks now have to be dequeued from the victim queue, enqueued into the initiating foreman's queue, then dequeued and executed by the worker. The new addition overhead can be minimized by stealing a vector of multiple tasks. The optimal size of this vector is found in [6, p. 35] to be half of the available tasks on the victim's queue in the context of distributed memory systems. Intuitively, on shared memory systems the optimal amount would be slightly lower. For the purposes of this Thesis, a value of one-half is used in the multi-threaded shepherds experiments.

4.7 Multi-threading Wrapper

When one or multiple operators in the DAPHNE DSL can be vectorized, they pass through one or multiple "lowering passes" and become instructions in the DAPHNE IR. Here the corresponding kernels are found and function pointers to the respective apply functions are created. The function pointers, along with the corresponding splits (rows, columns, or scalars) are passed to the Multi-threading wrapper. The multi-threading wrapper is an abstraction that distributes the work without regard to what functions are called, how the data is split, or how many kernels are called in one vectorized pipeline.

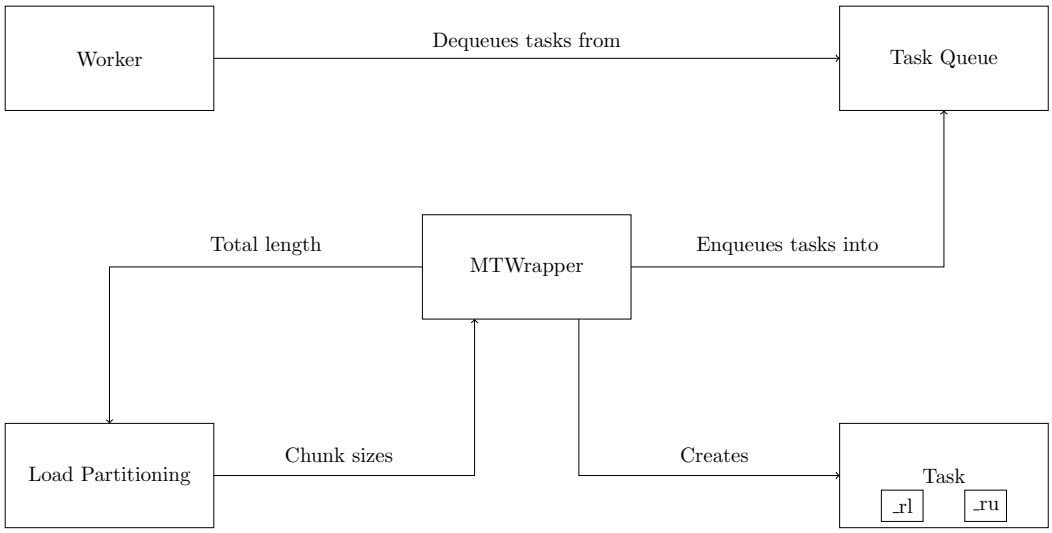


Figure 4.15: Architecture of the DAPHNE Multi-threading wrapper

The multi-threading wrapper is the central point that handles all the objects required to parallelize a vectorized pipeline. When it is called, the function pointers, resulting variable pointers, input data pointers, number of inputs, and number of outputs are passed to the multi-threading wrapper. It will then create a load partitioning object which will generate the chunk sizes used for the lower and upper bound values of tasks. Then one or multiple queues are created and workers are started with their respective queues. One the workers are ready tasks are created and added to the respective queues which the workers can deque from.

4.8 Vectorized Engine Trace Files

In order to diagnose slow performance in the vectorized execution engine, I added hooks in the code at the pointers where both the Multithreaded wrapper and tasks start and end. These hooks record timestamps and save them to a vector in the DAPHNE context, which can be printed to a file at the end of the program's execution. When these timestamps are plotted against time, a trace plot can be generated such as the one below. In this plot, the dark grey boxes in the background show instances of the Multithreading Wrapper, while the white rectangles show the executions of individual tasks.

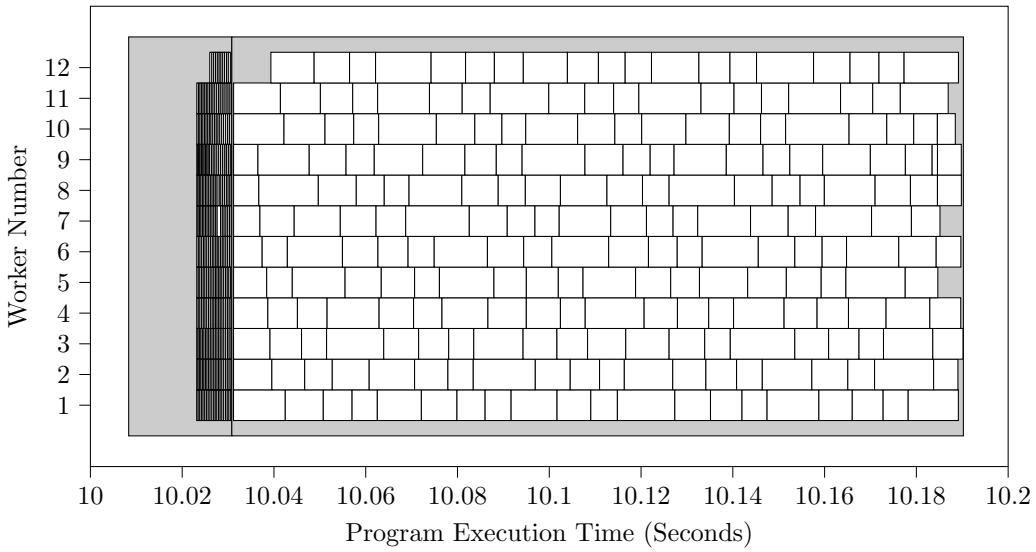


Figure 4.16: Task trace of a multithreaded vectorized pipeline execution in DAPHNE

The task trace plot shows weak points in the runtime scheduling system. For example, the grey area on the right side of the plot show how long workers are idle after executing their tasks, which is the load imbalance. Empty areas in between tasks show the time lost due to scheduling overheads such as context switching. However one aspect that must be recognized when designing a scheduler which is now display in this plot is the time lost due to non-optimal data locality. If a task is executed by an execution unit that does not have the data ready in cache, which another execution unit theoretically would have had the relevant data in cache, the task would take longer, but this type of plot does not make the reason for this longer execution time apparent.

4.9 Design of Factorial Experiments

All experiments were performed on the University of Basel miniHPC cluster on node001 and node027. Jobs were dispatched using Slurm and the execution times of the core algorithm were measured using the now() function within DAPHNE DSL which is printed to standard output which is then saved to the filesystem by Slurm. Each experiment configuration is executed 20 times as a Slurm Arrayjob and the mean result is reported in chapter 5. In order to satisfy various dependencies required by DAPHNE, all experiments are launched inside a Singularity container built from a Ubuntu 22.04 recipe. The standard deviations of the runtimes for each experiment are reported in Appendix A.

Table 4.3: Design of Experiments, Resulting in a Total of 11,000 Experiments

Factors		Values	Properties
Applications		Connected Components algorithm	Amazon product co-purchasing network Amazon0601.txt: scale factor 50 20,169,700 Nodes, 8.26% Sparsity
Scheduler Type		Work Stealing	Any worker can steal from any queue
		Multi-threaded Shepherds	No further configuration
Task Assignment	Queue Layout	Centralized Queue	One queue for the entire node
		Per-group Queues	One queue per NUMA-node
		Per-thread Queues	One queue per processor core
	Victim Selection	Sequential	Workers pick victim queues in sequentially
		Sequential Prioritized	Sequential prioritizing same group
		Random	Workers randomly pick victims
		Random	Random prioritizing same group
Further Arguments		Cyclic Assignment	Tasks assigned to queues cyclically
		Pre-partition	Use two levels of load partitioning
Partitioning Scheme		Static	Entire workload divided evenly into P tasks
		Guided Self-scheduling	Remaining tasks divided by workers
		Trapezoid Self-scheduling	Progressively smaller tasks enqueued
		Factoring (FAC2)	Tasks half the size of Guided self-scheduling
		Trapezoid Factoring Self-scheduling	Chunks half the size of GSS
		Fixed Increase Self-scheduling	Chunks of increasing size
		Variable Increase Self-scheduling	Similar to FISS
		Performance-based Self-scheduling	Uses a workload ratio to compute chunk
		Modified Static	Uses chunks one quarter the size of the Static
		Modified Fixed-Size Chunk	Chunks a fixed size based on an equation
		Probabilistic Self-scheduling	Uses CPU idle time
Computing Nodes		miniHPC-Broadwell	Intel E5-2640 v4 P=20 (2x10), hyperthreading disabled
		miniHPC-CascadeLake	Intel Xeon Gold 6258R P=56 (2x28), hyperthreading disabled
Repetitions		20	

5
Results

For the connected componenets application, the Stanford SNAP Amazon product co-purchasing network dataset is used. This dataset was collected by scraping the "Customers Who Bought This Item Also Bought" feature of the Amazon website. According to the SNAP website, "If a product i is frequently co-purchased with product j, the graph contains a directed edge from i to j" [19]. This dataset contains 403,394 nodes and 3,387,388 edges, resulting in a density (sparsity) of 0.002%. The dataset first parsed using a python script that converts all directional edges into non-directional edges and output the edges in COOFormat, which is compatible with the sparse matrix reader in DAPHNE. In order to increase the execution time of the microbenchmark, a scale factor of 50 was also applied to the source dataset, which resulted in an input matrix of 20,169,700 nodes and 244,340,800 directional edges. This scale factors resulted in program execution made up of approximately 90 seconds of file I/O and 10 seconds of parallel computation.

For each test system, the first experiment is a Work-Stealing setup with either per-Node, per-Group, or per-Core queue allocations. For each of these queue allocation the victim selection options sequential, sequential prioritized, random, and random prioritized are tested. Then, for each of these permutations all 11 available load partitioning options are also tested. Each of these experiments are repeated 20 times and the mean value is reported in the tables below. There is one unified color scale for the Broadwell test system and one unified color scale for the Cascade Lake test system.

These experiments are then repeated with the –pre-partitioning option enabled, which splits the incoming work in to sections for each queue, before applying the load partitioner to created chunks of various sizes. This option is expected to increase data locality which would theoretically reduce the number of cache misses and memory accesses through the socket interconnects. However, this split before the load partitioner is applies also results in smaller chunk sizes for the same experiment with the option disabled, since the input size as seen by the load partitioner is smaller, which resutls in smaller chunk sizes being generated. This additional effect can also affect the resulting runtimes because the smaller chunk sizes could reduce load imbalance in the program's execution. It is difficult to isolate the effects of the data locality and the effects from load imbalance when only the program runtime is available and this must be taken into account when interpreting the pre partitioning results.

5.1 Broadwell

5.1.1 Work-Stealing

		STATIC	GSS	TSS	FAC2	TFSS	FISS	VISS	PLS	MSTATI	MFSC	PSS
Per-Node	Sequential	13.71	13.56	13.35	12.68	12.54	13.92	12.53	13.32	13.1	11.83	12.72
	Sequential Prioritized	13.89	12.96	12.87	12.9	12.52	13.8	12.87	12.97	13.31	11.84	13.09
	Random	14.2	13.24	12.63	12.64	12.2	14.09	12.37	13.15	13.05	11.86	13.25
	Random Prioritized	14.2	13.36	13.58	13.03	12.43	14.03	12.28	13.4	13.61	11.71	12.93
Per-Group	Sequential	13.9	13.75	13.28	12.96	13.14	13.98	13.33	13.22	14.17	13.16	13.27
	Sequential Prioritized	14.24	13.23	13.62	12.93	13.43	14.39	13.51	13.59	14.15	12.76	13.39
	Random	13.63	13.27	13.38	12.92	13.18	14.23	13.08	13.19	14.41	13.27	13.76
	Random Prioritized	14.3	13.54	13.52	13.29	13.18	14.34	13.54	12.82	14.18	13.19	13.49
Per-Core	Sequential	13.71	13.72	13.52	13.16	13.57	13.68	12.1	13.24	13.22	12.4	13.83
	Sequential Prioritized	13.8	13.79	13.04	13.35	13.85	13.75	13.01	13.41	13.42	12.88	13.63
	Random	13.11	13.92	13.65	13.85	13.51	13.28	12.64	13.41	13.09	12.57	13.84
	Random Prioritized	13.58	13.51	13.45	13.04	13.39	13.91	12.17	13.97	13.55	12.25	13.75

11.22 14.41

Figure 5.1: Results from Work-Stealing Experiments: Broadwell

In this experiment all possible permutations of scheduling techniques, queue allocations, and victim selection techniques are measured separately with 20 repetitions each. Static, FISS, and MSTATIC performed the worst while MFSC and VISS performed the best. For input dataset of this size Static was expected to perform the worst since it has the highest load imbalance. The large difference in performance between FISS and VISS is an interesting result as they result in similar patterns in chunk sizes. The number of queues had a greater effect when using some scheduling schemes than on others, with the per-Core option performing better or worse than the other queue allocation schemes depending on the scheduling scheme. The victim selection algorithm did not have as large of an effect as the other factors, which can be explained by the input data not having a significant inherent load imbalance which results in most of the tasks being executed before task stealing takes effect.

5.1.2 Tiling

		STATIC	GSS	TSS	FAC2	TFSS	FISS	VISS	PLS	MSTATI	MFSC	PSS
Per-Node	Sequential	13.26	12.47	12.67	12.52	12.69	13.59	12.1	13.11	13.45	11.39	13.05
	Sequential Prioritized	13.49	12.59	12.53	12.1	12.73	13.63	12.21	13.18	13.45	11.82	12.7
	Random	13.19	12.4	12.62	13.1	12.6	13.83	12.36	12.48	13.41	11.51	13.3
	Random Prioritized	13.33	12.71	12.76	12.62	12.15	13.14	12.43	12.95	13.16	11.85	12.56
Per-Group	Sequential	13.24	11.89	11.63	11.66	11.6	12.5	11.59	12.71	12.32	11.53	12.16
	Sequential Prioritized	13.45	11.97	11.5	11.89	11.4	12.72	11.61	11.89	11.93	11.23	11.73
	Random	13.52	12.18	11.85	11.56	11.8	12.42	11.56	11.94	12.2	11.3	12.61
	Random Prioritized	12.7	12.02	11.99	11.51	11.22	12.37	11.33	12.37	11.94	11.23	11.8
Per-Core	Sequential	12.63	12.32	12.71	13.45	13.2	12.56	13.27	13.53	13.14	13.24	13.15
	Sequential Prioritized	12.39	12.34	13.11	12.98	13.11	13.23	12.6	13.79	13.12	13.48	13.21
	Random	12.92	12.48	12.89	12.87	12.93	13.6	13.2	13.67	13.15	13.97	13.24
	Random Prioritized	12.26	12.47	12.96	12.96	13.49	13.45	13.21	13.0	13.07	14.11	13.53

11.22 14.41

Figure 5.2: Results from Partitioned Work-Stealing Experiments: Broadwell

Adding the –pre-partitioning option greatly improved the performance when using one queue per group, which in this case is a CPU socket. This indicates either a large advantage to having tasks in consecutive order among Cores that share the same L3 cache, or a large advantage stemming from a smaller chunk size. The central queue per node did not show any significant differences in this experiment than the previous, which is to be expected as the pre-partitioning option has no effect in this case.

5.1.3 Hierarchical

	STATIC	GSS	TSS	FAC2	TFSS	FISS	VISS	PLS	MSTATI	MFSC	PSS
Broadwell	13.86	13.4	12.72	13.33	13.15	13.14	11.89	12.78	12.17	11.73	12.93

11.22 14.41

Figure 5.3: Results from Hierarchical Work-Stealing Experiments

In this experiment one queue is allocated for each processor and only one Core on each processor is assigned to be the Foreman, who is the only worker who is allowed to steal tasks from other queues. This resulted in similar performance to the per-Node option in the previous experiment, indicating that it does not add as much overhead due to lock contention as the other two options in the previous experiment. The best performing techniques in this experiment are VISS and MFSC, which can be explained by the hierarchical nature exploiting the benefits of a smaller chunk size while avoiding the excess overhead due to queue lock contention.

5.2 Cascade Lake

5.2.1 Work-Stealing

		STATIC	GSS	TSS	FAC2	TFSS	FISS	VISS	PLS	MSTATI	MFSC	PSS
Per-Node	Sequential	17.36	17.27	17.05	16.43	16.51	16.48	16.47	16.84	16.56	17.18	17.01
	Sequential Prioritized	17.83	17.45	16.58	16.68	16.72	16.9	16.32	17.44	16.81	16.34	17.08
	Random	17.68	17.31	16.27	17.24	16.64	16.81	15.78	17.39	16.88	16.59	17.79
	Random Prioritized	17.04	16.9	16.58	16.88	16.66	17.04	15.81	16.74	16.74	16.66	17.72
Per-Group	Sequential	17.59	17.97	17.34	17.65	17.44	16.91	16.03	17.28	17.7	16.37	17.89
	Sequential Prioritized	17.85	17.19	17.19	17.29	16.89	17.32	16.25	17.8	17.55	16.48	17.99
	Random	17.36	17.29	17.44	17.24	17.11	17.26	16.2	17.82	17.34	16.71	17.8
	Random Prioritized	17.46	17.8	17.32	17.41	17.45	17.37	16.3	17.67	17.13	16.24	18.03
Per-Core	Sequential	17.08	17.33	16.33	16.05	16.53	16.41	15.98	17.21	16.35	16.1	16.98
	Sequential Prioritized	17.46	16.55	16.22	16.54	16.53	16.78	16.0	16.84	16.02	16.11	16.53
	Random	17.43	16.98	16.38	16.59	16.84	16.41	16.48	17.08	16.49	16.44	17.43
	Random Prioritized	17.36	17.4	16.49	16.8	17.06	16.9	16.55	17.25	16.64	16.32	17.21

Figure 5.4: Results from Work-Stealing Experiments: Casecade Lake

Similar to the results from the Broadwell experiment, VISS and MFSC perform the best and Static performed the worst. However in contrast to the Broadwell experiments, the per-Core queue allocation option showed a more consistent advantage over the other queue allocations. This could be explained by a larger L1 and L2 cache relative to the L3 cache, however the Cascade Lake system has more interconnected cache lines in the microarchitecture than the Broadwell system so it is surprising that the per-Core option performed this much better than the per-Group option. As in the Broadwell experiments, the victim selection algorithm did not have as great of an effect as any other options which can be explained by the very low cost of a failed steal attempt on a shared-memory system.

The color scale for the Cascade Lake experiments ranges from 15 to 21 second, which is over a 30% increase from the Broadwell experiment results. This is interesting as the Cascade Lake system is a newer generation, has more CPU cores, a faster interconnect, and faster memory. Since only the execution time of the core connected components algorithm is measured, a possible network or file I/O bottleneck can be ruled out. This indicates that there may be a regression due to the number of CPU cores in the system.

5.2.2 Tiling

		STATIC	GSS	TSS	FAC2	TFSS	FISS	VISS	PLS	MSTATI	MFSC	PSS
Per-Node	Sequential	18.25	17.46	17.13	17.31	17.16	17.36	16.74	17.28	17.06	16.46	17.24
	Sequential Prioritized	18.66	17.6	16.84	17.4	17.04	17.29	16.65	17.29	17.46	16.46	17.24
	Random	18.46	17.85	17.07	17.32	17.34	17.03	16.63	17.47	16.93	16.75	17.06
	Random Prioritized	18.22	17.72	16.63	17.22	16.75	17.27	16.88	17.21	16.98	16.43	16.63
Per-Group	Sequential	16.92	16.85	16.26	16.18	16.09	16.84	16.27	16.68	16.37	16.52	16.64
	Sequential Prioritized	17.52	16.89	16.12	16.38	16.65	16.97	16.54	16.56	16.47	16.59	16.41
	Random	17.33	16.87	16.46	16.62	16.55	17.09	16.24	16.68	16.27	16.59	16.92
	Random Prioritized	17.28	16.45	16.45	16.31	16.41	16.57	16.33	16.75	16.56	16.41	16.63
Per-Core	Sequential	17.5	19.6	18.11	19.92	21.13	17.98	18.84	19.54	18.72	21.03	20.49
	Sequential Prioritized	17.86	19.06	18.17	19.78	20.96	18.04	18.44	19.79	18.54	20.73	20.45
	Random	17.93	19.41	18.35	20.46	21.18	18.09	19.15	19.66	18.79	20.95	20.84
	Random Prioritized	18.14	19.22	18.59	20.49	21.25	18.12	19.23	19.76	18.8	21.24	21.1

15.78 21.25

Figure 5.5: Results from Partitioned Work-Stealing Experiments: Cascade Lake

Similar to the results from the Broadwell system, the per-group option performed the best when pre partitioning the work before assigning it to the load partitioners. The per-core option, however, performed far worse even when the effect is compared to the the per Core option on the Broadwell experiment results. This would suggest that the per-Core option is not viable on systems with a large number of cores.

5.2.3 Hierarchical

	STATIC	GSS	TSS	FAC2	TFSS	FISS	VISS	PLS	MSTATI	MFSC	PSS
Cascade Lake	17.27	16.72	16.57	16.48	16.56	16.74	16.19	16.44	15.91	15.95	16.7

15.78 21.25

Figure 5.6: Results from Hierarchical Work-Stealing Experiments

Similar to the Broadwell results, MFSC and MSTATIC, and VISS performed the best on the hierarchical setup. On the Cascade Lake test system the hierarchical implementation resulted in slightly better performance than the baseline centralized queue implementation for certain scheduling schemes which indicates a potential for hierarchical work stealing architectures for systems with a high number of CPU cores.

6
Conclusion

As seen in the program execution time heatmaps, the choice of scheduling algorithm, the allocation of the task queues, and the victim selection algorithm all have an effect on the performance of parallelized operations in DAPHNE. The results also varied depending on the test system and application, so every use-case should be analysed individually. With a cyclic distribution of tasks the per-group queue allocation resulted in the lowest performance on both test systems, which would indicate that either the L1 and L2 cache has a larger effect than the L3 cache and the cost of an intra-socket memory access, or alternating between memory sockets for consecutive tasks performs worse than effectively assigning tasks to workers randomly using a centralized queue. When tiling the input into sections and using separate load paratitioners for each queue both test systems showed the best performed with per-socket queues and the worst performance with per-core queues. Since the poor performance for the per-core queue allocation does not appear in the non-tiling approach, it must be caused by the resulting smaller chunk sizes, which result in more tasks, which seem to cause an exponential slowdown when combined with the large number of queues on the per-core approach. The slightly better performance for the per-socket queues can be caused by better data locality and cache-awareness across multiple iterations of the components algorithm, or it can be a propagated effect of the smaller chunk sizes due to there being two load partitioners instead of one.

The scheduling schemes that are available in the DAPHNE infrastructure seem to cover a wide variety of situations. The different test systems showing different performance characteristics highlights the importance of exposing scheduling knobs to the user so that each application and system can be configured for each use-case individually.

7
Future Work

Further analysis could be conducted with experiments on additional systems and more quantitative measurements such as the number of cache misses, the number of stolen tasks, and the summation of the size of the stolen tasks. Recording the number of cache misses would further narrow down the cause of the results in this Thesis, and it would reveal to which extent the results are caused by better cache-awareness and how much can be attributed to other effects. Recording the number of stolen tasks would reveal to what extent the low performance results are caused by the victim selection logic, which can cause high lock contention when applied inefficiently. The number of stolen tasks can also be divided into the number of successful and failed steal attempts. The total cumulative task size stolen would provide a metric for how much load imbalance an application would have had if work stealing was not applied, similar to looking at a task trace profile.

In order to evaluate the results of the queue allocation configuration and the results from the different chunk sizes calculated using the scheduling scheme separately, further experiments could be conducted with fixed chunk sizes. Even though there are fixed-sized chunk scheduling schemes used in this Thesis, the calculation of the chunk size is dependant on the input parameters. Experiments with globally fixed size chunks can offer comparisons between various queue configurations while eliminating chunk size factors. Since the components microbenchmark involved file I/O, which in this case likely occurred in full by one thread, there can be theoretically be a further reduction in the number of cache misses and in turn an even larger increase in performance by intelligently distributing the file I/O among the memory domains. The readMatrix operation could split the file input operations evenly among threads across the NUMA domains, which can then be respected when assigning the further tasks to the workers across the NUMA domains. Depending on the implementation, the file I/O operations could even be fused with the calculation kernels. This would result in an improvement in memory access throughput and latency as it would reduce the use of the memory interconnect and it could also result in fewer cache misses, even for the first computation iteration directly after the file I/O.

As seen in the task trace plots, each operation is vectorized through the multithreaded wrapper separately. The configured options such as the scheduling technique, minimum chunk size, and number of threads is only provided once for the entire application. Since some

operations behave differently than others, it can be profitable to customize the scheduling behavior depending on the operation at hand. In the examples in this Thesis, the transpose operation seems to execute the fastest, while the fused operators that execute many kernels take orders of magnitude longer. A compensation factor for each kernel could be added to take this discrepancy into account. Another possibility to ease this discrepancy would be to add scheduling hints on a per function basis directly in the DAPHNE DSL, so that the programmer can optionally add what they believe would increase performance directly while writing the DAPHNE application. However, since only function pointers to the compiled operations are passed to the multithreaded wrapper, this can prove difficult to implement.

The execution trace plots proved useful to debug slow performance in the vectorized execution engine, as it shows the number of idle workers caused by load imbalance at the end of a program's execution. These plots can be further extended with the names of the kernels and the data type that each kernel is called with.

Another aspect that was not tested in this Thesis is the effects of simultaneous multithreading has on the performance of per-group and per-core queues. Since certain applications, particularly applications with sporadic file or network I/O can benefit from simultaneous multithreading but would in turn suffer from a busier L2 and L3 cache, the per-core queue option could become a more viable option to more effectively make use of this busier cache.

In summary, the DAPHNE infrastructure can greatly benefit from work-stealing task-based scheduling techniques to improve memory access times, reduce cache misses, and rearrange work resulting in an increase in performance. Future work parallelizing further operations and offering more fine-tuned scheduling configurations has the potential to improve performance when processing integrated data analysis pipelines even further.

Bibliography

[1] Umut A Acar, Guy E Blelloch, and Robert D Blumofe. The data locality of work stealing. In *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, pages 1–12, 2000. doi: 10.1145/341800.341801.

[2] Dana Akhmetova, Gokcen Kestor, Roberto Gioiosa, Stefano Markidis, and Erwin Laure. On the application task granularity and the interplay with the scheduling overhead in many-core shared memory systems. In *2015 IEEE International Conference on Cluster Computing*, pages 428–437. IEEE, 2015. doi: 10.1109/CLUSTER.2015.65.

[3] Nimar S Arora, Robert D Blumofe, and C Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of computing systems*, 34(2):115–144, 2001. doi: 10.1007/s00224-001-0004-z.

[4] Blaise Barney and Donald Frederick. Introduction to parallel computing tutorial, 2022. URL https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial.

[5] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999. doi: 10.1145/324133.324234.

[6] Quan Chen and Minyi Guo. *Task scheduling for multi-core and parallel architectures*. Springer, 2017. doi: 10.1007/978-981-10-6238-4.

[7] Florina M. Ciorba, Patrick Damme, Ahmed Eleliemy, Vasileios Karakostas, and Gabrielle Poerwawinata. D5.1 scheduler design for pipelines and tasks, 2022. URL http://daphne-eu.eu/wp-content/uploads/2021/11/Deliverable-5.1-fin.pdf.

[8] TechTarget Contributor. What is runtime system? - definition from whatis.com, Feb 2017. URL https://www.techtarget.com/whatis/definition/runtime-system.

[9] Idriss Daoudi, Philippe Virouleau, Thierry Gautier, Samuel Thibault, and Olivier Aumage. somp: Simulating openmp task-based applications with numa effects. In *International Workshop on OpenMP*, pages 197–211. Springer, 2020. doi: 10.1007/978-3-030-58144-2\{_}13.

[10] Ahmed Eleliemy and Florina M. Ciorba. A distributed chunk calculation approach for self-scheduling of parallel applications on distributed-memory systems. *Journal of Computational Science*, 51:101284, 2021. ISSN 1877-7503. doi: https://doi.org/

10.1016/j.jocs.2020.101284. URL https://www.sciencedirect.com/science/article/pii/S1877750320305792.

[11] Google. Best practices for running tightly coupled hpc applications on compute engine, 2021. URL https://cloud.google.com/architecture/best-practices-for-using-mpi-on-compute-engine#disable_simultaneous_multithreading.

[12] Pablo Halpern. Parallel program execution using work stealing. Lecture, 2015. URL https://github.com/CppCon/CppCon2015/blob/master/Presentations/Work%20Stealing/Work%20Stealing%20-%20Pablo%20Halpern%20-%20CppCon%202015.pptx.

[13] Susan Flynn Hummel, Edith Schonberg, and Lawrence E Flynn. Factoring: A practical and robust method for scheduling parallel loops. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 610–632, 1991. doi: 10.1145/125826.126137.

[14] Nina Ihde, Paula Marten, Ahmed Eleliemy, Gabrielle Poerwawinata, Pedro Silva, Ilin Tolovski, Florina M Ciorba, and Tilmann Rabl. A survey of big data, high performance computing, and machine learning benchmarks. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 98–118. Springer, 2021.

[15] Intel. Intel xeon processor e5-2640 v4 specifications. URL: https://ark.intel.com/content/www/us/en/ark/products/92984/intel-xeon-processor-e52640-v4-25m-cache-2-40-ghz.html, 2016.

[16] An Intel. Introduction to the intel quickpath interconnect. *White Paper*, 2009. URL https://www.intel.com/content/dam/doc/white-paper/quick-path-interconnect-introduction-paper.pdf.

[17] Vivek Kale and Martin Kong. Enhancing support in openmp to improve data locality in application programs using task scheduling. URL: https://openmpcon.org/wp-content/uploads/2018_Session3_Li.pdf, 7 2018.

[18] Christoph Lameter. Numa (non-uniform memory access): An overview, 2013. URL https://queue.acm.org/detail.cfm?id=2513149.

[19] Jure Leskovec, Lada A Adamic, and Bernardo A Huberman. The dynamics of viral marketing. *ACM Transactions on the Web (TWEB)*, 1(1):5–es, 2007. doi: 10.1145/1232722.1232727.

[20] LLVM. Llvm openmp runtime library kmp_tasking.cpp. https://github.com/llvm/llvm-project/blob/main/openmp/runtime/src/kmp_tasking.cpp, 2022. Accessed: 2022-07-05.

[21] David L Mulnix, Jul 2017. URL https://www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html.

[22] David L Mulnix. Intel Xeon Processor Scalable Family Technical Overview. Technical report, Intel, 2017. URL https://www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html.

[23] Stephen L Olivier, Allan K Porterfield, Kyle B Wheeler, and Jan F Prins. Scheduling task parallelism on multi-socket multicore systems. In *Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers*, pages 49–56, 2011. doi: 10.1145/1988796.1988804.

[24] Stephen L Olivier, Allan K Porterfield, Kyle B Wheeler, Michael Spiegel, and Jan F Prins. Openmp task scheduling strategies for multicore numa systems. *The International Journal of High Performance Computing Applications*, 26(2):110–124, 2012. doi: 10.1177/1094342011434065.

[25] Swann Perarnau and Mitsuhisa Sato. Victim selection and distributed work stealing performance: A case study. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 659–668. IEEE, 2014. doi: 10.1109/IPDPS.2014.74.

[26] Teebu Philip. *Increasing chunk size loop scheduling algorithms for data independent loops*. PhD thesis, Pennsylvania State University, 1995. URL https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.56.5370&rep=rep1&type=pdf.

[27] Constantine D. Polychronopoulos and David J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, C-36(12):1425–1439, 1987. doi: 10.1109/TC.1987.5009495.

[28] Jean-Noël Quintin and Frédéric Wagner. Hierarchical work-stealing. In *European Conference on Parallel Processing*, pages 217–229. Springer, 2010. doi: 10.1007/978-3-642-15277-1\{_}21.

[29] Florian Schmaus, Florian Fischer, Timo Hönig, and Wolfgang Schröder-Preikschat. Modern concurrency platforms require modern system-call techniques. *Technical Reports*, 2021. ISSN 2191-5008.

[30] Muhammad Shaaban. Eecc722 - simultaneous multithreading (smt). University Lecture, 2012. URL http://meseec.ce.rit.edu/eecc722-fall2012/722-9-3-2012.pdf.

[31] Kyle Singer, Yifan Xu, and I-Ting Angelina Lee. Proactive work stealing for futures. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 257–271, 2019. doi: 10.1145/3293883.3295735.

[32] Nikki Sonenberg, Grzegorz Kielanski, and Benny Van Houdt. Performance analysis of work stealing in large-scale multithreaded computing. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, 6(2):1–28, 2021. doi: 10.1145/3470887.

[33] Peter Thoman, Khalid Hasanov, Kiril Dichev, Roman Iakymchuk, Xavier Aguilar, Philipp Gschwandtner, Pierre Lemarinier, Stefano Markidis, Herbert Jordan, Erwin

Laure, Kostas Katrinis, Dimitrios Nikolopoulos, and Thomas Fahringer. *A Taxonomy of Task-Based Technologies for High-Performance Computing*, pages 264–274. Springer Cham, 03 2018. ISBN 978-3-319-78053-5. doi: 10.1007/978-3-319-78054-2\{_}25.

[34] Alexandros Tzannes, George C Caragea, Rajeev Barua, and Uzi Vishkin. Lazy binary-splitting: a run-time adaptive work-stealing scheduler. *ACM Sigplan Notices*, 45(5): 179–190, 2010. doi: 10.1145/1837853.1693479.

[35] Yizhuo Wang, Weixing Ji, Qi Zuo, and Feng Shi. A hierarchical work-stealing framework for multi-core clusters. In *2012 13th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 350–355. IEEE, 2012. doi: 10.1109/PDCAT.2012.17.

[36] 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. doi: 10.1016/j.parco.2014.09.012.

A

Result Data

This is a readout of all of the resulting data from the experiments performed that is used in the results section. In the table below, N refers to the number of repetitions of each experiment, μ refers to the mean of all the results from the given repetitions, and σ refers to the Standard Deviation of the results for the given repetitions.

Node	Branch	Application Arguments	N	μ	σ
1	work-stealing	–pin-workers –STATIC –CENTRALIZED –SEQ	20	13.71	0.90
1	work-stealing	–pin-workers –STATIC –CENTRALIZED –SEQPRI	20	13.89	1.06
1	work-stealing	–pin-workers –STATIC –CENTRALIZED –RANDOM	20	14.20	1.26
1	work-stealing	–pin-workers –STATIC –CENTRALIZED –RANDOMPRI	20	14.20	1.31
1	work-stealing	–pin-workers –STATIC –PERGROUP –SEQ	20	13.90	1.07
1	work-stealing	–pin-workers –STATIC –PERGROUP –SEQPRI	20	14.24	1.21
1	work-stealing	–pin-workers –STATIC –PERGROUP –RANDOM	20	13.63	0.88
1	work-stealing	–pin-workers –STATIC –PERGROUP –RANDOMPRI	20	14.30	1.16
1	work-stealing	–pin-workers –STATIC –PERCPU –SEQ	20	13.71	1.04
1	work-stealing	–pin-workers –STATIC –PERCPU –SEQPRI	20	13.80	1.20
1	work-stealing	–pin-workers –STATIC –PERCPU –RANDOM	20	13.11	0.84
1	work-stealing	–pin-workers –STATIC –PERCPU –RANDOMPRI	20	13.58	1.08
1	work-stealing	–pin-workers –GSS –CENTRALIZED –SEQ	20	13.56	1.24
1	work-stealing	–pin-workers –GSS –CENTRALIZED –SEQPRI	20	12.96	0.89
1	work-stealing	–pin-workers –GSS –CENTRALIZED –RANDOM	20	13.24	1.07
1	work-stealing	–pin-workers –GSS –CENTRALIZED –RANDOMPRI	20	13.36	1.19
1	work-stealing	–pin-workers –GSS –PERGROUP –SEQ	20	13.75	1.22
1	work-stealing	–pin-workers –GSS –PERGROUP –SEQPRI	20	13.23	0.93
1	work-stealing	–pin-workers –GSS –PERGROUP –RANDOM	20	13.27	0.93
1	work-stealing	–pin-workers –GSS –PERGROUP –RANDOMPRI	20	13.54	1.08
1	work-stealing	–pin-workers –GSS –PERCPU –SEQ	20	13.72	0.98
1	work-stealing	–pin-workers –GSS –PERCPU –SEQPRI	20	13.79	0.94
1	work-stealing	–pin-workers –GSS –PERCPU –RANDOM	20	13.92	1.13
1	work-stealing	–pin-workers –GSS –PERCPU –RANDOMPRI	20	13.51	0.67
1	work-stealing	–pin-workers –TSS –CENTRALIZED –SEQ	20	13.35	1.22
1	work-stealing	–pin-workers –TSS –CENTRALIZED –SEQPRI	20	12.87	0.90
1	work-stealing	–pin-workers –TSS –CENTRALIZED –RANDOM	20	12.63	0.53
1	work-stealing	–pin-workers –TSS –CENTRALIZED –RANDOMPRI	20	13.58	1.31
1	work-stealing	–pin-workers –TSS –PERGROUP –SEQ	20	13.28	0.70
1	work-stealing	–pin-workers –TSS –PERGROUP –SEQPRI	20	13.62	1.10
1	work-stealing	–pin-workers –TSS –PERGROUP –RANDOM	20	13.38	0.92
1	work-stealing	–pin-workers –TSS –PERGROUP –RANDOMPRI	20	13.52	0.99
1	work-stealing	–pin-workers –TSS –PERCPU –SEQ	20	13.52	1.20
1	work-stealing	–pin-workers –TSS –PERCPU –SEQPRI	20	13.04	0.83
1	work-stealing	–pin-workers –TSS –PERCPU –RANDOM	20	13.65	1.26
1	work-stealing	–pin-workers –TSS –PERCPU –RANDOMPRI	20	13.45	1.24
1	work-stealing	–pin-workers –FAC2 –CENTRALIZED –SEQ	20	12.68	0.93
1	work-stealing	–pin-workers –FAC2 –CENTRALIZED –SEQPRI	20	12.90	1.07
1	work-stealing	–pin-workers –FAC2 –CENTRALIZED –RANDOM	20	12.64	0.94
1	work-stealing	–pin-workers –FAC2 –CENTRALIZED –RANDOMPRI	20	13.03	1.32
1	work-stealing	–pin-workers –FAC2 –PERGROUP –SEQ	20	12.96	0.80
1	work-stealing	–pin-workers –FAC2 –PERGROUP –SEQPRI	20	12.93	0.76
1	work-stealing	–pin-workers –FAC2 –PERGROUP –RANDOM	20	12.92	0.73
1	work-stealing	–pin-workers –FAC2 –PERGROUP –RANDOMPRI	20	13.29	1.09
1	work-stealing	–pin-workers –FAC2 –PERCPU –SEQ	20	13.16	1.06
1	work-stealing	–pin-workers –FAC2 –PERCPU –SEQPRI	20	13.35	1.16
1	work-stealing	–pin-workers –FAC2 –PERCPU –RANDOM	20	13.85	1.24
1	work-stealing	–pin-workers –FAC2 –PERCPU –RANDOMPRI	20	13.04	0.91
1	work-stealing	–pin-workers –TFSS –CENTRALIZED –SEQ	20	12.54	0.88

1	work-stealing	–pin-workers –TFSS –CENTRALIZED –SEQPRI	20	12.52	0.87
1	work-stealing	–pin-workers –TFSS –CENTRALIZED –RANDOM	20	12.20	0.20
1	work-stealing	–pin-workers –TFSS –CENTRALIZED –RANDOMPRI	20	12.43	0.76
1	work-stealing	–pin-workers –TFSS –PERGROUP –SEQ	20	13.14	0.99
1	work-stealing	–pin-workers –TFSS –PERGROUP –SEQPRI	20	13.43	1.08
1	work-stealing	–pin-workers –TFSS –PERGROUP –RANDOM	20	13.18	1.01
1	work-stealing	–pin-workers –TFSS –PERGROUP –RANDOMPRI	20	13.18	1.02
1	work-stealing	–pin-workers –TFSS –PERCPU –SEQ	20	13.57	0.90
1	work-stealing	–pin-workers –TFSS –PERCPU –SEQPRI	20	13.85	1.10
1	work-stealing	–pin-workers –TFSS –PERCPU –RANDOM	20	13.51	0.87
1	work-stealing	–pin-workers –TFSS –PERCPU –RANDOMPRI	20	13.39	0.69
1	work-stealing	–pin-workers –FISS –CENTRALIZED –SEQ	20	13.92	1.01
1	work-stealing	–pin-workers –FISS –CENTRALIZED –SEQPRI	20	13.80	0.94
1	work-stealing	–pin-workers –FISS –CENTRALIZED –RANDOM	20	14.09	1.08
1	work-stealing	–pin-workers –FISS –CENTRALIZED –RANDOMPRI	20	14.03	1.09
1	work-stealing	–pin-workers –FISS –PERGROUP –SEQ	20	13.98	0.72
1	work-stealing	–pin-workers –FISS –PERGROUP –SEQPRI	20	14.39	1.05
1	work-stealing	–pin-workers –FISS –PERGROUP –RANDOM	20	14.23	0.98
1	work-stealing	–pin-workers –FISS –PERGROUP –RANDOMPRI	20	14.34	0.98
1	work-stealing	–pin-workers –FISS –PERCPU –SEQ	20	13.68	1.07
1	work-stealing	–pin-workers –FISS –PERCPU –SEQPRI	20	13.75	1.06
1	work-stealing	–pin-workers –FISS –PERCPU –RANDOM	20	13.28	0.69
1	work-stealing	–pin-workers –FISS –PERCPU –RANDOMPRI	20	13.91	1.11
1	work-stealing	–pin-workers –VISS –CENTRALIZED –SEQ	20	12.53	1.18
1	work-stealing	–pin-workers –VISS –CENTRALIZED –SEQPRI	20	12.87	1.21
1	work-stealing	–pin-workers –VISS –CENTRALIZED –RANDOM	20	12.37	1.09
1	work-stealing	–pin-workers –VISS –CENTRALIZED –RANDOMPRI	20	12.28	0.95
1	work-stealing	–pin-workers –VISS –PERGROUP –SEQ	20	13.33	1.03
1	work-stealing	–pin-workers –VISS –PERGROUP –SEQPRI	20	13.51	1.08
1	work-stealing	–pin-workers –VISS –PERGROUP –RANDOM	20	13.08	0.96
1	work-stealing	–pin-workers –VISS –PERGROUP –RANDOMPRI	20	13.54	1.16
1	work-stealing	–pin-workers –VISS –PERCPU –SEQ	20	12.10	0.86
1	work-stealing	–pin-workers –VISS –PERCPU –SEQPRI	20	13.01	1.32
1	work-stealing	–pin-workers –VISS –PERCPU –RANDOM	20	12.64	1.17
1	work-stealing	–pin-workers –VISS –PERCPU –RANDOMPRI	20	12.17	0.91
1	work-stealing	–pin-workers –PLS –CENTRALIZED –SEQ	20	13.32	1.22
1	work-stealing	–pin-workers –PLS –CENTRALIZED –SEQPRI	20	12.97	1.00
1	work-stealing	–pin-workers –PLS –CENTRALIZED –RANDOM	20	13.15	1.12
1	work-stealing	–pin-workers –PLS –CENTRALIZED –RANDOMPRI	20	13.40	1.24
1	work-stealing	–pin-workers –PLS –PERGROUP –SEQ	20	13.22	0.95
1	work-stealing	–pin-workers –PLS –PERGROUP –SEQPRI	20	13.59	1.15
1	work-stealing	–pin-workers –PLS –PERGROUP –RANDOM	20	13.19	0.90
1	work-stealing	–pin-workers –PLS –PERGROUP –RANDOMPRI	20	12.82	0.15
1	work-stealing	–pin-workers –PLS –PERCPU –SEQ	20	13.24	0.98
1	work-stealing	–pin-workers –PLS –PERCPU –SEQPRI	20	13.41	1.06
1	work-stealing	–pin-workers –PLS –PERCPU –RANDOM	20	13.41	1.03
1	work-stealing	–pin-workers –PLS –PERCPU –RANDOMPRI	20	13.97	1.26
1	work-stealing	–pin-workers –MSTATIC –CENTRALIZED –SEQ	20	13.10	0.85
1	work-stealing	–pin-workers –MSTATIC –CENTRALIZED –SEQPRI	20	13.31	1.03
1	work-stealing	–pin-workers –MSTATIC –CENTRALIZED –RANDOM	20	13.05	0.88
1	work-stealing	–pin-workers –MSTATIC –CENTRALIZED –RANDOMPRI	20	13.61	1.14
1	work-stealing	–pin-workers –MSTATIC –PERGROUP –SEQ	20	14.17	1.08
1	work-stealing	–pin-workers –MSTATIC –PERGROUP –SEQPRI	20	14.15	1.08
1	work-stealing	–pin-workers –MSTATIC –PERGROUP –RANDOM	20	14.41	1.24
1	work-stealing	–pin-workers –MSTATIC –PERGROUP –RANDOMPRI	20	14.18	1.21
1	work-stealing	–pin-workers –MSTATIC –PERCPU –SEQ	20	13.22	1.09
1	work-stealing	–pin-workers –MSTATIC –PERCPU –SEQPRI	20	13.42	1.20
1	work-stealing	–pin-workers –MSTATIC –PERCPU –RANDOM	20	13.09	0.97
1	work-stealing	–pin-workers –MSTATIC –PERCPU –RANDOMPRI	20	13.55	1.15
1	work-stealing	–pin-workers –MFSC –CENTRALIZED –SEQ	20	11.83	1.12
1	work-stealing	–pin-workers –MFSC –CENTRALIZED –SEQPRI	20	11.84	1.08
1	work-stealing	–pin-workers –MFSC –CENTRALIZED –RANDOM	20	11.86	1.12
1	work-stealing	–pin-workers –MFSC –CENTRALIZED –RANDOMPRI	20	11.71	1.13
1	work-stealing	–pin-workers –MFSC –PERGROUP –SEQ	20	13.16	0.70
1	work-stealing	–pin-workers –MFSC –PERGROUP –SEQPRI	20	12.76	0.69
1	work-stealing	–pin-workers –MFSC –PERGROUP –RANDOM	20	13.27	0.81
1	work-stealing	–pin-workers –MFSC –PERGROUP –RANDOMPRI	20	13.19	0.77
1	work-stealing	–pin-workers –MFSC –PERCPU –SEQ	20	12.40	0.99
1	work-stealing	–pin-workers –MFSC –PERCPU –SEQPRI	20	12.88	1.20
1	work-stealing	–pin-workers –MFSC –PERCPU –RANDOM	20	12.57	1.16
1	work-stealing	–pin-workers –MFSC –PERCPU –RANDOMPRI	20	12.25	0.77
1	work-stealing	–pin-workers –PSS –CENTRALIZED –SEQ	20	12.72	0.98
1	work-stealing	–pin-workers –PSS –CENTRALIZED –SEQPRI	20	13.09	1.15
1	work-stealing	–pin-workers –PSS –CENTRALIZED –RANDOM	20	13.25	1.17
1	work-stealing	–pin-workers –PSS –CENTRALIZED –RANDOMPRI	20	12.93	1.03
1	work-stealing	–pin-workers –PSS –PERGROUP –SEQ	20	13.27	1.06
1	work-stealing	–pin-workers –PSS –PERGROUP –SEQPRI	20	13.39	1.09
1	work-stealing	–pin-workers –PSS –PERGROUP –RANDOM	20	13.76	1.20
1	work-stealing	–pin-workers –PSS –PERGROUP –RANDOMPRI	20	13.49	1.18
1	work-stealing	–pin-workers –PSS –PERCPU –SEQ	20	13.83	0.90
1	work-stealing	–pin-workers –PSS –PERCPU –SEQPRI	20	13.63	0.93
1	work-stealing	–pin-workers –PSS –PERCPU –RANDOM	20	13.84	0.95
1	work-stealing	–pin-workers –PSS –PERCPU –RANDOMPRI	20	13.75	0.86
27	work-stealing	–pin-workers –STATIC –CENTRALIZED –SEQ	20	17.36	1.55

27	work-stealing	–pin-workers –STATIC –CENTRALIZED –SEQPRI	20	17.83	1.70
27	work-stealing	–pin-workers –STATIC –CENTRALIZED –RANDOM	20	17.68	1.46
27	work-stealing	–pin-workers –STATIC –CENTRALIZED –RANDOMPRI	20	17.04	1.19
27	work-stealing	–pin-workers –STATIC –PERGROUP –SEQ	20	17.59	0.92
27	work-stealing	–pin-workers –STATIC –PERGROUP –SEQPRI	20	17.85	0.97
27	work-stealing	–pin-workers –STATIC –PERGROUP –RANDOM	20	17.36	0.86
27	work-stealing	–pin-workers –STATIC –PERGROUP –RANDOMPRI	20	17.46	0.91
27	work-stealing	–pin-workers –STATIC –PERCPU –SEQ	20	17.08	0.69
27	work-stealing	–pin-workers –STATIC –PERCPU –SEQPRI	20	17.46	0.91
27	work-stealing	–pin-workers –STATIC –PERCPU –RANDOM	20	17.43	1.00
27	work-stealing	–pin-workers –STATIC –PERCPU –RANDOMPRI	20	17.36	0.93
27	work-stealing	–pin-workers –GSS –CENTRALIZED –SEQ	20	17.27	1.21
27	work-stealing	–pin-workers –GSS –CENTRALIZED –SEQPRI	20	17.45	1.40
27	work-stealing	–pin-workers –GSS –CENTRALIZED –RANDOM	20	17.31	1.36
27	work-stealing	–pin-workers –GSS –CENTRALIZED –RANDOMPRI	20	16.90	1.46
27	work-stealing	–pin-workers –GSS –PERGROUP –SEQ	20	17.97	0.91
27	work-stealing	–pin-workers –GSS –PERGROUP –SEQPRI	20	17.19	0.67
27	work-stealing	–pin-workers –GSS –PERGROUP –RANDOM	20	17.29	0.94
27	work-stealing	–pin-workers –GSS –PERGROUP –RANDOMPRI	20	17.80	0.91
27	work-stealing	–pin-workers –GSS –PERCPU –SEQ	20	17.33	0.75
27	work-stealing	–pin-workers –GSS –PERCPU –SEQPRI	20	16.55	0.97
27	work-stealing	–pin-workers –GSS –PERCPU –RANDOM	20	16.98	0.94
27	work-stealing	–pin-workers –GSS –PERCPU –RANDOMPRI	20	17.40	0.95
27	work-stealing	–pin-workers –TSS –CENTRALIZED –SEQ	20	17.05	1.76
27	work-stealing	–pin-workers –TSS –CENTRALIZED –SEQPRI	20	16.58	1.36
27	work-stealing	–pin-workers –TSS –CENTRALIZED –RANDOM	20	16.27	1.47
27	work-stealing	–pin-workers –TSS –CENTRALIZED –RANDOMPRI	20	16.58	1.32
27	work-stealing	–pin-workers –TSS –PERGROUP –SEQ	20	17.34	0.94
27	work-stealing	–pin-workers –TSS –PERGROUP –SEQPRI	20	17.19	0.93
27	work-stealing	–pin-workers –TSS –PERGROUP –RANDOM	20	17.44	1.02
27	work-stealing	–pin-workers –TSS –PERGROUP –RANDOMPRI	20	17.32	0.70
27	work-stealing	–pin-workers –TSS –PERCPU –SEQ	20	16.33	1.12
27	work-stealing	–pin-workers –TSS –PERCPU –SEQPRI	20	16.22	0.94
27	work-stealing	–pin-workers –TSS –PERCPU –RANDOM	20	16.38	0.82
27	work-stealing	–pin-workers –TSS –PERCPU –RANDOMPRI	20	16.49	0.89
27	work-stealing	–pin-workers –FAC2 –CENTRALIZED –SEQ	20	16.43	1.01
27	work-stealing	–pin-workers –FAC2 –CENTRALIZED –SEQPRI	20	16.68	1.53
27	work-stealing	–pin-workers –FAC2 –CENTRALIZED –RANDOM	20	17.24	1.48
27	work-stealing	–pin-workers –FAC2 –CENTRALIZED –RANDOMPRI	20	16.88	1.70
27	work-stealing	–pin-workers –FAC2 –PERGROUP –SEQ	20	17.65	0.87
27	work-stealing	–pin-workers –FAC2 –PERGROUP –SEQPRI	20	17.29	0.88
27	work-stealing	–pin-workers –FAC2 –PERGROUP –RANDOM	20	17.24	0.72
27	work-stealing	–pin-workers –FAC2 –PERGROUP –RANDOMPRI	20	17.41	0.88
27	work-stealing	–pin-workers –FAC2 –PERCPU –SEQ	20	16.05	0.93
27	work-stealing	–pin-workers –FAC2 –PERCPU –SEQPRI	20	16.54	0.88
27	work-stealing	–pin-workers –FAC2 –PERCPU –RANDOM	20	16.59	0.82
27	work-stealing	–pin-workers –FAC2 –PERCPU –RANDOMPRI	20	16.80	0.78
27	work-stealing	–pin-workers –TFSS –CENTRALIZED –SEQ	20	16.51	1.15
27	work-stealing	–pin-workers –TFSS –CENTRALIZED –SEQPRI	20	16.72	1.35
27	work-stealing	–pin-workers –TFSS –CENTRALIZED –RANDOM	20	16.64	1.39
27	work-stealing	–pin-workers –TFSS –CENTRALIZED –RANDOMPRI	20	16.66	1.17
27	work-stealing	–pin-workers –TFSS –PERGROUP –SEQ	20	17.44	0.92
27	work-stealing	–pin-workers –TFSS –PERGROUP –SEQPRI	20	16.89	0.91
27	work-stealing	–pin-workers –TFSS –PERGROUP –RANDOM	20	17.11	0.78
27	work-stealing	–pin-workers –TFSS –PERGROUP –RANDOMPRI	20	17.45	0.95
27	work-stealing	–pin-workers –TFSS –PERCPU –SEQ	20	16.53	0.93
27	work-stealing	–pin-workers –TFSS –PERCPU –SEQPRI	20	16.53	1.06
27	work-stealing	–pin-workers –TFSS –PERCPU –RANDOM	20	16.84	1.13
27	work-stealing	–pin-workers –TFSS –PERCPU –RANDOMPRI	20	17.06	0.97
27	work-stealing	–pin-workers –FISS –CENTRALIZED –SEQ	20	16.48	1.22
27	work-stealing	–pin-workers –FISS –CENTRALIZED –SEQPRI	20	16.90	1.07
27	work-stealing	–pin-workers –FISS –CENTRALIZED –RANDOM	20	16.81	1.33
27	work-stealing	–pin-workers –FISS –CENTRALIZED –RANDOMPRI	20	17.04	1.21
27	work-stealing	–pin-workers –FISS –PERGROUP –SEQ	20	16.91	0.65
27	work-stealing	–pin-workers –FISS –PERGROUP –SEQPRI	20	17.32	0.89
27	work-stealing	–pin-workers –FISS –PERGROUP –RANDOM	20	17.26	0.83
27	work-stealing	–pin-workers –FISS –PERGROUP –RANDOMPRI	20	17.37	0.81
27	work-stealing	–pin-workers –FISS –PERCPU –SEQ	20	16.41	0.69
27	work-stealing	–pin-workers –FISS –PERCPU –SEQPRI	20	16.78	1.06
27	work-stealing	–pin-workers –FISS –PERCPU –RANDOM	20	16.41	1.06
27	work-stealing	–pin-workers –FISS –PERCPU –RANDOMPRI	20	16.90	1.19
27	work-stealing	–pin-workers –VISS –CENTRALIZED –SEQ	20	16.47	1.26
27	work-stealing	–pin-workers –VISS –CENTRALIZED –SEQPRI	20	16.32	1.37
27	work-stealing	–pin-workers –VISS –CENTRALIZED –RANDOM	20	15.78	1.23
27	work-stealing	–pin-workers –VISS –CENTRALIZED –RANDOMPRI	20	15.81	1.38
27	work-stealing	–pin-workers –VISS –PERGROUP –SEQ	20	16.03	0.98
27	work-stealing	–pin-workers –VISS –PERGROUP –SEQPRI	20	16.25	0.79
27	work-stealing	–pin-workers –VISS –PERGROUP –RANDOM	20	16.20	0.79
27	work-stealing	–pin-workers –VISS –PERGROUP –RANDOMPRI	20	16.30	0.92
27	work-stealing	–pin-workers –VISS –PERCPU –SEQ	20	15.98	0.96
27	work-stealing	–pin-workers –VISS –PERCPU –SEQPRI	20	16.00	0.95
27	work-stealing	–pin-workers –VISS –PERCPU –RANDOM	20	16.48	0.98
27	work-stealing	–pin-workers –VISS –PERCPU –RANDOMPRI	20	16.55	1.36
27	work-stealing	–pin-workers –PLS –CENTRALIZED –SEQ	20	16.84	1.60

27	work-stealing	–pin-workers –PLS –CENTRALIZED –SEQPRI	20	17.44	1.57
27	work-stealing	–pin-workers –PLS –CENTRALIZED –RANDOM	20	17.39	1.55
27	work-stealing	–pin-workers –PLS –CENTRALIZED –RANDOMPRI	20	16.74	1.24
27	work-stealing	–pin-workers –PLS –PERGROUP –SEQ	20	17.28	0.96
27	work-stealing	–pin-workers –PLS –PERGROUP –SEQPRI	20	17.80	1.09
27	work-stealing	–pin-workers –PLS –PERGROUP –RANDOM	20	17.82	0.81
27	work-stealing	–pin-workers –PLS –PERGROUP –RANDOMPRI	20	17.67	0.80
27	work-stealing	–pin-workers –PLS –PERCPU –SEQ	20	17.21	0.95
27	work-stealing	–pin-workers –PLS –PERCPU –SEQPRI	20	16.84	1.02
27	work-stealing	–pin-workers –PLS –PERCPU –RANDOM	20	17.08	1.04
27	work-stealing	–pin-workers –PLS –PERCPU –RANDOMPRI	20	17.25	0.92
27	work-stealing	–pin-workers –MSTATIC –CENTRALIZED –SEQ	20	16.56	1.21
27	work-stealing	–pin-workers –MSTATIC –CENTRALIZED –SEQPRI	20	16.81	1.44
27	work-stealing	–pin-workers –MSTATIC –CENTRALIZED –RANDOM	20	16.88	1.32
27	work-stealing	–pin-workers –MSTATIC –CENTRALIZED –RANDOMPRI	20	16.74	1.45
27	work-stealing	–pin-workers –MSTATIC –PERGROUP –SEQ	20	17.70	0.63
27	work-stealing	–pin-workers –MSTATIC –PERGROUP –SEQPRI	20	17.55	0.68
27	work-stealing	–pin-workers –MSTATIC –PERGROUP –RANDOM	20	17.34	0.79
27	work-stealing	–pin-workers –MSTATIC –PERGROUP –RANDOMPRI	20	17.13	0.80
27	work-stealing	–pin-workers –MSTATIC –PERCPU –SEQ	20	16.35	0.69
27	work-stealing	–pin-workers –MSTATIC –PERCPU –SEQPRI	20	16.02	0.83
27	work-stealing	–pin-workers –MSTATIC –PERCPU –RANDOM	20	16.49	0.94
27	work-stealing	–pin-workers –MSTATIC –PERCPU –RANDOMPRI	20	16.64	0.74
27	work-stealing	–pin-workers –MFSC –CENTRALIZED –SEQ	20	17.18	1.21
27	work-stealing	–pin-workers –MFSC –CENTRALIZED –SEQPRI	20	16.34	1.23
27	work-stealing	–pin-workers –MFSC –CENTRALIZED –RANDOM	20	16.59	1.40
27	work-stealing	–pin-workers –MFSC –CENTRALIZED –RANDOMPRI	20	16.66	1.60
27	work-stealing	–pin-workers –MFSC –PERGROUP –SEQ	20	16.37	0.91
27	work-stealing	–pin-workers –MFSC –PERGROUP –SEQPRI	20	16.48	0.84
27	work-stealing	–pin-workers –MFSC –PERGROUP –RANDOM	20	16.71	0.62
27	work-stealing	–pin-workers –MFSC –PERGROUP –RANDOMPRI	20	16.24	0.48
27	work-stealing	–pin-workers –MFSC –PERCPU –SEQ	20	16.10	1.15
27	work-stealing	–pin-workers –MFSC –PERCPU –SEQPRI	20	16.11	0.98
27	work-stealing	–pin-workers –MFSC –PERCPU –RANDOM	20	16.44	0.96
27	work-stealing	–pin-workers –MFSC –PERCPU –RANDOMPRI	20	16.32	1.21
27	work-stealing	–pin-workers –PSS –CENTRALIZED –SEQ	20	17.01	1.35
27	work-stealing	–pin-workers –PSS –CENTRALIZED –SEQPRI	20	17.08	1.24
27	work-stealing	–pin-workers –PSS –CENTRALIZED –RANDOM	20	17.79	1.51
27	work-stealing	–pin-workers –PSS –CENTRALIZED –RANDOMPRI	20	17.72	1.60
27	work-stealing	–pin-workers –PSS –PERGROUP –SEQ	20	17.89	0.88
27	work-stealing	–pin-workers –PSS –PERGROUP –SEQPRI	20	17.99	1.02
27	work-stealing	–pin-workers –PSS –PERGROUP –RANDOM	20	17.80	1.12
27	work-stealing	–pin-workers –PSS –PERGROUP –RANDOMPRI	20	18.03	0.76
27	work-stealing	–pin-workers –PSS –PERCPU –SEQ	20	16.98	0.96
27	work-stealing	–pin-workers –PSS –PERCPU –SEQPRI	20	16.53	0.88
27	work-stealing	–pin-workers –PSS –PERCPU –RANDOM	20	17.43	0.86
27	work-stealing	–pin-workers –PSS –PERCPU –RANDOMPRI	20	17.21	0.72
1	hierarchical	–pin-workers –PERGROUP –STATIC	20	13.86	1.11
1	hierarchical	–pin-workers –PERGROUP –GSS	20	13.40	1.16
1	hierarchical	–pin-workers –PERGROUP –TSS	20	12.72	1.20
1	hierarchical	–pin-workers –PERGROUP –FAC2	20	13.33	1.16
1	hierarchical	–pin-workers –PERGROUP –TFSS	20	13.15	1.18
1	hierarchical	–pin-workers –PERGROUP –FISS	20	13.14	1.16
1	hierarchical	–pin-workers –PERGROUP –VISS	20	11.89	1.11
1	hierarchical	–pin-workers –PERGROUP –PLS	20	12.78	1.00
1	hierarchical	–pin-workers –PERGROUP –MSTATIC	20	12.17	1.00
1	hierarchical	–pin-workers –PERGROUP –MFSC	20	11.73	1.05
1	hierarchical	–pin-workers –PERGROUP –PSS	20	12.93	1.22
27	hierarchical	–pin-workers –PERGROUP –STATIC	20	17.27	0.77
27	hierarchical	–pin-workers –PERGROUP –GSS	20	16.72	0.66
27	hierarchical	–pin-workers –PERGROUP –TSS	20	16.57	0.82
27	hierarchical	–pin-workers –PERGROUP –FAC2	20	16.48	0.61
27	hierarchical	–pin-workers –PERGROUP –TFSS	20	16.56	0.68
27	hierarchical	–pin-workers –PERGROUP –FISS	20	16.74	0.85
27	hierarchical	–pin-workers –PERGROUP –VISS	20	16.19	0.85
27	hierarchical	–pin-workers –PERGROUP –PLS	20	16.44	1.10
27	hierarchical	–pin-workers –PERGROUP –MSTATIC	20	15.91	0.74
27	hierarchical	–pin-workers –PERGROUP –MFSC	20	15.95	0.64
27	hierarchical	–pin-workers –PERGROUP –PSS	20	16.70	0.76
27	work-stealing	–pin-workers –pre-partition –STATIC –CENTRALIZED –SEQ	20	18.25	1.62
27	work-stealing	–pin-workers –pre-partition –STATIC –CENTRALIZED –SEQPRI	20	18.66	1.15
27	work-stealing	–pin-workers –pre-partition –STATIC –CENTRALIZED –RANDOM	20	18.46	1.98
27	work-stealing	–pin-workers –pre-partition –STATIC –CENTRALIZED –RANDOMPRI	20	18.22	1.59
27	work-stealing	–pin-workers –pre-partition –STATIC –PERGROUP –SEQ	20	16.92	0.84
27	work-stealing	–pin-workers –pre-partition –STATIC –PERGROUP –SEQPRI	20	17.52	0.84
27	work-stealing	–pin-workers –pre-partition –STATIC –PERGROUP –RANDOM	20	17.33	0.74
27	work-stealing	–pin-workers –pre-partition –STATIC –PERGROUP –RANDOMPRI	20	17.28	0.87
27	work-stealing	–pin-workers –pre-partition –STATIC –PERCPU –SEQ	20	17.50	1.03
27	work-stealing	–pin-workers –pre-partition –STATIC –PERCPU –SEQPRI	20	17.86	1.01
27	work-stealing	–pin-workers –pre-partition –STATIC –PERCPU –RANDOM	20	17.93	0.94
27	work-stealing	–pin-workers –pre-partition –STATIC –PERCPU –RANDOMPRI	20	18.14	0.81
27	work-stealing	–pin-workers –pre-partition –GSS –CENTRALIZED –SEQ	20	17.46	1.35
27	work-stealing	–pin-workers –pre-partition –GSS –CENTRALIZED –SEQPRI	20	17.60	1.31
27	work-stealing	–pin-workers –pre-partition –GSS –CENTRALIZED –RANDOM	20	17.85	1.50

27	work-stealing	–pin-workers –pre-partition –GSS –CENTRALIZED –RANDOMPRI	20	17.72	1.49
27	work-stealing	–pin-workers –pre-partition –GSS –PERGROUP –SEQ	20	16.85	0.92
27	work-stealing	–pin-workers –pre-partition –GSS –PERGROUP –SEQPRI	20	16.89	0.82
27	work-stealing	–pin-workers –pre-partition –GSS –PERGROUP –RANDOM	20	16.87	0.92
27	work-stealing	–pin-workers –pre-partition –GSS –PERGROUP –RANDOMPRI	20	16.45	0.78
27	work-stealing	–pin-workers –pre-partition –GSS –PERCPU –SEQ	20	19.60	0.92
27	work-stealing	–pin-workers –pre-partition –GSS –PERCPU –SEQPRI	20	19.06	1.00
27	work-stealing	–pin-workers –pre-partition –GSS –PERCPU –RANDOM	20	19.41	1.13
27	work-stealing	–pin-workers –pre-partition –GSS –PERCPU –RANDOMPRI	20	19.22	0.96
27	work-stealing	–pin-workers –pre-partition –TSS –CENTRALIZED –SEQ	20	17.13	1.43
27	work-stealing	–pin-workers –pre-partition –TSS –CENTRALIZED –SEQPRI	20	16.84	1.52
27	work-stealing	–pin-workers –pre-partition –TSS –CENTRALIZED –RANDOM	20	17.07	1.38
27	work-stealing	–pin-workers –pre-partition –TSS –CENTRALIZED –RANDOMPRI	20	16.63	1.10
27	work-stealing	–pin-workers –pre-partition –TSS –PERGROUP –SEQ	20	16.26	0.71
27	work-stealing	–pin-workers –pre-partition –TSS –PERGROUP –SEQPRI	20	16.12	0.65
27	work-stealing	–pin-workers –pre-partition –TSS –PERGROUP –RANDOM	20	16.46	0.94
27	work-stealing	–pin-workers –pre-partition –TSS –PERGROUP –RANDOMPRI	20	16.45	0.78
27	work-stealing	–pin-workers –pre-partition –TSS –PERCPU –SEQ	20	18.11	0.75
27	work-stealing	–pin-workers –pre-partition –TSS –PERCPU –SEQPRI	20	18.17	1.15
27	work-stealing	–pin-workers –pre-partition –TSS –PERCPU –RANDOM	20	18.35	1.13
27	work-stealing	–pin-workers –pre-partition –TSS –PERCPU –RANDOMPRI	20	18.59	0.96
27	work-stealing	–pin-workers –pre-partition –FAC2 –CENTRALIZED –SEQ	20	17.31	1.49
27	work-stealing	–pin-workers –pre-partition –FAC2 –CENTRALIZED –SEQPRI	20	17.40	1.52
27	work-stealing	–pin-workers –pre-partition –FAC2 –CENTRALIZED –RANDOM	20	17.32	1.05
27	work-stealing	–pin-workers –pre-partition –FAC2 –CENTRALIZED –RANDOMPRI	20	17.22	1.26
27	work-stealing	–pin-workers –pre-partition –FAC2 –PERGROUP –SEQ	20	16.18	0.69
27	work-stealing	–pin-workers –pre-partition –FAC2 –PERGROUP –SEQPRI	20	16.38	1.11
27	work-stealing	–pin-workers –pre-partition –FAC2 –PERGROUP –RANDOM	20	16.62	0.92
27	work-stealing	–pin-workers –pre-partition –FAC2 –PERGROUP –RANDOMPRI	20	16.31	0.93
27	work-stealing	–pin-workers –pre-partition –FAC2 –PERCPU –SEQ	20	19.92	0.76
27	work-stealing	–pin-workers –pre-partition –FAC2 –PERCPU –SEQPRI	20	19.78	1.01
27	work-stealing	–pin-workers –pre-partition –FAC2 –PERCPU –RANDOM	20	20.46	0.86
27	work-stealing	–pin-workers –pre-partition –FAC2 –PERCPU –RANDOMPRI	20	20.49	1.10
27	work-stealing	–pin-workers –pre-partition –TFSS –CENTRALIZED –SEQ	20	17.16	1.13
27	work-stealing	–pin-workers –pre-partition –TFSS –CENTRALIZED –SEQPRI	20	17.04	1.35
27	work-stealing	–pin-workers –pre-partition –TFSS –CENTRALIZED –RANDOM	20	17.34	1.29
27	work-stealing	–pin-workers –pre-partition –TFSS –CENTRALIZED –RANDOMPRI	20	16.75	1.15
27	work-stealing	–pin-workers –pre-partition –TFSS –PERGROUP –SEQ	20	16.09	0.84
27	work-stealing	–pin-workers –pre-partition –TFSS –PERGROUP –SEQPRI	20	16.65	0.83
27	work-stealing	–pin-workers –pre-partition –TFSS –PERGROUP –RANDOM	20	16.55	0.71
27	work-stealing	–pin-workers –pre-partition –TFSS –PERGROUP –RANDOMPRI	20	16.41	0.94
27	work-stealing	–pin-workers –pre-partition –TFSS –PERCPU –SEQ	20	21.13	1.08
27	work-stealing	–pin-workers –pre-partition –TFSS –PERCPU –SEQPRI	20	20.96	1.25
27	work-stealing	–pin-workers –pre-partition –TFSS –PERCPU –RANDOM	20	21.18	1.22
27	work-stealing	–pin-workers –pre-partition –TFSS –PERCPU –RANDOMPRI	20	21.25	1.08
27	work-stealing	–pin-workers –pre-partition –FISS –CENTRALIZED –SEQ	20	17.36	1.62
27	work-stealing	–pin-workers –pre-partition –FISS –CENTRALIZED –SEQPRI	20	17.29	1.12
27	work-stealing	–pin-workers –pre-partition –FISS –CENTRALIZED –RANDOM	20	17.03	1.35
27	work-stealing	–pin-workers –pre-partition –FISS –CENTRALIZED –RANDOMPRI	20	17.27	1.34
27	work-stealing	–pin-workers –pre-partition –FISS –PERGROUP –SEQ	20	16.84	0.88
27	work-stealing	–pin-workers –pre-partition –FISS –PERGROUP –SEQPRI	20	16.97	1.04
27	work-stealing	–pin-workers –pre-partition –FISS –PERGROUP –RANDOM	20	17.09	0.59
27	work-stealing	–pin-workers –pre-partition –FISS –PERGROUP –RANDOMPRI	20	16.57	0.89
27	work-stealing	–pin-workers –pre-partition –FISS –PERCPU –SEQ	20	17.98	0.80
27	work-stealing	–pin-workers –pre-partition –FISS –PERCPU –SEQPRI	20	18.04	1.11
27	work-stealing	–pin-workers –pre-partition –FISS –PERCPU –RANDOM	20	18.09	0.85
27	work-stealing	–pin-workers –pre-partition –FISS –PERCPU –RANDOMPRI	20	18.12	0.85
27	work-stealing	–pin-workers –pre-partition –VISS –CENTRALIZED –SEQ	20	16.74	1.07
27	work-stealing	–pin-workers –pre-partition –VISS –CENTRALIZED –SEQPRI	20	16.65	1.20
27	work-stealing	–pin-workers –pre-partition –VISS –CENTRALIZED –RANDOM	20	16.63	1.28
27	work-stealing	–pin-workers –pre-partition –VISS –CENTRALIZED –RANDOMPRI	20	16.88	1.68
27	work-stealing	–pin-workers –pre-partition –VISS –PERGROUP –SEQ	20	16.27	0.68
27	work-stealing	–pin-workers –pre-partition –VISS –PERGROUP –SEQPRI	20	16.54	0.69
27	work-stealing	–pin-workers –pre-partition –VISS –PERGROUP –RANDOM	20	16.24	0.67
27	work-stealing	–pin-workers –pre-partition –VISS –PERGROUP –RANDOMPRI	20	16.33	0.94
27	work-stealing	–pin-workers –pre-partition –VISS –PERCPU –SEQ	20	18.84	1.04
27	work-stealing	–pin-workers –pre-partition –VISS –PERCPU –SEQPRI	20	18.44	0.69
27	work-stealing	–pin-workers –pre-partition –VISS –PERCPU –RANDOM	20	19.15	0.89
27	work-stealing	–pin-workers –pre-partition –VISS –PERCPU –RANDOMPRI	20	19.23	1.07
27	work-stealing	–pin-workers –pre-partition –PLS –CENTRALIZED –SEQ	20	17.28	1.16
27	work-stealing	–pin-workers –pre-partition –PLS –CENTRALIZED –SEQPRI	20	17.29	1.28
27	work-stealing	–pin-workers –pre-partition –PLS –CENTRALIZED –RANDOM	20	17.47	1.40
27	work-stealing	–pin-workers –pre-partition –PLS –CENTRALIZED –RANDOMPRI	20	17.21	1.18
27	work-stealing	–pin-workers –pre-partition –PLS –PERGROUP –SEQ	20	16.68	0.66
27	work-stealing	–pin-workers –pre-partition –PLS –PERGROUP –SEQPRI	20	16.56	0.77
27	work-stealing	–pin-workers –pre-partition –PLS –PERGROUP –RANDOM	20	16.68	0.87
27	work-stealing	–pin-workers –pre-partition –PLS –PERGROUP –RANDOMPRI	20	16.75	0.82
27	work-stealing	–pin-workers –pre-partition –PLS –PERCPU –SEQ	20	19.54	1.03
27	work-stealing	–pin-workers –pre-partition –PLS –PERCPU –SEQPRI	20	19.79	0.96
27	work-stealing	–pin-workers –pre-partition –PLS –PERCPU –RANDOM	20	19.66	1.08
27	work-stealing	–pin-workers –pre-partition –PLS –PERCPU –RANDOMPRI	20	19.76	0.97
27	work-stealing	–pin-workers –pre-partition –MSTATIC –CENTRALIZED –SEQ	20	17.06	1.39
27	work-stealing	–pin-workers –pre-partition –MSTATIC –CENTRALIZED –SEQPRI	20	17.46	1.60
27	work-stealing	–pin-workers –pre-partition –MSTATIC –CENTRALIZED –RANDOM	20	16.93	1.39

27	work-stealing	–pin-workers –pre-partition –MSTATIC –CENTRALIZED –RANDOMPRI	20	16.98	1.49
27	work-stealing	–pin-workers –pre-partition –MSTATIC –PERGROUP –SEQ	20	16.37	0.78
27	work-stealing	–pin-workers –pre-partition –MSTATIC –PERGROUP –SEQPRI	20	16.47	0.87
27	work-stealing	–pin-workers –pre-partition –MSTATIC –PERGROUP –RANDOM	20	16.27	0.79
27	work-stealing	–pin-workers –pre-partition –MSTATIC –PERGROUP –RANDOMPRI	20	16.56	1.02
27	work-stealing	–pin-workers –pre-partition –MSTATIC –PERCPU –SEQ	20	18.72	0.87
27	work-stealing	–pin-workers –pre-partition –MSTATIC –PERCPU –SEQPRI	20	18.54	0.85
27	work-stealing	–pin-workers –pre-partition –MSTATIC –PERCPU –RANDOM	20	18.79	0.91
27	work-stealing	–pin-workers –pre-partition –MSTATIC –PERCPU –RANDOMPRI	20	18.80	1.08
27	work-stealing	–pin-workers –pre-partition –MFSC –CENTRALIZED –SEQ	20	16.46	1.18
27	work-stealing	–pin-workers –pre-partition –MFSC –CENTRALIZED –SEQPRI	20	16.46	1.23
27	work-stealing	–pin-workers –pre-partition –MFSC –CENTRALIZED –RANDOM	20	16.75	1.36
27	work-stealing	–pin-workers –pre-partition –MFSC –CENTRALIZED –RANDOMPRI	20	16.43	1.01
27	work-stealing	–pin-workers –pre-partition –MFSC –PERGROUP –SEQ	20	16.52	0.63
27	work-stealing	–pin-workers –pre-partition –MFSC –PERGROUP –SEQPRI	20	16.59	0.96
27	work-stealing	–pin-workers –pre-partition –MFSC –PERGROUP –RANDOM	20	16.59	1.14
27	work-stealing	–pin-workers –pre-partition –MFSC –PERGROUP –RANDOMPRI	20	16.41	0.80
27	work-stealing	–pin-workers –pre-partition –MFSC –PERCPU –SEQ	20	21.03	0.78
27	work-stealing	–pin-workers –pre-partition –MFSC –PERCPU –SEQPRI	20	20.73	0.97
27	work-stealing	–pin-workers –pre-partition –MFSC –PERCPU –RANDOM	20	20.95	1.12
27	work-stealing	–pin-workers –pre-partition –MFSC –PERCPU –RANDOMPRI	20	21.24	1.01
27	work-stealing	–pin-workers –pre-partition –PSS –CENTRALIZED –SEQ	20	17.24	1.24
27	work-stealing	–pin-workers –pre-partition –PSS –CENTRALIZED –SEQPRI	20	17.24	1.32
27	work-stealing	–pin-workers –pre-partition –PSS –CENTRALIZED –RANDOM	20	17.06	1.48
27	work-stealing	–pin-workers –pre-partition –PSS –CENTRALIZED –RANDOMPRI	20	16.63	1.03
27	work-stealing	–pin-workers –pre-partition –PSS –PERGROUP –SEQ	20	16.64	0.98
27	work-stealing	–pin-workers –pre-partition –PSS –PERGROUP –SEQPRI	20	16.41	0.64
27	work-stealing	–pin-workers –pre-partition –PSS –PERGROUP –RANDOM	20	16.92	0.76
27	work-stealing	–pin-workers –pre-partition –PSS –PERGROUP –RANDOMPRI	20	16.63	0.96
27	work-stealing	–pin-workers –pre-partition –PSS –PERCPU –SEQ	20	20.49	1.10
27	work-stealing	–pin-workers –pre-partition –PSS –PERCPU –SEQPRI	20	20.45	1.06
27	work-stealing	–pin-workers –pre-partition –PSS –PERCPU –RANDOM	20	20.84	0.87
27	work-stealing	–pin-workers –pre-partition –PSS –PERCPU –RANDOMPRI	20	21.10	1.02
1	work-stealing	–pin-workers –pre-partition –STATIC –CENTRALIZED –SEQ	20	13.26	0.98
1	work-stealing	–pin-workers –pre-partition –STATIC –CENTRALIZED –SEQPRI	20	13.49	1.26
1	work-stealing	–pin-workers –pre-partition –STATIC –CENTRALIZED –RANDOM	20	13.19	1.06
1	work-stealing	–pin-workers –pre-partition –STATIC –CENTRALIZED –RANDOMPRI	20	13.33	1.21
1	work-stealing	–pin-workers –pre-partition –STATIC –PERGROUP –SEQ	20	13.24	1.22
1	work-stealing	–pin-workers –pre-partition –STATIC –PERGROUP –SEQPRI	20	13.45	1.34
1	work-stealing	–pin-workers –pre-partition –STATIC –PERGROUP –RANDOM	20	13.52	1.30
1	work-stealing	–pin-workers –pre-partition –STATIC –PERGROUP –RANDOMPRI	20	12.70	0.92
1	work-stealing	–pin-workers –pre-partition –STATIC –PERCPU –SEQ	20	12.63	1.03
1	work-stealing	–pin-workers –pre-partition –STATIC –PERCPU –SEQPRI	20	12.39	1.01
1	work-stealing	–pin-workers –pre-partition –STATIC –PERCPU –RANDOM	20	12.92	1.24
1	work-stealing	–pin-workers –pre-partition –STATIC –PERCPU –RANDOMPRI	20	12.26	0.67
1	work-stealing	–pin-workers –pre-partition –GSS –CENTRALIZED –SEQ	20	12.47	1.05
1	work-stealing	–pin-workers –pre-partition –GSS –CENTRALIZED –SEQPRI	20	12.59	1.27
1	work-stealing	–pin-workers –pre-partition –GSS –CENTRALIZED –RANDOM	20	12.40	0.81
1	work-stealing	–pin-workers –pre-partition –GSS –CENTRALIZED –RANDOMPRI	20	12.71	1.14
1	work-stealing	–pin-workers –pre-partition –GSS –PERGROUP –SEQ	20	11.89	1.11
1	work-stealing	–pin-workers –pre-partition –GSS –PERGROUP –SEQPRI	20	11.97	0.81
1	work-stealing	–pin-workers –pre-partition –GSS –PERGROUP –RANDOM	20	12.18	1.33
1	work-stealing	–pin-workers –pre-partition –GSS –PERGROUP –RANDOMPRI	20	12.02	1.05
1	work-stealing	–pin-workers –pre-partition –GSS –PERCPU –SEQ	20	12.32	0.82
1	work-stealing	–pin-workers –pre-partition –GSS –PERCPU –SEQPRI	20	12.34	1.07
1	work-stealing	–pin-workers –pre-partition –GSS –PERCPU –RANDOM	20	12.48	1.09
1	work-stealing	–pin-workers –pre-partition –GSS –PERCPU –RANDOMPRI	20	12.47	1.03
1	work-stealing	–pin-workers –pre-partition –TSS –CENTRALIZED –SEQ	20	12.67	1.19
1	work-stealing	–pin-workers –pre-partition –TSS –CENTRALIZED –SEQPRI	20	12.53	1.21
1	work-stealing	–pin-workers –pre-partition –TSS –CENTRALIZED –RANDOM	20	12.62	1.17
1	work-stealing	–pin-workers –pre-partition –TSS –CENTRALIZED –RANDOMPRI	20	12.76	1.21
1	work-stealing	–pin-workers –pre-partition –TSS –PERGROUP –SEQ	20	11.63	0.98
1	work-stealing	–pin-workers –pre-partition –TSS –PERGROUP –SEQPRI	20	11.50	0.89
1	work-stealing	–pin-workers –pre-partition –TSS –PERGROUP –RANDOM	20	11.85	1.16
1	work-stealing	–pin-workers –pre-partition –TSS –PERGROUP –RANDOMPRI	20	11.99	1.25
1	work-stealing	–pin-workers –pre-partition –TSS –PERCPU –SEQ	20	12.71	1.33
1	work-stealing	–pin-workers –pre-partition –TSS –PERCPU –SEQPRI	20	13.11	1.58
1	work-stealing	–pin-workers –pre-partition –TSS –PERCPU –RANDOM	20	12.89	1.04
1	work-stealing	–pin-workers –pre-partition –TSS –PERCPU –RANDOMPRI	20	12.96	1.19
1	work-stealing	–pin-workers –pre-partition –FAC2 –CENTRALIZED –SEQ	20	12.52	1.06
1	work-stealing	–pin-workers –pre-partition –FAC2 –CENTRALIZED –SEQPRI	20	12.10	1.02
1	work-stealing	–pin-workers –pre-partition –FAC2 –CENTRALIZED –RANDOM	20	13.10	1.29
1	work-stealing	–pin-workers –pre-partition –FAC2 –CENTRALIZED –RANDOMPRI	20	12.62	1.18
1	work-stealing	–pin-workers –pre-partition –FAC2 –PERGROUP –SEQ	20	11.66	1.12
1	work-stealing	–pin-workers –pre-partition –FAC2 –PERGROUP –SEQPRI	20	11.89	1.15
1	work-stealing	–pin-workers –pre-partition –FAC2 –PERGROUP –RANDOM	20	11.56	1.18
1	work-stealing	–pin-workers –pre-partition –FAC2 –PERGROUP –RANDOMPRI	20	11.51	0.97
1	work-stealing	–pin-workers –pre-partition –FAC2 –PERCPU –SEQ	20	13.45	1.54
1	work-stealing	–pin-workers –pre-partition –FAC2 –PERCPU –SEQPRI	20	12.98	1.34
1	work-stealing	–pin-workers –pre-partition –FAC2 –PERCPU –RANDOM	20	12.87	1.34
1	work-stealing	–pin-workers –pre-partition –FAC2 –PERCPU –RANDOMPRI	20	12.96	1.35
1	work-stealing	–pin-workers –pre-partition –TFSS –CENTRALIZED –SEQ	20	12.69	1.33
1	work-stealing	–pin-workers –pre-partition –TFSS –CENTRALIZED –SEQPRI	20	12.73	1.26
1	work-stealing	–pin-workers –pre-partition –TFSS –CENTRALIZED –RANDOM	20	12.60	1.28

1	work-stealing	–pin-workers –pre-partition –TFSS –CENTRALIZED –RANDOMPRI	20	12.15	1.09
1	work-stealing	–pin-workers –pre-partition –TFSS –PERGROUP –SEQ	20	11.60	1.15
1	work-stealing	–pin-workers –pre-partition –TFSS –PERGROUP –SEQPRI	20	11.40	1.04
1	work-stealing	–pin-workers –pre-partition –TFSS –PERGROUP –RANDOM	20	11.80	1.17
1	work-stealing	–pin-workers –pre-partition –TFSS –PERGROUP –RANDOMPRI	20	11.22	0.93
1	work-stealing	–pin-workers –pre-partition –TFSS –PERCPU –SEQ	20	13.20	1.50
1	work-stealing	–pin-workers –pre-partition –TFSS –PERCPU –SEQPRI	20	13.11	0.90
1	work-stealing	–pin-workers –pre-partition –TFSS –PERCPU –RANDOM	20	12.93	1.29
1	work-stealing	–pin-workers –pre-partition –TFSS –PERCPU –RANDOMPRI	20	13.49	1.12
1	work-stealing	–pin-workers –pre-partition –FISS –CENTRALIZED –SEQ	20	13.59	1.02
1	work-stealing	–pin-workers –pre-partition –FISS –CENTRALIZED –SEQPRI	20	13.63	0.97
1	work-stealing	–pin-workers –pre-partition –FISS –CENTRALIZED –RANDOM	20	13.83	1.02
1	work-stealing	–pin-workers –pre-partition –FISS –CENTRALIZED –RANDOMPRI	20	13.14	0.48
1	work-stealing	–pin-workers –pre-partition –FISS –PERGROUP –SEQ	20	12.50	0.92
1	work-stealing	–pin-workers –pre-partition –FISS –PERGROUP –SEQPRI	20	12.72	1.02
1	work-stealing	–pin-workers –pre-partition –FISS –PERGROUP –RANDOM	20	12.42	0.90
1	work-stealing	–pin-workers –pre-partition –FISS –PERGROUP –RANDOMPRI	20	12.37	0.63
1	work-stealing	–pin-workers –pre-partition –FISS –PERCPU –SEQ	20	12.56	0.22
1	work-stealing	–pin-workers –pre-partition –FISS –PERCPU –SEQPRI	20	13.23	1.10
1	work-stealing	–pin-workers –pre-partition –FISS –PERCPU –RANDOM	20	13.60	1.33
1	work-stealing	–pin-workers –pre-partition –FISS –PERCPU –RANDOMPRI	20	13.45	1.29
1	work-stealing	–pin-workers –pre-partition –VISS –CENTRALIZED –SEQ	20	12.10	0.94
1	work-stealing	–pin-workers –pre-partition –VISS –CENTRALIZED –SEQPRI	20	12.21	0.94
1	work-stealing	–pin-workers –pre-partition –VISS –CENTRALIZED –RANDOM	20	12.36	1.15
1	work-stealing	–pin-workers –pre-partition –VISS –CENTRALIZED –RANDOMPRI	20	12.43	1.14
1	work-stealing	–pin-workers –pre-partition –VISS –PERGROUP –SEQ	20	11.59	1.06
1	work-stealing	–pin-workers –pre-partition –VISS –PERGROUP –SEQPRI	20	11.61	1.05
1	work-stealing	–pin-workers –pre-partition –VISS –PERGROUP –RANDOM	20	11.56	1.13
1	work-stealing	–pin-workers –pre-partition –VISS –PERGROUP –RANDOMPRI	20	11.33	0.99
1	work-stealing	–pin-workers –pre-partition –VISS –PERCPU –SEQ	20	13.27	1.10
1	work-stealing	–pin-workers –pre-partition –VISS –PERCPU –SEQPRI	20	12.60	0.18
1	work-stealing	–pin-workers –pre-partition –VISS –PERCPU –RANDOM	20	13.20	1.01
1	work-stealing	–pin-workers –pre-partition –VISS –PERCPU –RANDOMPRI	20	13.21	1.05
1	work-stealing	–pin-workers –pre-partition –PLS –CENTRALIZED –SEQ	20	13.11	1.17
1	work-stealing	–pin-workers –pre-partition –PLS –CENTRALIZED –SEQPRI	20	13.18	1.22
1	work-stealing	–pin-workers –pre-partition –PLS –CENTRALIZED –RANDOM	20	12.48	0.55
1	work-stealing	–pin-workers –pre-partition –PLS –CENTRALIZED –RANDOMPRI	20	12.95	1.07
1	work-stealing	–pin-workers –pre-partition –PLS –PERGROUP –SEQ	20	12.71	1.20
1	work-stealing	–pin-workers –pre-partition –PLS –PERGROUP –SEQPRI	20	11.89	0.74
1	work-stealing	–pin-workers –pre-partition –PLS –PERGROUP –RANDOM	20	11.94	0.84
1	work-stealing	–pin-workers –pre-partition –PLS –PERGROUP –RANDOMPRI	20	12.37	1.22
1	work-stealing	–pin-workers –pre-partition –PLS –PERCPU –SEQ	20	13.53	1.14
1	work-stealing	–pin-workers –pre-partition –PLS –PERCPU –SEQPRI	20	13.79	1.22
1	work-stealing	–pin-workers –pre-partition –PLS –PERCPU –RANDOM	20	13.67	1.30
1	work-stealing	–pin-workers –pre-partition –PLS –PERCPU –RANDOMPRI	20	13.00	0.83
1	work-stealing	–pin-workers –pre-partition –MSTATIC –CENTRALIZED –SEQ	20	13.45	1.21
1	work-stealing	–pin-workers –pre-partition –MSTATIC –CENTRALIZED –SEQPRI	20	13.45	1.14
1	work-stealing	–pin-workers –pre-partition –MSTATIC –CENTRALIZED –RANDOM	20	13.41	1.22
1	work-stealing	–pin-workers –pre-partition –MSTATIC –CENTRALIZED –RANDOMPRI	20	13.16	1.09
1	work-stealing	–pin-workers –pre-partition –MSTATIC –PERGROUP –SEQ	20	12.32	1.24
1	work-stealing	–pin-workers –pre-partition –MSTATIC –PERGROUP –SEQPRI	20	11.93	0.93
1	work-stealing	–pin-workers –pre-partition –MSTATIC –PERGROUP –RANDOM	20	12.20	1.15
1	work-stealing	–pin-workers –pre-partition –MSTATIC –PERGROUP –RANDOMPRI	20	11.94	0.98
1	work-stealing	–pin-workers –pre-partition –MSTATIC –PERCPU –SEQ	20	13.14	1.05
1	work-stealing	–pin-workers –pre-partition –MSTATIC –PERCPU –SEQPRI	20	13.12	0.92
1	work-stealing	–pin-workers –pre-partition –MSTATIC –PERCPU –RANDOM	20	13.15	1.04
1	work-stealing	–pin-workers –pre-partition –MSTATIC –PERCPU –RANDOMPRI	20	13.07	0.90
1	work-stealing	–pin-workers –pre-partition –MFSC –CENTRALIZED –SEQ	20	11.39	0.90
1	work-stealing	–pin-workers –pre-partition –MFSC –CENTRALIZED –SEQPRI	20	11.82	1.18
1	work-stealing	–pin-workers –pre-partition –MFSC –CENTRALIZED –RANDOM	20	11.51	1.00
1	work-stealing	–pin-workers –pre-partition –MFSC –CENTRALIZED –RANDOMPRI	20	11.85	1.18
1	work-stealing	–pin-workers –pre-partition –MFSC –PERGROUP –SEQ	20	11.53	1.05
1	work-stealing	–pin-workers –pre-partition –MFSC –PERGROUP –SEQPRI	20	11.23	0.92
1	work-stealing	–pin-workers –pre-partition –MFSC –PERGROUP –RANDOM	20	11.30	1.03
1	work-stealing	–pin-workers –pre-partition –MFSC –PERGROUP –RANDOMPRI	20	11.23	0.92
1	work-stealing	–pin-workers –pre-partition –MFSC –PERCPU –SEQ	20	13.24	0.85
1	work-stealing	–pin-workers –pre-partition –MFSC –PERCPU –SEQPRI	20	13.48	1.18
1	work-stealing	–pin-workers –pre-partition –MFSC –PERCPU –RANDOM	20	13.97	1.37
1	work-stealing	–pin-workers –pre-partition –MFSC –PERCPU –RANDOMPRI	20	14.11	1.19
1	work-stealing	–pin-workers –pre-partition –PSS –CENTRALIZED –SEQ	20	13.05	1.19
1	work-stealing	–pin-workers –pre-partition –PSS –CENTRALIZED –SEQPRI	20	12.70	1.01
1	work-stealing	–pin-workers –pre-partition –PSS –CENTRALIZED –RANDOM	20	13.30	1.29
1	work-stealing	–pin-workers –pre-partition –PSS –CENTRALIZED –RANDOMPRI	20	12.56	0.92
1	work-stealing	–pin-workers –pre-partition –PSS –PERGROUP –SEQ	20	12.16	1.04
1	work-stealing	–pin-workers –pre-partition –PSS –PERGROUP –SEQPRI	20	11.73	0.72
1	work-stealing	–pin-workers –pre-partition –PSS –PERGROUP –RANDOM	20	12.61	1.26
1	work-stealing	–pin-workers –pre-partition –PSS –PERGROUP –RANDOMPRI	20	11.80	0.85
1	work-stealing	–pin-workers –pre-partition –PSS –PERCPU –SEQ	20	13.15	0.98
1	work-stealing	–pin-workers –pre-partition –PSS –PERCPU –SEQPRI	20	13.21	1.01
1	work-stealing	–pin-workers –pre-partition –PSS –PERCPU –RANDOM	20	13.24	1.06
1	work-stealing	–pin-workers –pre-partition –PSS –PERCPU –RANDOMPRI	20	13.53	1.01