CSE 539S Final Project Report

Amanda Hua, Funda Atik

Abstract: NAS parallel benchmarks have five kernels, including the Multigrid Poisson solver. This paper inspects three parallel implementations of the Multi-grid benchmark and describes our methodology taken to improve the performance of these implementations. We study three different implementations implemented with various parallel programming frameworks, namely, OpenMP, Cilk, and Fast-Flow. Our main goal is to improve the scalability on a multi-socket machine. The focus, therefore, is on understanding performance bottlenecks. Our analysis focuses on Non-uniform Memory Access (NUMA) behavior which is found to be the main issue preventing scalability. We show that different memory allocation policies and thread affinity settings changes the performance significantly. We use pre-existing code [1] coming from Aaron Handleman (PhD Student in Dr. Angelina Lee's lab at Washington University in St. Louis), and from [2] the Efficient NAS Benchmark Kernels with C++ Parallel Programming's authors.

1. Introduction

The original suggestion for this project was to look into a parallel runtime besides Cilk and code several optimizations, then report on any findings. OpenMP had been the runtime of choice, but then we were provided with existing benchmarks (NPB), and the project switched to performance engineering the Multigrid benchmark. It was suggested that we approached comparing three parallel implementations: Fast-Flow, OpenMP, and Cilk. The Cilk version was derived from the OpenMP implementation. The provided code found that there was room for improvement with the Cilk and OpenMP implementations of this benchmark in comparison to the Fast Flow scalability, so these were the areas of focus of this project.

2. Experimental Setup

2.1. Hardware Settings

We used two multi-socket machines for this project: (1) a Skylake machine with 24 cores, and (2) a LinuxLab machine with 16 cores. Both machines have two sockets that contain half of the total cores. We were able to perform the micro-architectural analysis only on the Skylake machine because the LIKWID tool[5] requires root privileges, and LIKWID and numactl tools were not installed on the LinuxLab machine. Further details of the machines can be found in Table 1. Skylake machine's operating system is Ubuntu 16 with kernel 4.15.0-99-generic and the gcc/g++ version is 7.5. We used CilkPlus vanilla runtime for the CILK implementation and compiled it with clang++ version 6.0.1. We enabled -O3 optimizations. We also disabled hyper-threading. For all scalability figures, we take the average of five runs, and standard deviation is less than 0.2.

Table 1: Characteristics of the machines.

Machine Type/ Machine Detail	Thread(s) per core	Core(s) per socket	Socket(s)	L1d cache	L1i cache	L2 cache	L3 cache
Intel(R) Xeon(R) CPU E5-2630 (LinuxLab)	2	8	2	32KB	32KB	256KB	20480KB
Intel(R) Xeon(R) Gold 6126 (Skylake)	2	12	2	32KB	32KB	1024KB	19712KB

2.2. Benchmark Program

We focused on the Multi-grid benchmark which is part of NAS Parallel Benchmark [6]. Multi-grid benchmark implements an iterative V-cycle multi grid solver. This benchmark essentially takes a randomly propagated three dimensional matrix of N 1's and 0's and runs it through transformations until it hits a threshold, at which point it calls a smoothing function on the matrix to finish the solver. Each transformation is modularized into a method called by the main method sequentially. Parallelization occurs in each transformation method across dimensions of the matrix. In the OpenMP implementation of this benchmark, parallel regions must be dictated before any parallel methods can be called. We are using input class C for our experiments which is the largest input size available in given implementations [1,2]. In class C, the 3D grid is a 512x512x512 grid and the number of iterations is 20. The application reports timing values for three parts which are initialization, benchmark, and verification. We only use the timing value for the benchmark part.

2.3. Analysis Approach

The existing versions of this benchmark running with compact scheduling showed that improvements can be made to the scalability of the OpenMP and Cilk implementations. There would be two different approaches to improve scalability of the OpenMP and Cilk implementations. First, we can look at the code and detect the regions for further optimizations. However, NAS benchmarks are widely used and highly optimized so we expected that there is not enough room for further parallelization. The second approach is to inspect the system level considerations such as NUMA memory allocation and NUMA thread scheduling and experiment with loop scheduling policies, e.g. dynamic vs static loop scheduling. This is the approach that we have selected.

3. Analyzing Scalability of Multigrid Application

In this section we analyze the scalability behaviour of given Multi-Grid (MG) implementations. We use *taskset* utility to bind the threads of an application to a particular range of cpus to minimize the effect of default Linux scheduling.

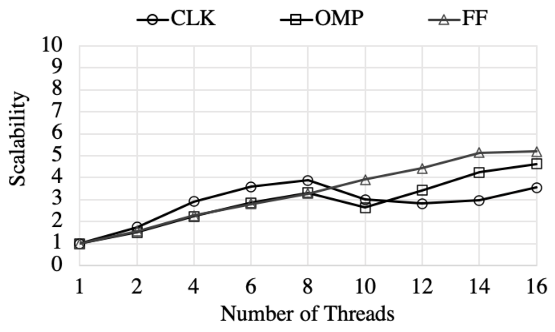


Figure 1: Existing benchmark code's scalability on the LinuxLab machine.

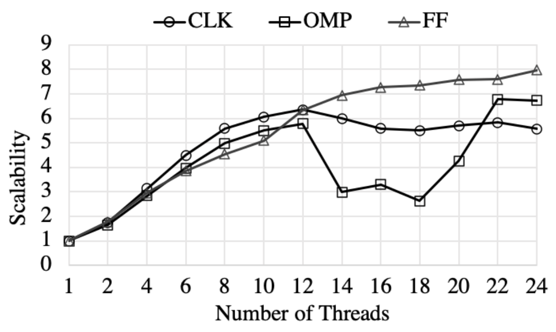


Figure 2: Existing benchmark code's scalability on the Skylake machine.

In both machines, we observed that the performance drops drastically after using the second socket. This is a major problem especially for OpenMP implementation. Figure 1 and Figure 2 suggest that we are observing performance drop due to non uniform memory access (NUMA) effect. To understand the causes of high performance drop observed in OpenMP implementation, we use *perf* tool for a quick analysis. As can be seen in Figure 3, the performance of OpenMP version drops drastically when it is running with 14, 16, 20 threads. The right plot in Figure 3 shows that a number of CPU migrations is very high for these configurations. This also the case for the Cilk and Fast-Flow versions.

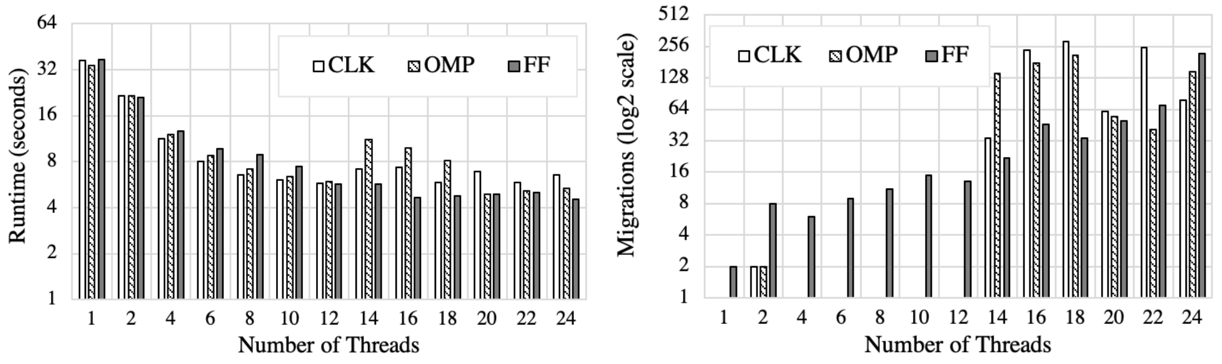


Figure 3: Runtime and a number of CPU migrations if we use taskset

To stabilize the number of migrations across different runs, we use *likwid-pin* tool to pin each thread to a particular core. For OpenMP, we also set the *OMP_PROC_BIND* environment variable to *true*. Figure 4 shows that we eliminated high migrations that we see with OpenMP implementation before. Moreover, the variation in the number of migrations across different parallel implementations also reduces. Changing thread affinity also improves the performance of the OpenMP version considerably when it is running with 14, 16, 20 threads.

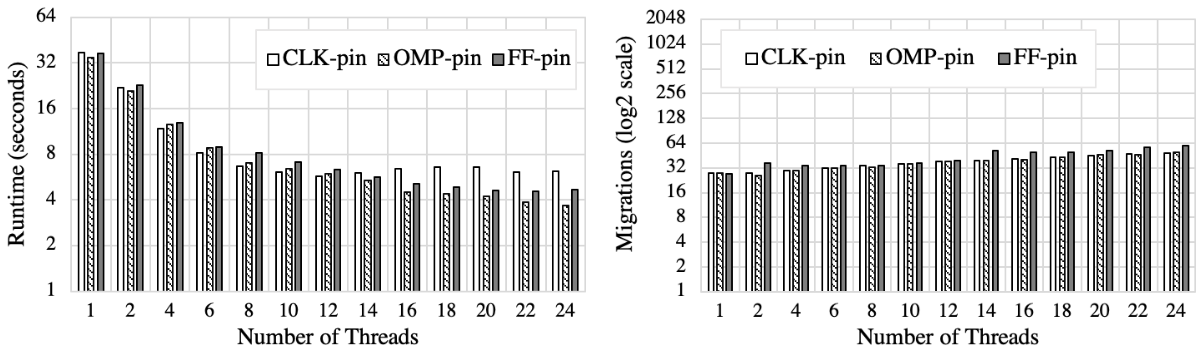


Figure 4: Runtime and a number of CPU migrations if we use likwid-pin.

Figure 5 shows how scalability of the parallel implementations changes when we bind each thread to a particular core instead of binding all threads to a particular cpuset. This change eliminates the performance variabilities that are seen in the OpenMP implementation.

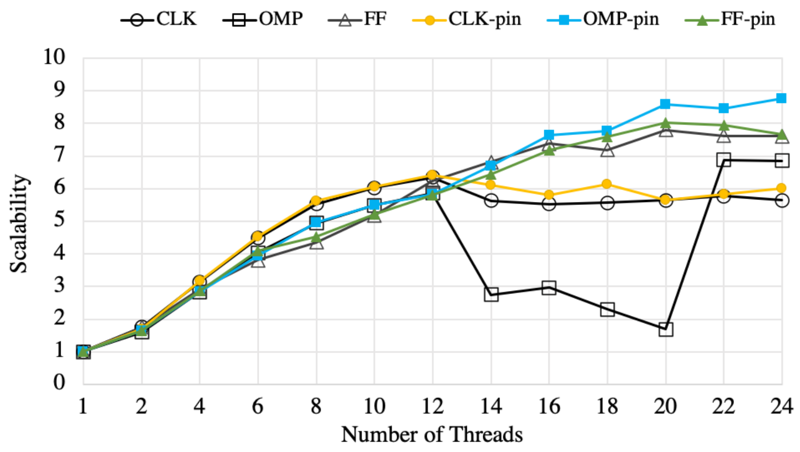


Figure 5: After pinning each thread to a particular core, the OpenMP version outperforms the Cilk and Fast-Flow versions, and high variation in its runtime decreases.

4. Analysis of Scheduling Mechanisms

In this section, we analyze different scheduling policies. We consider two orthogonal mechanisms: (1) loop scheduling for parallel loops and (2) thread scheduling on NUMA sockets. Since we observed that scalability issues are possibly related to NUMA complexities, we analyze the NUMA memory behavior. Mainly, we check for memory data volume read and write data volume for each socket and memory bandwidth. We will use *Likwid-perfctr* tool to have a better understanding of how each parallel implementation leverages the underlying system to get the best performance. Due to the time limitations, we only experimented with the OpenMP version. However, the main difference between the Cilk for loop and OpenMP for loop is how they handle loop scheduling. The Cilk uses dynamic scheduling; whereas the OpenMP and the Fast-Flow use static scheduling in the initial code base. Since we could not use LIKWID with the Cilk runtime for this project, we changed the loop scheduling policy for the OpenMP and the Fast-Flow versions from static to dynamic to understand the impact of loop scheduling on the performance.

4.1. Overview of Loop Scheduling

Scheduling is a way of implementing division of labor in a parallel runtime [3]. Given a number of threads to run a program, runtimes will have a load balancing protocol that dictates which threads gain which tasks, and whether or not work stealing is permitted. Each policy can be configured with multiple chunk sizes.

Dynamic scheduling: The dynamic load balancing protocol states that tasks may be redistributed/accessible to allow an idle worker to retrieve any non-started tasks. This requires coordination between threads to allow for work stealing, or accessing a centralized queue. This protocol is helpful where certain tasks may take longer than others, and there may be one or several idle workers at a time without work stealing.

Static scheduling: The static load balancing protocol distributes tasks evenly among workers and does not require coordination between threads in regards to stealing work or accessing a centralized queue. Workers get their allocated tasks at the beginning of parallelization, and are done when they finish their set of tasks. This protocol is helpful where tasks can be distributed evenly and workers are infrequently idle for long periods of time.

4.2. Overview of Thread Scheduling

There are several ways to schedule threads on multiple NUMA nodes. In this section, we only focus on compact scheduling and scatter (round-robin) scheduling.

Compact scheduling: Initially, we schedule all of the threads in an application in a compact way by keeping all of the threads running on a single physical processor if possible. This strategy works optimally if all of the threads repeatedly access different parts of a large array. All of the cores on the same physical cpus can access the memory banks owned by that processor at the same speed. However, when we fill up all of the cores in the first socket, we need to schedule the remaining threads in the application on the second socket.

Scatter scheduling: We change our CPU binding strategy such that it allows us to bind threads across two sockets evenly when the number of threads is larger than the total number of cores on a single socket. If the threads in the application do not need to access a lot of memory that other threads need, we can effectively double the memory bandwidth and the last level cache size available. However, the trade off is as threads start accessing memory owned by a different processor, the memory latency becomes higher and the interconnect between the sockets may become a bottleneck too.

The rule of thumb is keeping threads of the application as close to the memory in which they are operating as possible. Therefore, the impact of thread scheduling policy on the performance is closely related with how all of the threads in the application access the memory and where the data resides.

4.3. Observations on Scheduling

We look at how much memory bandwidth is achieved and how memory data volume changes as we increase the number of threads for statically and dynamically scheduled loops using compact thread scheduling policy. We expect that after starting to use the second processor, the effective memory bandwidth and cache size are doubled. Since L3 miss ratio values are also very high for all configurations (see Figure 6), we can say that having more bandwidth would have more impact on the MG benchmark than having more cache size.

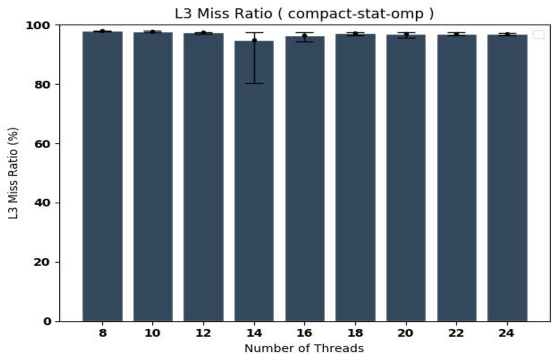
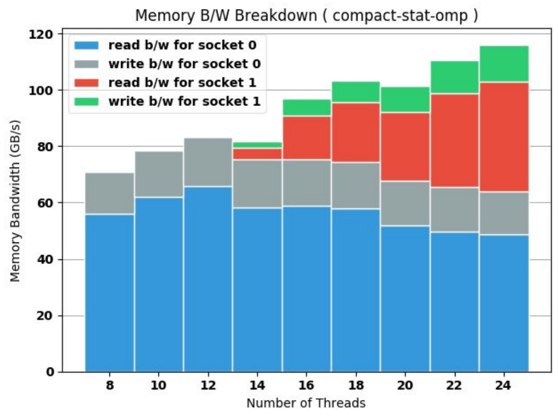
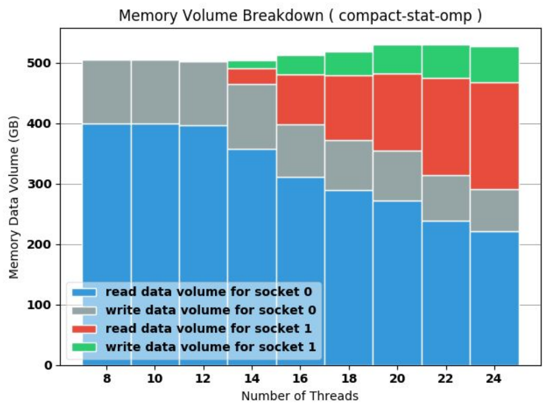


Figure 6: L3 Miss Ratio for OpenMP version with compact scheduling and static loop scheduling.

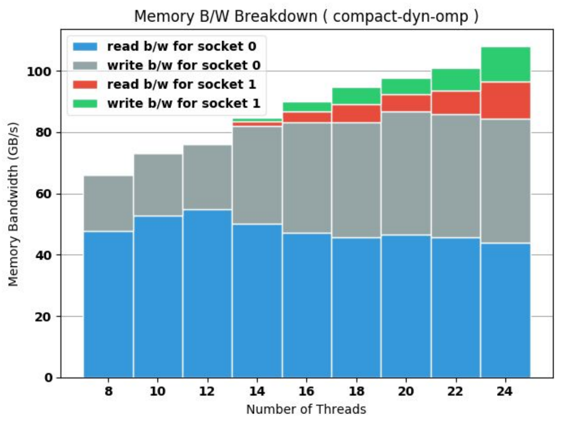
As can be seen in Figure 7, both static and dynamic scheduling with compact scheduling shows higher values for socket 0. For dynamic scheduling case, the memory pressure on socket 0 is much higher compared to static scheduling. Furthermore, utilization of socket 1 memory is very low even with 24 threads for dynamic scheduling case (see Figure 6(c)-(d)). On the other hand, using 24 threads with static scheduling improves utilization of socket 1 memory significantly. Although we were not able to pinpoint the reason behind better NUMA memory utilization across sockets for static scheduling case, we think it is due to the parallel memory allocations during the initialization stage of application. With static scheduling, tasks are distributed across cores on multiple sockets roughly equally. After inspecting memory allocations across NUMA nodes with numastat tool, we believe that the equal distribution of tasks across sockets interacting with first-touch policy is creating a more balanced memory allocation across sockets. Moreover, comparing data transfers from each socket for static and dynamic scheduling cases (Figures 6(b) and 6(d)), we see that the amount of memory transfer from each socket is more balanced with static scheduling when 24 cores are used. Additionally, we also experimented with scatter thread scheduling policy. With scatter thread scheduling policy, we observed that static loop scheduling policy again gives better performance compared to dynamic loops scheduling policy.



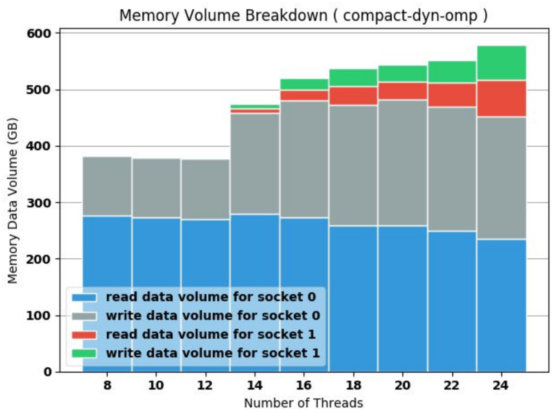
(a) Memory Bandwidth for Static-Compact Setup



(b) Memory Read/Write Volume for Static-Compact Setup



(c) Memory Bandwidth for Dynamic-Compact Setup



(d) Memory Read/Write Volume for Dynamic-Compact Setup

Figure 7: Behavior of NUMA memory with different loop and thread scheduling policies.

5. Effect of NUMA Memory Allocation Policies

Where data resides and where computation occurs have a large impact on the application performance. The main problem that we notice with the different parallel implementations of the MG benchmark is where the initial data is allocated changes according to loop scheduling policy. For instance, if we change all of the parallel loops in OpenMP code from static to dynamic, most of the data is allocated in the memory of the first socket (see Figure 7(d)).

First-Touch Policy: First touch policy requires memory regions to be stacked in one socket until it is full, then moves on to the next socket [4]. This is the default policy for the machines used in this project, and shows higher performance than an interleaving policy for scenarios where threads can fit on one socket.

Interleaving Policy: Interleaving memory policy requires threads to be pinned in an alternating fashion across sockets in the given machine. This policy is helpful in programs that use more memory bandwidth, as using more sockets gives the program access to the L3 cache and memory bandwidth if all of the threads in the program work largely independently. If the threads that are running on multiple sockets need to access the memory in the other sockets frequently, then the interleaving policy gives higher performance than the first-touch policy.

In this section, we combine thread scheduling policies with memory interleaving policies. In these experiments, we use static loop scheduling policy.

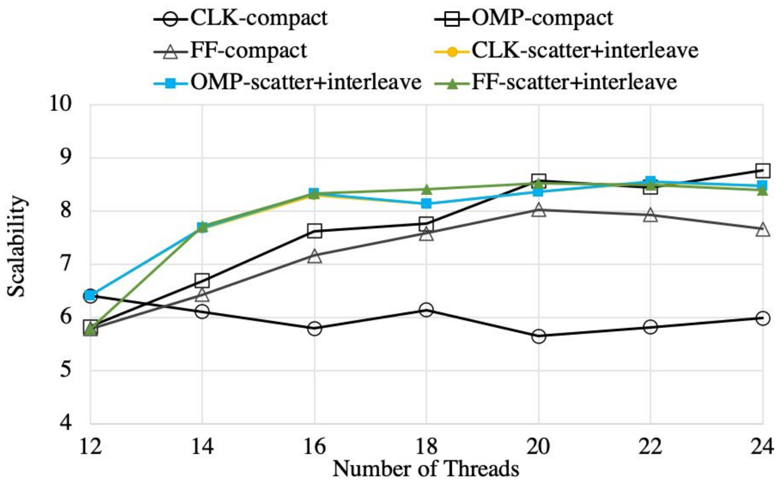
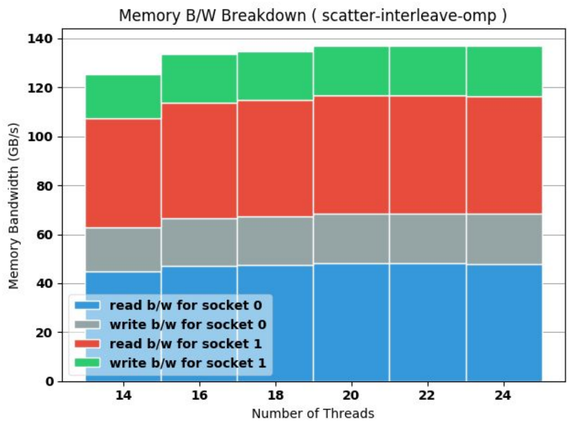
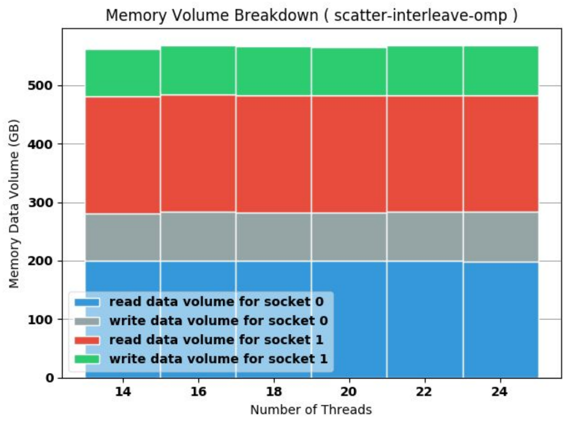


Figure 8: Changing default memory policy to interleaving with scatter scheduling improves performance of all code bases.

Figure 8 shows scalability of compact scheduling and scatter scheduling with interleaving. For all parallel frameworks, scatter scheduling with interleaving achieves the best scalability. We again inspect memory behavior for scatter+interleave case. In Figure 9, we see that overall memory bandwidth is increased with respect to compact scheduling cases (Figures 7(a) and 7(c)). Furthermore, NUMA memory utilization is more balanced across cores both in terms of bandwidth and data volume transferred.



(a) Memory Bandwidth for Scatter+Interleave Setup



(b) Memory Read/Write Volume for Scatter+Interleave Setup

Figure 9: Memory Behavior of Scatter+Interleave.

6. Combination of All Optimizations

In Section 5, we see that using interleaved memory policy combined with scatter thread scheduling delivers the best performance. In light of this information, we revisit static and dynamic loop scheduling policies. In Figure 10, we show the scalability behavior of different parallel frameworks using static and dynamic loop policies. All experiments use scatter thread policy and interleaved memory policy. We observe that dynamic scheduling improves the performance of the OpenMP version compared to the static scheduling if it is used with memory interleaving. However, changing loop scheduling from static to dynamic does not have any impact on the performance of the Fast-Flow implementation.

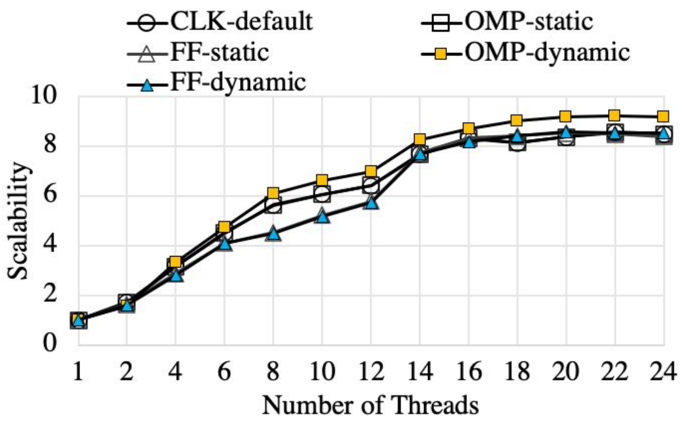


Figure 10: Using dynamic and scatter scheduling together with interleaving policy delivers the best performance for OpenMP and Cilk programs.

Finally, Figure 11 shows how far we were able to improve scalability with respect to initial implementations. We see that combining memory interleaving with scatter thread scheduling and dynamic loop scheduling improved the scalability of OpenMP implementation significantly.

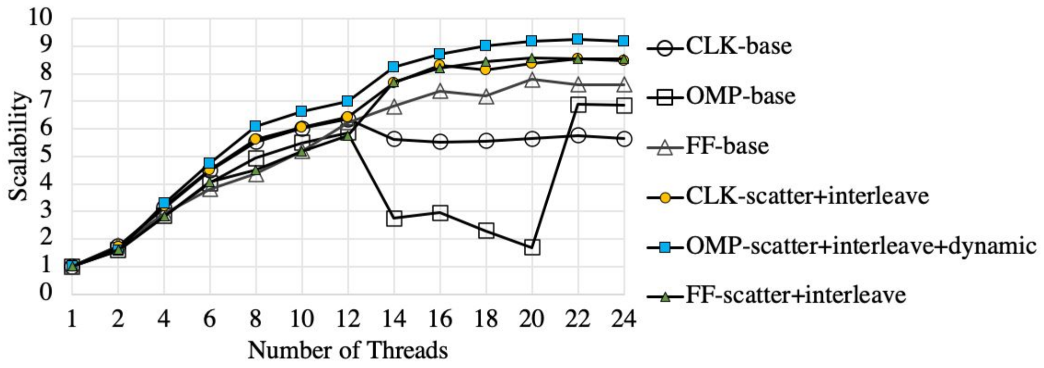
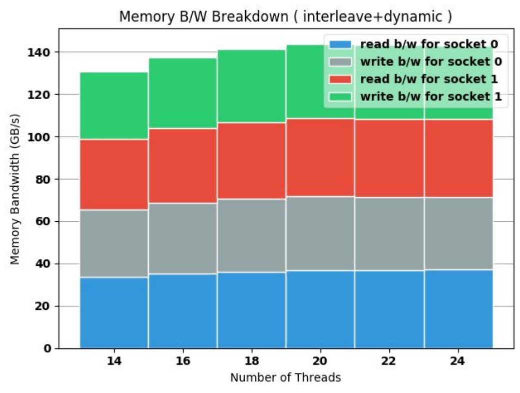
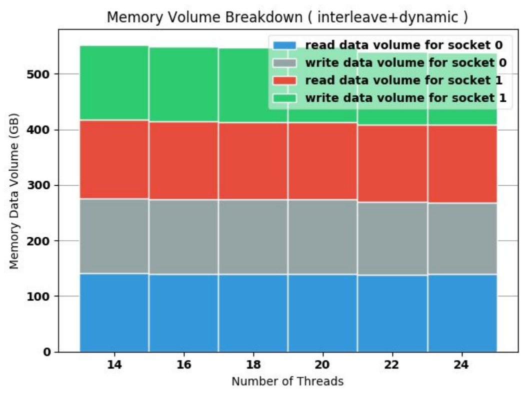


Figure 11: After final settings and optimizations, scalability of all three parallel frameworks outperform the scalability of the base implementations. The OpenMP version has the highest scalability.

According to Figure 12, the dynamic scheduling improves the locality and reduces the miss ratio for OpenMP implementation, which helps the scalability to become higher. The memory bandwidth and the data volume imbalance between the two sockets that we saw in Figures 7(c) and 7(d) decreased significantly with interleaving memory allocation policy.



(a) Memory bandwidth for interleave+dynamic setup



(b) Memory Read/Write Volume for interleave+dynamic setup

Figure 12: Performance metrics for the OpenMp version with interleaved memory policy, scatter thread scheduling and dynamic loops scheduling.

7. Conclusions & Future Work

The scheduling and memory policy are just two factors that can affect the performance in a program with the locality of this particular benchmark. Given more time, it may have been possible to use the LIKWID markers to distinguish the best way to optimize certain functions in the code for better scalability. This paper shows, however, that it is very important to take into account the thread scheduling policy and memory policies to run the code while trying to optimize, and not just focus on how the code itself can be optimized.

As future work, scheduling with different chunk sizes for static and dynamic scheduling in OpenMP can be tested. Similar memory bandwidth analysis can also be applied to FastFlow and Cilk parallel frameworks. However, we expect similar behavior to our OpenMP experiments. Furthermore, other techniques such as vectorization (SIMD) can also be considered for further improving the performance. LIKWID can also be helpful in analyzing the success of compiler's auto-vectorization (e.g. using FLOPS_DP, FLOPS_SP, FLOPS_AVX groups).

For this project, Funda provided the micro-architectural analysis and optimizations on the Skylake machine and with the Cilk implementation, and prepared the setup for testing different runtimes. Amanda worked on the analysis of the initial code and background information in the report, and changes to the OpenMP and Fast Flow implementations for dynamic scheduling testing.

References

[1] Griebler, Dalvan. “Gitlab Repository of NAS Parallel Benchmark Kernels Held by the WUSTL Parallel Computing Technology Group.” https://gitlab.com/wustl-pctg/NPB-CPP/-/tree/master, Jan 2018. Accessed on 2020-03-28.
[2] D. Griebler, J. Loff, G. Mencagli, M. Danelutto and L. G. Fernandes. **Efficient NAS Benchmark Kernels with C++ Parallel Programming**. *In proceedings of the 26th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. Cambridge, United Kingdom, 2018.
[3] Lee, Angelina. Concurrent Programming 101. *WUSTL CSE 539 Lectures.* Slides 8-12 (2020).
[4] Z. Majo and T. R. Gross, "(Mis)understanding the NUMA memory system performance of multithreaded workloads," *2013 IEEE International Symposium on Workload Characterization (IISWC)*, Portland, OR, 2013, pp. 4, 6.
[5] Likwid tool, likwid-5.0.1, https://github.com/RRZE-HPC/likwid
[6] NAS Parallel Benchmarks, https://www.nas.nasa.gov/publications/npb.html