Hybrid work distribution for parallel programs

Anonymous Author(s)

1 Introduction

In modern computing systems, the increasing number of cores in processors emphasizes the need for efficient utilization of resources. To achieve this goal, it is important to distribute the work generated by a parallel algorithm optimally among the cores.

Typically, there are two paradigms to achieve that: static and dynamic. The standard static way, used in OpenMP [1], is just the fork barrier that splits the whole work into the fixed number of presumably "equal" parts, e.g., split into equally-sized ranges for a `parallel_for`. The classic dynamic approaches are: 1) work-stealing task schedulers like in OneTBB [2] or BOLT [7], or 2) work-sharing [6]. As one can guess, the static approach has a very low overhead on the work distribution, but it does not work well in terms on parallel programs for general-purpose tasks, e.g., nested `parallel_for` or `parallel_for` with uneven iterations. When talking about the dynamic approaches, it is known that work-sharing is good for the initial distribution in comparison with work-stealing — a thread wants to steal a task from some thread with a non-empty queue but it doesn't know which are and there are lots of threads with empty queues at that moment (the random doesn't work well), while it produces more overhead after the initial distribution — a thread wants to give a task to a thread with an empty queue but it doesn't know which are (again, the random doesn't work well). Here, we present a task scheduler that unites two dynamic work distribution paradigms at the same time and achieves a low overhead with good task distribution.

2 The Hybrid Approach

In this work, for simplicity, we consider just `parallel_for`. Any fork-join primitive can be implemented with it. The work distribution consists of two phases: the initial range division among threads and the following load balancing, if necessary. To achieve the best performance of the initial distribution we use work-sharing [6] instead of the classic random work-stealing: we explicitly distribute task blocks among other threads by pushing these blocks to per-thread queues. This approach eliminates contention between threads and has reduced latency, particularly for the last tasks. We achieve $O(\log N)$ complexity on the latency by distributing the work in a tree rather than in a linear manner. The threads form a hierarchical tree structure, and the work is distributed along the tree — each thread that receives a range divides it in half and passes these parts to its "children" threads.

After the initialization, we use the standard work-stealing approach for load balancing. For that, the thread receives its range of work, divides it into tasks, and puts them into its queue, so, they are available for other threads.

Here lies the key conflict between work-sharing in the initial distribution and the following work-stealing — a steal can occur earlier than we distribute the tasks via work-sharing. In this case, the original deterministic distribution will be violated leading to unexpected overheads and to worse cache utilization.

The proposed solution (presented [8, 9] on industry conferences, with no proceedings) delays the creation of load-balancing tasks for a certain timespan. The idea is that a thread, after receiving its range of work, does not create balancing tasks immediately but executes part of the given range during this timespan and only then creates these tasks. The value of the timespan is calculated experimentally and depends on the number of threads and a CPU model, but does not depend on an application. We evaluate it once using a script that calls 10^5 times the `parallel_for` with the range of size `num_threads`. For each loop, a starting thread reports the start time using an rdtsc instruction-based timer and then waits for other threads to start executing the body of the loop. Then, we calculate how long the task scheduling took as the difference between the start and the latest time for a thread starting to execute. Based on these results, we choose a timespan as the 99th percentile of the task distribution time. It is a good approximation of the time when the last thread receives its range of work.

Another question is the granularity problem, i.e., how to define the grain size for balancing tasks — we split a task if it is bigger than that size. For that, we use an adaptive strategy — the grain size is chosen based on the number of executed iterations during the initialization timespan. The idea is that the more iterations are executed, the bigger the grain size should be to avoid situations when the task creation and stealing overheads are more significant than the task execution time.

Our algorithm is implemented in C++ over the simple work-stealing thread pool from Eigen library [4].

3 Experiments

For the experiments, we use the machine with ARM architecture and four NUMA nodes with 32 cores each, giving 128 cores in total. Our benchmarks use all available cores. When the setup was fixed, we calculated the timespan once before the executions. We compile with GCC15 and `-O3 -ffast-math` parameters.

We run the same experiments but for the Intel x86 machine. The results are available in Appendix A.

PL'18, January 01–03, 2018, New York, NY, USA
2018.

We compare our approach against OpenMP (LLVM) and OneTBB with multiple different settings: 1) static, dynamic, and guided scheduling strategies for OpenMP, and 2) auto, affinity, and simple partitioners for OneTBB. Our algorithm appears on the figures with `TIMESPAN_GRAINSIZE` label.

We use several benchmarks to evaluate the performance of the proposed approach (by performance here we mean the execution time). In the first benchmark, we perform sparse matrix-vector multiplication (SPMV) on two different square matrices: a balanced matrix with uniformly distributed non-zero elements and an unbalanced triangle matrix with a decreasing number of elements per row. We are interested in this workload to compare the difference in performance on balanced and unbalanced workloads which should have different load distributions. The second benchmark is Scan — a parallel algorithm that calculates the prefix sum of an array of integers [3]. This benchmark is interesting for us since it has multiple ($2 \log N$) `parallel_for` calls, allowing us to observe how the scheduler behaves in the case of multiple calls. Additionally, we explore the performance of our approach on two similar nested parallel algorithms: matrix multiplication and matrix transposition. All matrices in these benchmarks are square ones.

We use Google Benchmark [5] with `UseRealTime` parameter: the resulting time (in microseconds) is calculated as an average over at least 10 tries. Note, that we run small, i.e., fast, benchmarks more times to get a stable result.

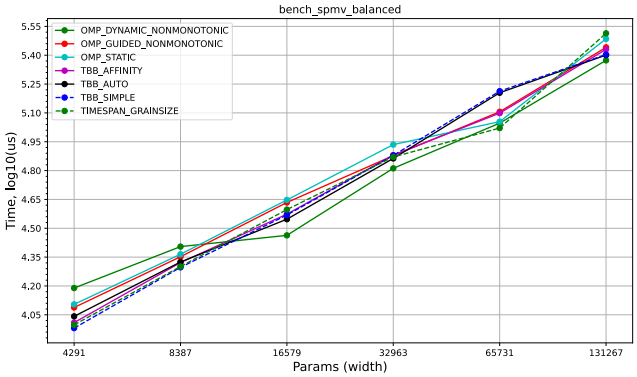

Figure 1. SPMV Balanced, ARM

Results of the SPMV benchmark are presented in Figures 1 and 2. Absolute time is placed on the Y-axis in binary logarithmic scale and the size of the matrix is placed on the X-axis.

As expected, the static OpenMP scheduler has poor performance on the unbalanced matrix. The proposed algorithm, green-dotted, shows the best performance on the unbalanced matrix and is comparable with others on the balanced matrix.

In Figure 3, we can see the results of the Scan benchmark. The Y-axis represents the logarithm of the absolute time, while the X-axis represents the logarithm of the size of the array. Our solution works better than other approaches except for the static OpenMP, but still close to it.

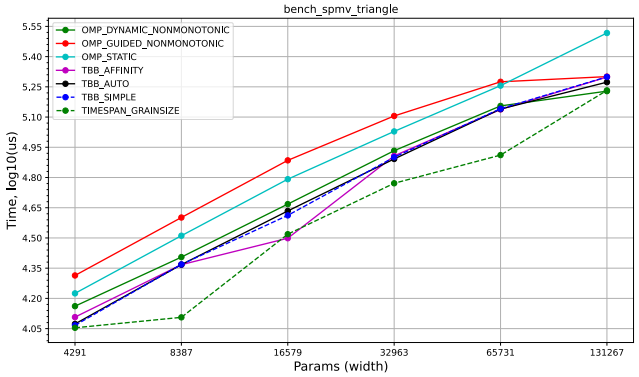

Figure 2. SPMV Triangle, ARM

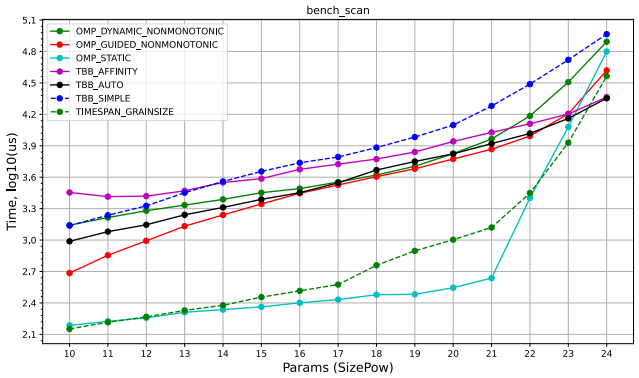

Figure 3. Scan, ARM

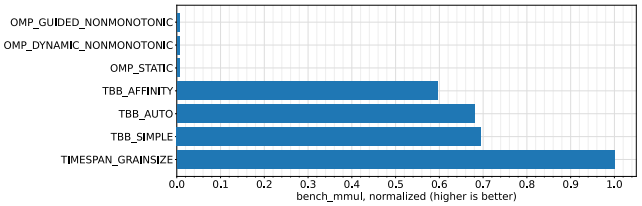

Figure 4. Matrix Multiplication, ARM

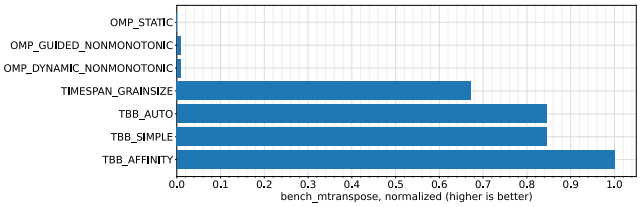

Figure 5. Matrix Transpose, ARM

Finally, we run the algorithms on the nested parallelism benchmarks: Matrix Multiplication (Figure 4) and Matrix Transpose (Figure 5), both with matrices of size $num\_threads \cdot 2^4$. X-axis shows the normalized execution time (higher is better, the best has 1.0), while Y-axis shows the respective algorithm. We see that the proposed algorithm is also as fast as TBB on nested parallelism, in contrast to OpenMP.

4 Conclusion

In this work, we presented a novel algorithm for work distribution that unites the best of two worlds, work-stealing and work-sharing. Our experiments show, that our algorithm achieves either the best or competitive performance.

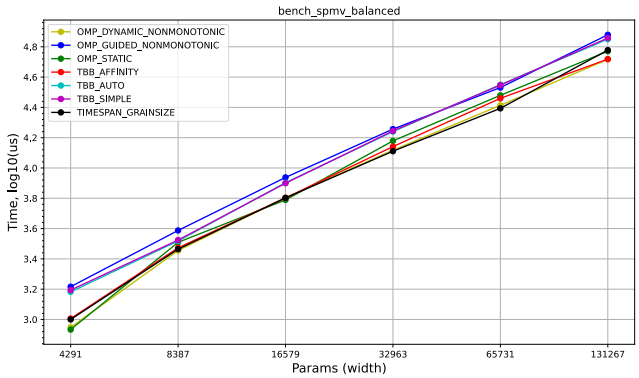

Figure 6. SPMV Balanced, Intel

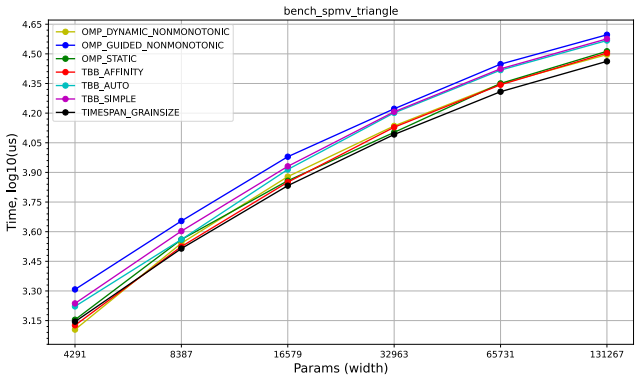

Figure 7. SPMV Triangle, Intel

References

[1] 2021. *OpenMP Application Programming Interface Specification Version 5.2.* https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf

[2] 2022. *oneTBB - oneAPI Specification 1.2-rev-1 documentation.* https://spec.oneapi.io/versions/latest/elements/oneTBB/source/nested-gen-info.html

[3] 2023. *Chapter 39. Parallel Prefix Sum (Scan) with CUDA | NVIDIA Developer.* https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda

[4] 2023. *Eigen C++ Library, Non Blocking Thread Pool.* https://gitlab.com/libeigen/eigen/-/blob/master/Eigen/src/ThreadPool/NonBlockingThreadPool.h

[5] 2023. *Google Benchmark User Guide, Runtime and Reporting Considerations.* https://github.com/google/benchmark/blob/main/docs/user_guide.md#runtime-and-reporting-considerations

[6] Derek L Eager, Edward D Lazowska, and John Zahorjan. 1986. Adaptive load sharing in homogeneous distributed systems. *IEEE transactions on software engineering* 5 (1986), 662–675.

[7] Shintaro Iwasaki, Abdelhalim Amer, Kenjiro Taura, Sangmin Seo, and Pavan Balaji. 2019. BOLT: Optimizing OpenMP Parallel Regions with User-Level Threads. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 29–42. https://doi.org/10.1109/PACT.2019.00011

[8] Anton Malakhov. 2022. Fusing Efficient Parallel For Loops with a Composable Task Scheduler. *Hydra Conference* (2022).

[9] Anton Malakhov and Evgeny Fiksman. 2013. Pushing the limits of work-stealing. *Compiler, Architecture and Tools Conference* (2013).

A Experiments on Intel

We also tested the proposed solution on a machine with two sockets Intel Xeon Gold 6338 (x86) with 24 cores each, giving 48 cores in total. We compile with clang16 and `-O3 -ffast-math` parameters for this platform, all benchmarks use all 48 cores.

We can see that the proposed algorithm is competitive to others on the balanced matrix (marked with a black line in Figure 6) and exhibits the best performance on the triangle matrix (Figure 7) but with less difference than on 128 cores. Additionally, it demonstrates the best results after OpenMP static in almost all parameters of the Scan benchmark (Figure 8) (yellow line).

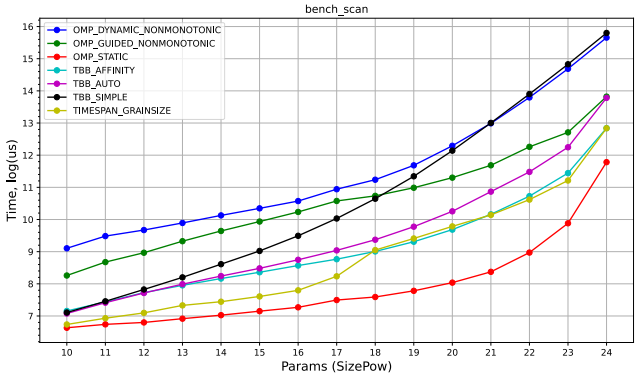

Figure 8. Scan, Intel

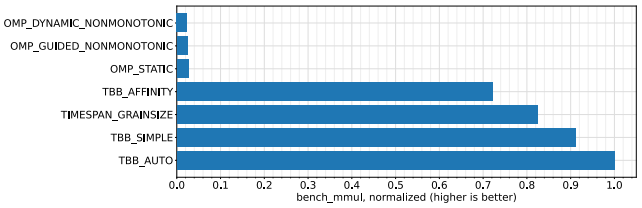

Figure 9. Matrix Multiplication, Intel

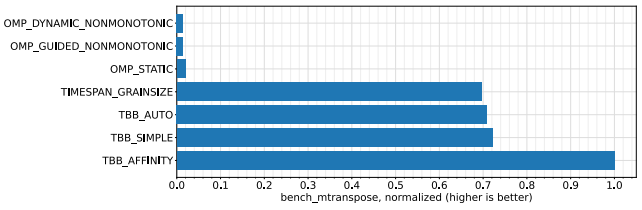

Figure 10. Matrix Transpose, Intel

Similarly, our solution performs at the TBB level while using the nested parallelism in Matrix Multiplication (Figure 9) and Matrix Transpose (Figure 10) benchmarks.