Report on Parallel Execution Performance using OpenMP in C++

Nedal Abu Haltam

May 2025

Overview

This report analyzes the performance results of a parallel computing application, focusing on speedup achieved through multithreading. The `Run.py` script executes a serial and a parallel version of a program with varying thread counts, then computes speedup metrics and visualizes the data through a generated graph that it saves. at the end of this report you can find a graph that was generated when I ran the script

Analysis of C++ Source Code

The provided C++ code is the core computational logic behind the benchmarking tests discussed in this report. It supports both serial and parallel execution modes and is compiled with conditional compilation based on the `_OPENMP` macro. Below is a breakdown of its key components:

Command Line Interface

The program expects command line arguments that include:

- `<Buffer size>` — the size of the vector used for averaging.
- `<Number of threads>` — only in the parallel version.
- Optional flags like `-v` for verbose output and `-DRACE_COND` to induce race conditions for testing.

Execution Flow

The entry point of the program is `main`. Depending on whether OpenMP is enabled (`#ifdef _OPENMP`), it calls either `ParallelExecution` or `SerialExecution`.

SerialExecution

This function creates a vector of doubles initialized with sequential values. It computes the average using a straightforward loop and measures execution time using the C-style `clock()` function.

ParallelExecution

This version utilizes OpenMP for multithreading. It partitions the vector among threads to compute partial sums. A critical section ensures safe accumulation of results to avoid race conditions unless explicitly enabled via `-DRACE_COND`.

Supporting Functions

- `ShiftArgs` — utility to shift and consume command-line arguments.
- `StartClock, EvaluateClock` — measure execution time.
- `GetAverageSerial, GetAverageParallel` — compute averages serially and in parallel.

Observations

- The code is careful to account for non-uniform thread workloads.
- Proper synchronization using `#pragma omp critical` ensures correctness.
- The verbose mode provides internal state details which are useful for debugging and benchmarking.

Handling Non-Uniform Thread Workloads

The C++ code explicitly handles the issue of non-uniform workloads when parallelizing the averaging operation over a vector. This is particularly important in parallel programming, where the total number of elements may not be perfectly divisible by the number of threads.

Thread Partitioning Logic

The code calculates a base chunk size as:

```
int SizeOverCount = vec.size() / ThreadCount;
```

Then, each thread computes its range of indices based on its thread ID:

```
int start = ThreadID * SizeOverCount;
int end = ((vec.size() <= ThreadCount || vec.size() % ThreadCount != 0) &&
           ThreadID == ThreadCount - 1) ? vec.size() : (ThreadID + 1) * SizeOverCount;
```

Explanation

- `start` is the index at which the thread begins processing.
- `end` is the index at which the thread stops. For the last thread, it ensures that it picks up any remaining elements that would otherwise be left out due to integer division.

Why This Matters

This logic guarantees that:

1. All elements are processed exactly once.
2. No thread processes beyond the bounds of the vector.
3. Load imbalance is minimized by assigning extra work to the last thread, if necessary.

Such careful partitioning is crucial for both correctness and performance in parallel programs, especially when data sizes do not evenly divide by thread counts.

Analysis of `Run.py`

The script `Run.py` performs a benchmarking task. It runs a serial executable and then runs a parallel version of the same task with thread counts ranging from 1 to 12. The execution time for each is recorded and used to calculate speedup compared to the serial baseline.

Additionally, the script tests parallel execution with 120 threads and compares the result to the serial execution time, highlighting the impact of excessive threading on performance.

Results Summary

Although speedup was generally observed with increasing thread count, the performance gains were not always linear. Notably, using 120 threads did not yield the best results. Instead, fewer threads sometimes produced better speedups. This is attributed to the overhead associated with hyperthreading and inter-thread communication at higher thread counts.

Conclusion

The experiment underscores the importance of tuning the number of threads in parallel applications. More threads do not always equate to better performance due to communication overhead. For optimal performance, a balance must be struck between parallelism and system limitations.

1 Image Generated from `Run.py` script

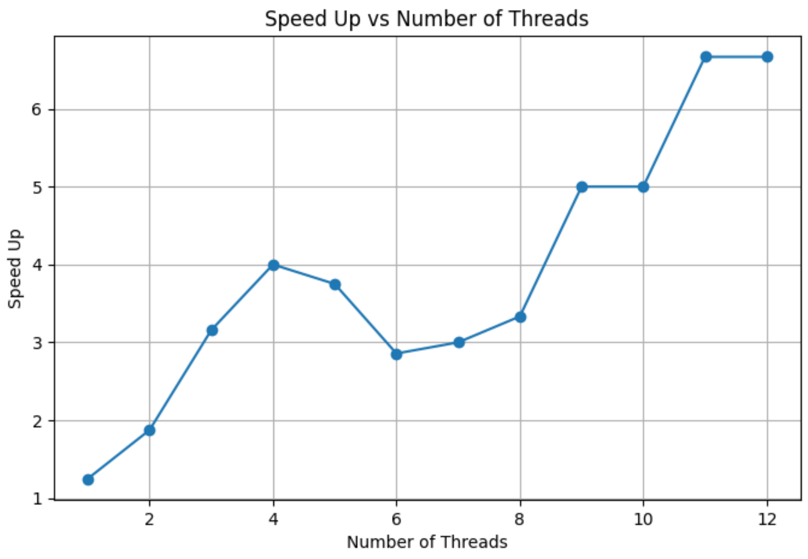

Figure 1: Speed Up vs Number of Threads