Sum Performance Study
Assignment #1, CSC 746, Fall 2024

Daniil Matveev* SFSU
E. Wes Bethel† SFSU

ABSTRACT

This report compares the performances of three summation algorithms in a computational and memory perspective. The tests were executed on a virtual machine using an Intel Xeon 8462Y+ processor, measuring metrics such as MFLOP/s, memory bandwidth usage, and memory latency. From the results, direct summation yielded the highest number of MFLOP/s due to a minimal amount of memory latency, whereas the indirect one underperformed quite significantly because of the poor spatial locality and increased memory latency. It turned out that this access pattern of memory has a massive impact on algorithm performance, since the indirect is 1622 times slower than the direct method.

1 INTRODUCTION

In this experiment, we studied the influence of the algorithms' memory access patterns on the performance of the implementation. The algorithms we studied are three iterative ways to compute the sum of the first N non-negative integers: $\sum_{i=0}^{N-1} i$. Although essentially all three algorithms do the same job they significantly differ in the ways they use the memory.

The first one computes the sum without the usage of external memory. In contrast, the second and the third ones use external arrays to store integers and access them with and without compliance with the memory locality principle. We measured the runtime for each algorithm on a series of problem sizes and computed three metrics - MFLOP/s, % of memory bandwidth utilized, and average memory latency.

Program using external memory designed to benefit from spatial locality is still ≈ 7.3 worse in megaflops than the program without external memory. Furthermore, the program with external memory and low spatial locality has even worse performance in megaflops, memory bandwidth and memory latency are ≈ 222.2 worse compared to the program with high spatial locality.

2 IMPLEMENTATION

In order to measure the influence of memory accesses on the performance of the algorithm we considered three different algorithms computing $\sum_{i=1}^{N-1} i = \frac{n(n-1)}{2}$. In the following subsections, we give pseudocode and implementation details of these three algorithms.

2.1 Direct Sum

For the first algorithm, we wanted to test the performance without any access to RAM/Cache. So our insensitive was to write the algorithm which with maximal optimization will store all variables in CPU registers.

In Listing 1 we give the C++-style pseudocode for this algorithm. As you can notice the algorithm uses only two variables and doesn't use any additional memory. The maximal N we considered is 2^{28}, so for the loop index we used int variable. In total, the algorithm has N addition operations with int64 variable answer.

```
// adding integers without external memory
int64 answer = 0;
for (int i = 0; i < N; i++)
  answer += i;
```

Listing 1: Summing integers using only two variables.

2.2 Vector Sum

For the second algorithm, we wanted to test the same summing algorithm, but with reading elements, we will add from the memory. Additionally, we wanted the algorithm to has high spatial locality, so the integers $0, 1 \dots N$ will be stored consequently.

In Listing 2 we give the C++-style pseudocode for this algorithm. It consists of two stages firstly we initialize the array with integers $0, 1 \dots N$ and then add them in a single for loop which is timed. Similarly to the previous algorithm, we can use the int32 variable for the loop index since maximal problem size N is 2^{28}. In total, we have N memory access operations and N addition operations with int64 variable answer.

Let's consider a small example of this algorithm for $N = 3$.

```
// initializing A with 0, 1, ..., N - 1
vector<uint64> A(N);
for (int i = 0; i < N; i++)
  A[i] = i;

// summing integers from vector
// the following code is being timed
int64 answer = 0;
for (int i = 0; i < N; i++)
  answer += A[i];
```

Listing 2: Summing integers consecutively stored in an array with.

2.3 Indirect Sum

The final algorithm counts sum $\sum_{i=1}^{N} i$, with storing integers in the array, but without spatial locality and therefore with higher memory latency. To achieve this we needed to ensure that the next integers we will add in the for loop have a low probability of being in cache after the previous loop iteration. Additionally, we wanted it to still compute the correct answer while iterating over all numbers only once.

To achieve this goal we designed an algorithm that create a permutation of integers $0, 1, \dots, N-1$ with a single cycle. The pseudocode is given in the Listing 3. Although we are not giving the formal proof of correctness the idea is that in each loop iteration, we merge two disjoint cycles so after $N-1$ iterations we will have $N-(N-1)=1$ cycle in the permutation. Additionally, we note that the actual implementation was tested on 1) creating a single cycle 2) calculating the

*email:dmatveev@sfsu.edu
†email:ewbethel@sfsu.edu

correct sum result. In total, we have N memory access operations and $N-1$ int64 additions for the timed section of the code.

```
// initializing A with cycle-permutation of 0, 1,
    ..., N - 1
vector<uint64> A(N);
for (int i = 0; i < N; i++)
  A[i] = i;
for (int pivot = 0; pivot < N - 1; pivot++)
  swap(A[pivot], A[randint(pivot + 1, N - 1));

// iterating over permutation and summing integers
// the following code is being timed
int64 answer = 0;
for (int i = A[0]; i != 0; i = A[i])
  answer += i;
```

Listing 3: Firstly we create cycle-permutation of integers $0 \dots N-1$. Then we iterate over the permutation and compute the sum.

3 Evaluation

In order to test the hypotheses we touched on in the previous sections for each of the designed algorithms we designed and performed the evaluation benchmark on a real machine. To estimate the effects that external memory accesses on the expected number of arithmetic operations we measured MFLOP/s metric. To test the effects of the spatial locality principle that helps us to rate different algorithms with external memory we measured memory latency and the percentage of maximal memory bandwidth the program utilized.

3.1 Computational platform and Software Environment

We were running our test on a virtual machine hosted by SFSU cloud compute cluster. The machine had

- Intel(R) Xeon(R) Platinum 8462Y+ CPU with
  - x86_64 architecture
  - 2.8 GHz clock rate
  - 8 Cores
  - 640 KiB of L1 cache and 16/480 MiB of L2/L3 cache
- Ubuntu 22.04 OS
- 128 GB of RAM with 38 GB/S limit on memory bandwidth (the information about maximal memory bandwidth was provided by the hosts)

All codes were written in C++ with C++11 standard and compiled with gcc 11.4 compiler with -O3 optimization flag.

3.2 Methodology

For each of the three ways to count the sum, we measured the elapsed time on each problem size N from the exponential range $2^{23}, 2^{24}, \dots, 2^{28}$. Since considered problem sizes double $N \in (2^i)_{23}^{28}$ we firstly checked that we indeed get the runtime exponential growth as you can see in the figure 1.

Then using a runtime and the analysis of the algorithm we derived three metrics - megaflops, % of memory bandwidth utilized, and memory latency.

3.3 MFLOP/s

The first metric we derived is megaflops = $\frac{\text{number of operations}/10^6}{\text{time}}$. As we derived in the section 2 direct, vector, and indirect programs do N, N, and $N-1$ int64 additions for the problem size N. The computed MFLOP/s values are presented in the figure 2. In our experiment, the average MFLOP/s of vector method decreased by 7.3 times compared to the direct method. The usage of the indirect method further decreases the megaflops (and increases runtime) by 222.2 times. Additionally, as it can be seen much better in the figure 2 compared to the figure 1 the MFLOP/s monotonously decreases

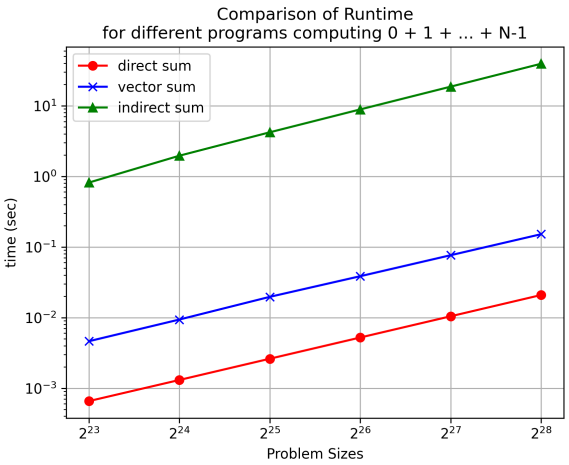

Figure 1: Runtimes of the summing algorithms tested on exponentially increasing problem size and plotted in the log scale. As you can see all three algorithms look very linear in the log scale, and the runtime indeed approximately doubles when problem size doubles.

for the indirect summation method. Since this cost of arithmetic operations doesn't change this means that the slowdown comes from the increase of memory latency (/decrease of memory bandwidth), which we will verify in the following sections.

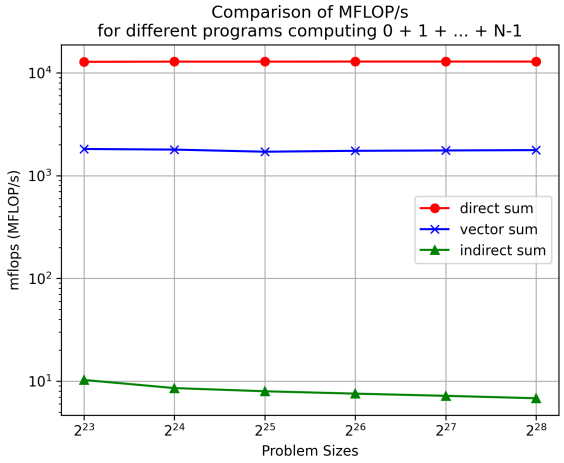

Figure 2: Megaflops of tested programs presented in the log scale. While the direct and vector methods is closer to a constant, the indirect method performance monotonously decreases with problem size growth.

3.4 % of memory bandwidth utilized

The next metric we derived was the percentage of theoretical peak bandwidth utilized = $\frac{\text{bytes accessed}/\text{time}}{\text{capacity}}$. The maximal memory capacity on our machine was 38GB as reported by the system administrator. The direct sum method did not use external memory, while both vector and indirect methods computed N memory accesses to int64 variables. Therefore each of vector and indirect methods accessed $8N$ bytes. As was predicted in the previous section on the figure 3 you see that memory bandwidth monotonously decreases over time for the indirect method.

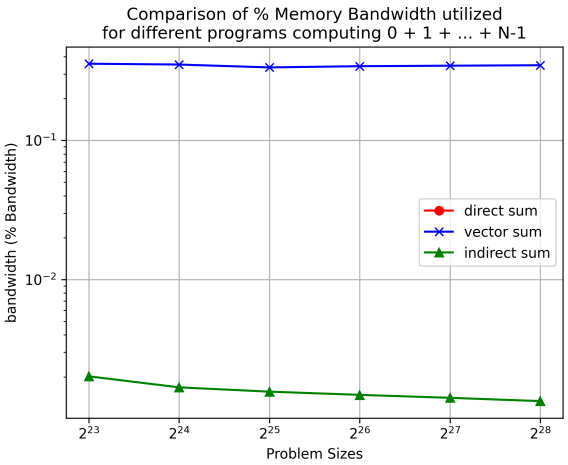

Figure 3: Memory bandwidth utilized in percentage the log scale. Since the direct method doesn't use memory it's results are outside of the chart. Memory bandwidth % of indirect summation method monotonously decreases with time.

3.5 Memory latency

The final metric we derived was memory latency $= \frac{\text{time}}{\text{\#of memory acceses}}$. As we predicted in section 2, the memory latency of the indirect method increased with time, what means that the cache miss rate increases with the growth of N. Interestingly the latency of vector memory seems to be convex (with one global minimum), which can probably be explained by the cache properties.

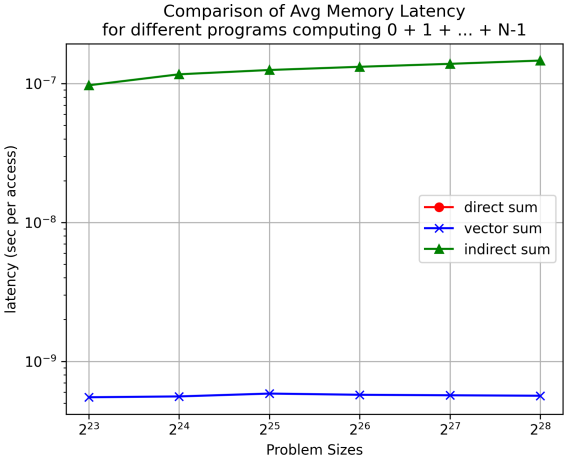

Figure 4: Memory latency in the log scale. Since the direct method doesn't use memory its results are outside of the chart. Memory bandwidth % of the indirect summation method monotonously increases with time.

3.6 Findings and Discussion

The maximal MFLOP/s we were able to realize was ≈ 12865 and it was achieved on the direct method of sum computation. This is the expected behavior since this method doesn't suffer from memory latency. However according to Intel reports the original processor on our machine should have 2150.4 maximal GFLOP/s, what is ≈ 40 times faster than our result considering that our virtual machine had 4 times fewer cores than the real intel 8462Y processor. Unfortunately due to the limited access to the virtual machine, we did not figure out what might cause this problem.

The maximal memory bandwidth of $\approx 35.59\%$ was realized by the vector summation method and the lowest of $\approx 0.13\%$ by the indirect method. This is explainable by the fact that the next element we access next in the loop has a low probability of being already in the cache since it has some random position compared to the previous ones. In contrast, the direct method will maximally utilize the spatial locality principle since integers are accessed in the natural order. The same reasoning explains why the minimal memory latency of $5.51 \cdot 10^{-10} \frac{seconds}{operation}$ was achieved by the vector method and the maximal of $1.17 \cdot 10^{-7}$ by the indirect method.

We consider the 1622 times difference between the method without additional memory and the method with additional memory with bad spatial locality the most important result of this report, since it shows the how largely different might be algorithms doing the same job but with different memory access behaviors.