TMA4280 computing parallel sums of vectors.

Jon Christian Halvorsen, Anette Fossum Morken,
Monica Kappeslåen Plassen

February 20, 2015

In this project we have studied the vector $v \in \mathbb{R}^n$ where the elements are defined as

$$v(i) = \frac{1}{i^2}, \quad i = 1, 2, ..., n \tag{1}$$

and the sum S_n of all the vector elements

$$S_n = \sum_{i=1}^{n} v(i). \tag{2}$$

First, a single processor C program (sum.c) generating v, computing S_n in double precision and computing and printing the difference $S - S_n$ for different values of n was written. Next, the program was changed to utilize shared memorize paralellization through OpenMP (sum_ omp.c).

Then, a program computing the sum using P processors and a distributed memory model, as described in [1], was made (sum_ mpi.c). In this program we used the convenient MPI calls MPI_ Scatter and MPI_ Reduce. The scatter call were chosen instead of the send/receive calls because this operation was well suited for a one-to-all operation. The scatter call were also used instead of the distribute call since the separate threads don't need all the data (saves memory). The reduce call was used instead of MPI_ Gather since all we wanted was the sum, and the the gather was not needed. Necessary MPI calls were as usual the MPI_ Init, MPI_ Comm_ size, MPI_ Comm_ rank and MPI_ Finalize. Error values of the sum are presented in table 1. We assume the process to be load balanced since we let MPI do the scattering and the processes are constant in time (summing equal amount of numbers). Obviously the rank 0 processor gets a little more to do before and after the parallel period, but in the actual summing it should be load balanced.

Lastly a program using both MPI and OpenMP were tested (sum_ omp_ mpi.c) and confirmed working. The difference in the sum was exactly the same as in table 1 if we ran the serial code, the MPI code with $P = 2$ or $P = 8$ threads. Since the MPI code scatters the data over more threads we don't have control

Table 1: Error values for different values of n using the MPI program with two cores.

n	$S - s_n$
8	1.175120e-01
16	6.058753e-02
32	3.076680e-02
64	1.550357e-02
128	7.782062e-03
256	3.898631e-03
512	1.951219e-03
1024	9.760858e-04
2048	4.881621e-04
4096	2.441108e-04
8192	1.220629e-04
16384	6.103329e-05

over the order which the elements are summed and is thus not guaranteed an equal result. In this case it worked out well.

A floating point operation is addition, subtraction, multiplication and addition of two floating point numbers. We define addition and subtraction as flop type a, multiplication as type b and division as type c. In our programs we defined the i as an integer and the one in the numerator of (1) as a floating point. Therefore, the only floating point operation per element in the vector v is the division, and in total n floating point operations (of type c) are needed to generate v. If we had defined i as a float, we would also have one type b operation per element, and the total number of operations would be $n \cdot b + n \cdot c$. To compute the sum S_n in (2) after v is computed, $n - 1$ additions has to be made, so the total number of operations is $a \cdot (n - 1)$.

We consider the memory requirement for our single and multi-processor programs. In the single processor program, we need to store the n elements of the vector v, in other words n floating points, plus all variables of different data types that exist in the program. We define m to be the memory required to store one element in v and m_{var} the total memory required to store all other variables in the program, for example the analytical sum S, which is stored as a float, and the integer n. Let f_s be the total memory requirement for the single-processor program and we see that

$$f_s(n) = m_{var} + m \cdot n. \tag{3}$$

Now we consider the same for the multi-processor program utilizing MPI, and look at the memory requirement for each of the P individual processors. All of them will have to store a copy of the global variables and their own local variables. In addition, processor 0 will have to store the entire vector v, and

all processors, including processor 0, will get a part of the vector of size $\frac{n}{P}$ that they store individually. Let m_{glob} and m_{loc} be the memory requirement for the global and local variables respectively, and define m as before. Define f_M^j to be the total memory requirement for processor j.

$$f_M^j(n,P) = \begin{cases} m_{glob} + m_{loc} + m \cdot (n + \frac{n}{P}) & j = 0 \\ m_{glob} + m_{loc} + m \cdot \frac{n}{P} & j = 1, ..., P \end{cases} \tag{4}$$

We notice that when n becomes very large, the number of local and global variables become very small compared to the number of elements in the vector, so the memory requirement for the variables become negligible compared to the memory requirement to store v. Then the parts including n in (3) and (4) can be compared and we observe that processor 0 has to store more than the single processor, while the other ones have to store less.

For this problem, we do not think parallel programming is very attractive, especially since we used a quite limited vector size n. The single processor program managed the computations and storage fast and without issues for these sizes.

References

[1] *TMA4280: Introduction to supercomputing, Problem set 4*, Einar M. Rønquist.