TMA4280 - Superdatamaskiner
Exercise 4
Eirik Mildestveit Hammerstad

I just want to say sorry if anything is not up to level of your expectation. I've caught the flu and 2 hours before delivery I managed to rm * -rf on the wrong directory, so sorry if some of the code/report is not up to standards.

1)
The serial version can be found in serial.c. It outputs the following:

S - S(16)	1.5843465334449871	Error: 0.0605875334032393
S - S(32)	1.6141672628279242	Error: 0.0307668040203022
S - S(64)	1.6294305014088870	Error: 0.0155035654393394
S - S(128)	1.6371520049544612	Error: 0.0077820618937652
S - S(256)	1.6410354363086801	Error: 0.0038986305395463
S - S(512)	1.6429828479550965	Error: 0.0019512188931299
S - S(1024)	1.6439579810301654	Error: 0.0009760858180610
S - S(2048)	1.6444459047881168	Error: 0.0004881620601096
S - S(4096)	1.6446899560231332	Error: 0.0002441108250932
S - S(8192)	1.6448120039860050	Error: 0.0001220628622214
S - S(16384)	1.6448730335545856	Error: 0.0000610332936408

The serial code is quite straight forward and naive: For n = 2^k, k = 4,..,16 i do the following: I create a vector where all elements are 1^-2, and sum the elements in the vector. The vector_t struct can be found in common.h.

2)
The parallel version with shared memory can be found in parallel-shared.c. It outputs the following:

S - S(16)	1.5843465334449871	Error: 0.0605875334032393
S - S(32)	1.6141672628279242	Error: 0.0307668040203022
S - S(64)	1.6294305014088870	Error: 0.0155035654393394
S - S(128)	1.6371520049544612	Error: 0.0077820618937652
S - S(256)	1.6410354363086803	Error: 0.0038986305395461
S - S(512)	1.6429828479550954	Error: 0.0019512188931310
S - S(1024)	1.6439579810301632	Error: 0.0009760858180632
S - S(2048)	1.6444459047881148	Error: 0.0004881620601116
S - S(4096)	1.6446899560231238	Error: 0.0002441108251026
S - S(8192)	1.6448120039860075	Error: 0.0001220628622189
S - S(16384)	1.6448730335545798	Error: 0.0000610332936466

The parallel version with shared memory works much like the serial, with the exception that the summation is done in parallel (with a reduction on the sum variable).

3)
The parallel version with distributed memory can be found in parallel-distributed.c. It outputs the following:

S - S(16)	1.5843465334449871	Error: 0.0605875334032393
S - S(32)	1.6141672628279242	Error: 0.0307668040203022
S - S(64)	1.6294305014088870	Error: 0.0155035654393394
S - S(128)	1.6371520049544612	Error: 0.0077820618937652
S - S(256)	1.6410354363086801	Error: 0.0038986305395463
S - S(512)	1.6429828479550965	Error: 0.0019512188931299
S - S(1024)	1.6439579810301654	Error: 0.0009760858180610
S - S(2048)	1.6444459047881168	Error: 0.0004881620601096
S - S(4096)	1.6446899560231332	Error: 0.0002441108250932
S - S(8192)	1.6448120039860050	Error: 0.0001220628622214
S - S(16384)	1.6448730335545856	Error: 0.0000610332936408

The parallel version with distributed memory had to be rewritten after an accidental removal of some files. The first processor creates a vector and splits it up to the other processors, which sum the elements and return the answer, which is added together on the first processor.

4)
Necessary/convenient MPI calls:
MPI_Init
MPI_Comm_size
MPI_Comm_rank
MPI_Get_processor_name
MPI_Send
MPI_Recv
MPI_Finalize

5)
If we compare the results from the different programs, they are identical for n=4, but for n=16, then parallel yield a better result. I believe this has to do with the representation of floating point numbers. Once we start losing precision when adding new elements, the sum gets influenced, something which happens at a stronger rate in the sequential version.

6)
In the sequential version, we generate and hold everything in one shared memory, while the parallel distributed version creates everything in a single process, and copies parts of the vector to other processes, which would use more ram.

7)
I assume that it would create two floating point operations per item in vector_t->data, one to sum them, and one for the minus, so 4n where n is elements.
Only one processor creates the vector, which is a significant part of the exercise, so it is not loadbalanced.

8)
No, I believe we would get better results with a dynamic programming implementation.