Peter Gruber
3rd Homework
HPSC 2004

Exercise 3.4:
Implement the summation of vectors each of N doubles. Use four vectors and four processes and suppose that each process knows initially only a single vector. Use MPI point-to-point communication to implement your version of the vector summation. Test your program for small and large vectors. Comment results! Compare the performance of your implementation with that of MPI_ALLREDUCE. Explain any differences!

Basic concept of the implemented algorithm
The basic idea behind this algorithm is that the processes involved send their local values (that is their local vector) to the neighbouring process in a ring structure. In fact, process number i would send it's local data to process number $(i+1) \bmod m$, where m denotes the number of processes involved (m will be 4 in our case). Processes are numbered (in consistence with MPI semantics) from 0 to $m-1$.
When a process receives a vector from a neighbouring process, it updates it's value by adding the vector this process was assigned initially to the vector just received. As a first step, each process sends it's initially known vector to the neighbouring process. Then, after m iterations, each process should finally receive a vector representing the sum of all m vectors that were initially assigned to the different processes.
In particular, process i starts by sending its initially known vector v_i to process $(i+1) \bmod m$. We will denote the vector that is passed on from one process to another by v, so after the first step we have $v = v_i$. Process number $(i+1) \bmod 4$ adds it's own local vector $v_{(i+1) \bmod 4}$ to the vector v just received. Thus, v represents know the sum of these two vectors, that is $v = v_i + v_{(i+1) \bmod 4}$. After m iterations, process number $i + m \bmod m = i$ receives the vector $v = \sum_{k=0}^{m-1} v_{i+k \bmod m} = \sum_{k=0}^{m-1} v_k$. So we have proved that this algorithm is indeed an implementation of the vector summation.
Before starting the iteration process, we have to initialize the local vector for each process. For this example, we will use pseudo-random numbers, using the formulas $v[0] = i$ and

$$v[n] \;=\; (i \cdot v[n-1] + 7) \bmod N^2 \quad \forall 1 \leq n \leq N-1 \;,$$

where N denotes the size of the vector and i the number of the process. This method won't creta particularly good pseudo-random numbers, however, this is not really necessary for our purpose.
The implementation of the algorithm may look like this (the number of participating processes is supposed to be 4):

```
int N          // size of vector
int[N] v, myv  // local and communicated vector
int i          // number of process

// Initialize local vector myv
myv[0] = i;
for (n = 1; n <= N-1; n++)
  {myv[n] = (i * v[n-1] + 7) mod (N*N);}

// Start algorithm by sending local vector to neighbouring process
Send vector myv to process number (i+1) mod 4;

// Start iteration of updating and passing on vector v
for (k=1; k <= 3; k++) {
  Get vector v from process number (i-1) mod 4;
  v = v + myv;
  Send vector v to process number (i+1) mod 4;
}

// vector v now contains sum of all local vectors for every process
```

For different sizes N of the vectors used, the following computational times were measured for this example:

N	10	100	1000	10000
time	0.0097	0.0103	0.0153	0.0633

From this data we may conclude that the time required for communication between processes is an important factor for small vectors. For larger vectors, communication time becomes more and more neglible compared to the computation time needed to update the vectors in each iteration.

Replacing the algorithm based on point-to-point communication simply by an MPI_ALLREDUCE call, the following values for computation time were measured:

N	10	100	1000	10000
time	0.0003	0.0008	0.0026	0.0226

A possible explanation for the difference in computation time might be that MPI_ALLREDUCE implements optimizations possible for such function calls that yield a significant speed up.

The source code of the program used is cited below:

```
#include <stdio.h>
#include <math.h>
#include "mpi.h"


   int main(int argc, char **argv){

      int        rank, size, n, k, N, right, left;
      MPI_Status status;
      double     atime1, etime1, atime2, etime2;

      MPI_Init(&argc, &argv);

      MPI_Comm_rank( MPI_COMM_WORLD, &rank );
      MPI_Comm_size( MPI_COMM_WORLD, &size );

      N = 10000;
      double      v1[N], v2[N], myv[N];

      myv[0]    = rank;
      v1[0]     = rank;
      for (k = 1; k < N; k++){
         myv[k] = (rank * (int) v1[k-1] + 7)%(N*N);
         v1[k]  = myv[k];
      }

      if (rank == 0)        left = size - 1;
      else                  left = rank - 1;
      if (rank == size-1)   right = 0;
      else                  right = rank + 1;

    /************************* my own method ************************/
      atime1 = MPI_Wtime();
      for (k = 0; k < 3; k++){
         MPI_Send(v1, N, MPI_DOUBLE, right , 0, MPI_COMM_WORLD);
         MPI_Recv(v2, N, MPI_DOUBLE, left, 0, MPI_COMM_WORLD, &status);
         for (n = 0; n < N; n++)
            v1[n] = v2[n] + myv[n];
      }
      etime1 = MPI_Wtime();
    /****************************************************************/

    /************************ MPI_ALLREDUCE ************************/
      atime2 = MPI_Wtime();
      MPI_Allreduce (myv, v2, N, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
      etime2 = MPI_Wtime();
```

```
   /*************************************************************/

   if (rank == 0){
      printf("Time needed using my own method: %lf\n",etime1-atime1);
      printf("Time needed using MPI_ALLREDUCE: %lf\n",etime2-atime2);
   }
   MPI_Finalize( );
   return 0;
}
```