IMPLEMENTATION & DESIGN CHOICES

1) max-serial.c

Very straightforward, all the integers are read in sequence, and the maximum element is found in a single loop and written to an output file.

Input file name is taken as an argument to the program, at argv[1]. The output file created by this program is max-serial-out.txt and includes the result, number of integers read and the runtime.

2) max-mpi-pp1v.c

Again like the serial version, the input path is passed to the program as argument, at argv[1]. The output created by this program is named max-mpi-pp1v-out.txt and it includes the local maxima in the processors, the overall maximum, number of integers read and the runtime. I think it is quite unlikely for this project, but if we ever have our number of processors greater than number of integers to read in total, then the program uses just one processor to compute the maximum. Otherwise, if np is greater than number of integers to read, which I think is mostly the case, the master dispatches parts of the array (if the array size is size, and number of processors is np - the size that is given to each processor is size/np) to the slaves. Then when the slaves compute the maximum integer in the part of the array that was given to them, the master collects these local values and find the maximum among them.

3) max-mpi-pp2v.c

Pretty much the same as pp1v but this time master does not do any computation instead Allreaduce function is used collecting and reducing all the local maxima from the slaves. The output file name is max-mpi-pp2v-out.txt and it includes the local maxima, overall maximum and the runtime.

4) matmult-serial.c

The input files are arguments to the program, argv[1] is the first path and argv[2] is the second path. This serial program computes the resulting matrix in three nested loops. Since both matrices are known to be square matrices all the loops run dimension times. The outer loop runs dimension times and used to iterate the rows, the one inside that also runs dimension times and goes through the columns and the inner loop gets the i-th element (where i in [1..dimension]) from both the row and the column. The products found in the innermost loop are summed and the sum is put in the result matrix in row with the current value of the iterator of the outer loop and in column with the current value of the second outer loop. The output file is named matmult-serial-out.txt and it includes the dimension, the resulting matrix each row printed in a single line and each column printed in separate lines and the runtime.

5) matmult-mpi-1d.c

The input files are arguments to the program, argv[1] is the first path and argv[2] is the second path. First of all, you'll probably remember me, I sent you an e-mail about the implementation of this program saying that I don't use the master processor to do any computation, so one of the important things is that if the matrix dimension is N, then the number of processors should satisfy the following statement where np is the number of processors:

N % sqrt(np - 1) == 0

So for example, if we're to multiply two matrices of dimension 6, the expected number of processors should divide 36, so possible numbers of processors are [1,4,9,36], however my implementation of the program does not include the root processor so when running the program please run it with an additional processor and in this case possible numbers of processors with the redundant root processor would be [2,5,10,37]

The rows and columns are distributed equally, all processors have (N^2)/(np-1) combinations of row and column pairs. The processors are distributed a row vector first and then given as many column vectors as possible, if they don't have enough number of row and column pair combinations, then other row vectors are passed. So for example, if ran the program with two matrices of dimension 6 and 9 processors (or 10 with a lazy master), first, the number of combinations of row and column pair that should be reached for one processor is (6^2)/(9) = 4. So each processor will have 4 row&column pairs, in other words each processor will compute 4 dot products for the resulting matrix. In this case, the first processor will first receive the first row vector, and then the first four column vectors. Then the second processor will receive the first vector and start receiving the column vectors where the previous one left off, so the second processor will receive the fifth and the sixth column vectors, and because the column vectors are finished it will check if there are any more row vectors to receive or not and it will get the second row vector and start receiving the column vectors from the start, getting the first and the second column vectors and having 4 pairs in total.

So in this case

Processor 1	will get	row [0]	column [0,1,2,3]
Processor 2	will get	row [0,1]	column [4,5,0,1]
Processor 3	will get	row [1,]	column [2,3,4,5]
Processor 4	will get	row [2]	column [0,1,2,3]
Processor 5	will get	row [2,3]	column [4,5,0,1]
Processor 6	will get	row [3]	column [2,3,4,5]
Processor 7	will get	row [4]	column [0,1,2,3]
Processor 8	will get	row [4,5]	column [4,5,0,1]
Processor 9	will get	row [5]	column [2,3,4,5]

One improvement to this distribution algorithm could be to send blocks of rows and columns so that the number of rows and the number of columns that are received by a processor are equal. And that would prevent redundancy such as Processor 2 having the data for row[1] and columns[4,5] but not taking part in their computation. In my implementation a processor received either (N^2)/(p) or ((N^2)/(p))+1 vectors. However, a by equally distributing the vectors communication size of only (N^2)/(p) vectors for all the processors could be achieved. An example for 6x6 matrices with 9 processors would be:

Processor 1	will get	row [0,1]	column [0,1]
Processor 2	will get	row [0,1]	column [2,3]
Processor 3	will get	row [0,1]	column [4,5]
Processor 4	will get	row [2,3]	column [0,1]
Processor 5	will get	row [2,3]	column [2,3]
Processor 6	will get	row [2,3]	column [4,5]
Processor 7	will get	row [4,5]	column [0,1]
Processor 8	will get	row [4,5]	column [2,3]
Processor 9	will get	row [4.5]	column [4,5]

And this is possible since N^2 is divisible by p and p is assured to be a perfect square in which case sqrt(p) is an integer and divides N, so we could divide both matrices into p blocks and have 2*sqrt(p) blocks and have possible combinations of p blocks, and distribute each combination to one processor as done in the most recent example.

So, once the computation done in the slave processors, the master receives the results and puts them in another 2d array and once everything is complete it writes the resulting matrix to the output file named "matmult-mpi-1d-out.txt" which also includes the matrix dimension and the runtime.

--- Note for parts 1, 2 and 3. I defined the integer array size that'll hold the read integers to be 10,000. I just didn't want to do additional work to count how many lines there are and dynamically create the array, I hope you don't cut off points I feel too tired and overwhelmed to do it but know that I can.

PERFORMANCE

A) For input size 100, serial's runtime is 0.0002ms on the average, and for input size 10,000 it is 0.004ms on the average. I think the reason why it wasn't 0.02ms as should be expected for a program that runs on O(n) with 100 times the bigger input size is because I/O overhead. Anyway, with the same input file with 10,000 integers max-mpi-pp1v

performed	0.0037ms~	with	-np 2,
performed	0,008ms~	with	-np 5,
performed	0.01ms~	with	-np 10,
performed	0.02ms~	with	-np 50.

First of all I think the reason why parallel always performed worse after I went past the number of physical processors has a couple of reasons. One is the overhead of initialization of MPI system and finalization, the second is communication cost between the processing elements the existence of which can be clearly seen by the deteroriating performance with the increasing number of processors that add the to communication cost. And the communication cost is not just related to the size of the data that is transferred over the program course but number of message sent and received affects it quite a lot since with the changing number of processors the size of data transfer does not change.

max-mpi-pp2v also performed quite similarly,

performed	0.0037ms~	with	-np 2,
performed	0.007ms~	with	-np 5,
performed	0.009ms~	with	-np 10,
performed	0.02ms~	with	-np 50.

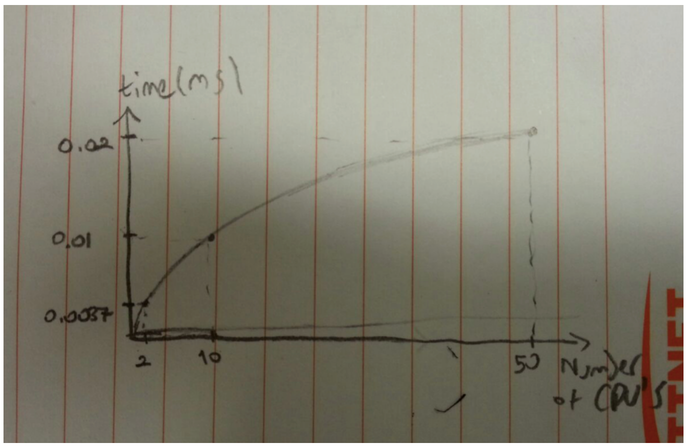
The same graph could also apply to pp2v too and I don't think Allreduce adds increases the communication performance but it is only handy that we could tell what function it should apply to reduce the input so when it is boiled down it is just the ease of use I suppose.

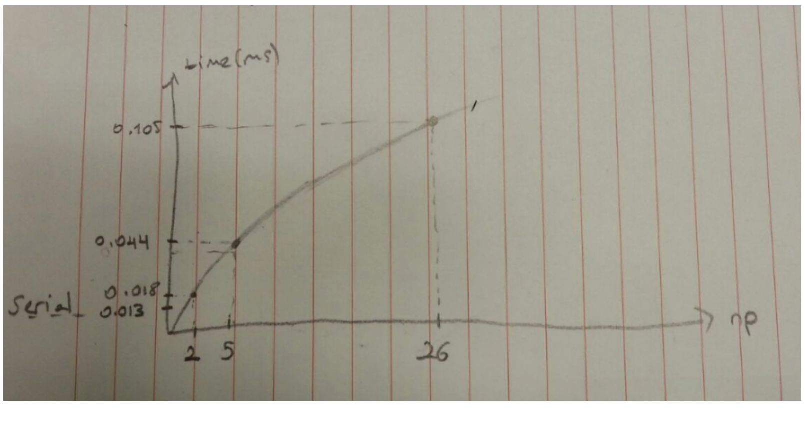
For input size 100, my results remained in the same proportion to the results with input size 10,000.

In brief words, when going past the physical number of processing elements, the communication overhead becomes significant and causes the parallel program to perform worse than the serial program, however it performs better than the serial program with 2 processors in parallel as I have seen on my machine.

B) Serial program ran on an average of 0.013ms~ for 100x100 matrices, whereas the parallel program ran on average durations 0.018ms~, 0.044ms~ and 0.105ms~ with number of processors 2, 5 and 26 (1,4,25) respectively. The fact that the number of processors should be a perfect square and its square root must divide matrix dimension and errors I receive when wanted to run a program with a large number of processors, (100 processors gave me an error) limited my experimentations. However, I could say the same principals that applied to the parallel program in part A applies to part B as well and the with the increasing number of processors, although the communicated data size gets smaller the number of messages sent and received become a bottleneck.

GRAPHS (1st for part a, 2nd for part b, sorry for the professional amateurism with the graphs)


time(ms)
Number of CPU's


time(ms)
Serial
np