CS434: Introduction to Parallel and Distributed Computing

Ndze'dzenyuy, Lemfon K.

01st April 2021

Lab 3

1 Providing a high level overview of the approach used.

To write this program we use p + 1 number of processes, where one process serves as a master process, and the remaining p serve as servant processes that receive relevant chunks of the matrix to be multiplied from the master process, multiply them, and then send back the result to the main process. The main process times the entire time taken to compute the result using the parallel approach. In the main process too, we execute the multiplication sequentially, time the sequential computation, and then compare the results.

For each matrix dimension, there are constraints to the number of processes that can be used. This constraint specified that the number of servant processes must be such that its square must be a factor of the dimension of the matrix. Let us imagine for example, that we want to work with matrices of dimension 6. To determine the valid number of servant processes, we begin by stating all the factors of 6. Which in this case are $\{1, 2, 3, 6\}$. We then proceed to square these numbers, obtaining $\{1, 4, 9, 36\}$ as the set from which we can choose a valid number of processes. Because in this implementation we use the master process only for coordination and not for any computation, the arguments that actual number of processes that one will have to use when running this process will be $\{2, 5, 10, 37\}$. However, this program does not allow a user to use a single servant process, as this is clearly a sequential approach. Although the program will not prevent a user from using a number of servant processes that is the square of the dimension, one must realise that such an approach will most likely not be the best.

When the program starts, we determine if the user has entered a valid set of processes, and exit the code in the event that they have not. If the number of processes entered was valid, we move into the master process in which we generate A and B using a predefined function. Using the rank of each servant process, we determine the appropriate rows and columns of A and B respectively that the process will need to carry out its computations and extract them using predefined functions. We then convert these into vectors to obtain a contiguous memory location and send to each process.

When each process receives its rows and columns, it converts them into matrices and then carries out the multiplication using the serial function for multiplying two matrices of compatible rows and columns. After the multiplication is completed, the resulting matrix is converted to a vector for reasons explained in the paragraph above and sent back to the master process. Aside sending the results of computation to the master process, for each process we send a derived MPI data type (A Struct to be specific) that holds information about the position in the final result matrix at which we must begin the insertion of the results obtained from that process. The master process then receives the results from each servant process, alongside the index at which it should begin the insertion. After which the results are inserted into the final result matrix appropriately.

Finally, the master process runs the multiplication in a sequential manner and measures the time that was used.

The following information is printed out for the user at several stages of the program's life:

- The matrix A
- The matrix B
- The results as obtained from each process
- The final result matrix
- The amount of time for the parallel computation
- The amount of time for the serial computation

2 Presenting the functions used and the pseudo-codes

2.1 void determine_valid_procs(int n)

This function helps us to determine the set of valid processor numbers that could be used for a given matrix dimension.

```
1 ALGORITHM: determine_valid_procs (int n)
2 for i ← 2 to n do
3 |  If i is a factor of n, print (i ∗ i) + 1
4 end
```

2.2 int ** create_matrix(int dimension)

This function creates a matrix initialized with random numbers and returns it. The matrix is created using dynamic allocation of memory.

```
1 ALGORITHM: create_matrix(int dimension)
2 Create a matrix through dynamic allocation of memory
3 for each matrix position do
4 |  generate a random number between 1 and 100 and save it
5 end
6 Return the matrix.
```

2.3 int ** create_matrix_value(int dimension, int value)

This function creates a matrix initialized with specific numbers and returns it. The matrix is created using dynamic allocation of memory.

```
1 ALGORITHM: create_matrix_value(int dimension, int value)
2 Create a matrix through dynamic allocation of memory
3 for each matrix position do
4 |  store the value that was passed as an argument.
5 end
6 Return the matrix.
```

2.4 void print_matrix(int **mat, int dim, char name)

This function allows us to print a square matrix, alongside its name.

```
1 ALGORITHM: print_matrix(int ∗ ∗ mat, int dim, char name)
2 Print out the matrix name
3 for each matrix position do
4 |  Print out the value
5 end
```

2.5 void print_nonsquare(int **mat, int rows, int cols)

This function allows us to print a non-square matrix.

```
1 ALGORITHM: print_nosquare(int ∗ ∗ mat, int rows, int cols)
2 for each matrix position do
3 |  Print out the value
4 end
```

2.6 int **allocarray(int rows, int cols)

This function allows us to dynamically allocate memory to be used later. The allocated space could be a square matrix or not.

```
1 ALGORITHM: allocarray(int rows, int cols)
2 Dynamically allocate the space, but put nothing in it.
```

2.7 int **get_division_cols(int **mat, int dimension, int start, int end)

This function allows us to extract a given number of columns from a matrix. The start and end variables represent the indices at which we must start and end the extraction and are used inclusively

```
1 ALGORITHM: get_division_cols(int **mat, int dimension, int start, int end)
2 temp = create a dynamic array that has the required number of columns and rows
3 for each matrix position of mat that has to be copied do
4 | Copy it into temp
5 end
6 Return temp
```

2.8 int **get_division_rows(int **mat, int dimension, int start, int end)

This function allows us to extract a given number of rowss from a matrix. The start and end variables represent the indices at which we must start and end the extraction and are used inclusively

```
1 ALGORITHM: get_division_rows(int **mat, int dimension, int start, int end)
2 temp = create a dynamic array that has the required number of columns and rows
3 for each matrix position of mat that has to be copied do
4 | Copy it into temp
5 end
6 Return temp
```

2.9 int **allocvector(int len)

This function allows us to dynamically allocate memory to be used later. The allocated space is used to represent a vector.

```
1 ALGORITHM: allocarray(int rows, int cols)
2 Dynamically allocate the space, but put nothing in it.
```

2.10 void printvector(int *vec, int len)

This function allows us to print a vector.

```
1 ALGORITHM: printvector(int *vec, int len)
2 for each vector position do
3 | Print out the value
4 end
```

2.11 int *matrix_linearize(int **mat, int rows, int cols)

This function allows us to take a matrix, and store it into a vector

```
1 ALGORITHM: matrix_linearize(int **mat, int rows, int cols)
2 temp = dynamically create a vector for each matrix position do
3 | Store the value in temp
4 end
5 return temp
```

2.12 int **vec_to_array(int *vec, int len, int rows, int cols)

This function allows us to take a vector, and return the matrix the vector represents. To do this, we integer-divide the index of the vector by the number of columns to know the row in which the element

at the index should be placed, and we calculate the modulus to determine the column.

```
1  ALGORITHM: vec_to_array(int *vec, int len, int rows, int cols)
2  temp = dynamically create an array of rows rows and cols columns
3  for i = 1 ← len do
4  |   r ←i / cols
5  |   c ←i % cols
6  |   temp[r][c] ← vec[i]
7  end
8  return temp
```

2.13 int **multiply_seq(int **a, int **b, int a_rows, int a_cols, int b_rows, int b_cols)

This function allows us to multiply two matrices. The constraint that the columns of the first matrix must equal the rows of the second matrix is expected to be verified.

```
1  ALGORITHM: multiply_seq(int **a, int **b, int a_rows, int a_cols, b_rows, int b_cols)
2  temp = dynamically create an matrix of a_rows rows and b_cols columns
3  for i = 0 ← a_rows do
4  |   for j = 0 ← b_cols do
5  |   |   temp[i][j] ←0
6  |   |   for k = 0 ← a_cols do
7  |   |   |   temp[i][j] ←temp[i][j] + (a[i][k] * b[k][i])
8  |   |   end
9  |   end
10 end
11 return temp
```

2.14 Discussing Results

The table below presents the amounts of time taken in seconds to multiply random matrices generated by our code.The results are presented for situations in which 4, 9, 16, 36, 64 and 144 worker processes are used. Although the use of 4 processes when $n = 4$ is not optimal, it is used here for the purpose of demonstration.

n	Sequential	Worker processes and time taken					
		4	9	16	36	64	144
2	0.000004	0.000575	N/A	N/A	N/A	N/A	N/A
4	0.000003	0.000139	N/A	N/A	N/A	N/A	N/A
6	0.000004	0.004124	0.014832	N/A	N/A	N/A	N/A
12	0.000017	0.000831	0.031026	0.036360	0.073429	N/A	N/A
24	0.000126	0.000826	0.024172	0.032883	0.074059	0.084698	1.298576

As expected, the results demonstrate that as the dimension increases, the amount of time used to multiply the matrices sequentially increases. However, a quick look at the performance in the parallel situation indicates that the sequential algorithm generally performs better. Another thing to remark is that as the number of processes increases, it is not a guarantee that the computation will become more efficient. These observations suggest two things:

- It may be the case that for small matrices, it is most efficient to multiply them in a sequential manner, as the introduction of a parallel approach that sends and receives messages comes along with overhead incurred during message sending and receiving.
- It is also the case that increasing the number of processes does not always guarantee better performance. For example, when $n = 24$, and we increased the number of worker processes from 64 to 144, we are sending approximately 240 more messages.

Therefore, computer scientists must realize that although parallel methods come with the advantage of being able to utilize more computing power, it must be understood that the context must be taken into consideration and an investigation carried out to ensure that efficiency is not rather lost in the pursuit of parallelisation.

Although this program has restrictions on the number of processes, one possible way in which it could be improved is to use padding the matrices with zeros to make the code work when the number of processes entered is less than the dimension, but not in conformity with the constraint as stated in the first paragraph of this document. A second improvement could be made by combining the rows and columns sent to each process into a custom MPI Datatype and then effecting only a single call to MPI_Send. This for example, will reduce the number of sends by approximately 33.33% for all computations.