MONASH University

Lab Session Week 7

Student ID	Student Name	Student Email
32844700	Teh Jia Xuan	Jteh0015@student.monash.edu

Task 1 – Code with comments and the required functionality
OpenMPI

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <stdbool.h>
#include <string.h>
#include <mpi.h>

//function prototype
bool isprime(int number);
int compare(const void *a, const void *b);

int main (int argc, char *argv[])
{
    //function declaration and initialisation
    int *arr = NULL;
    struct timespec start, end, startComp, endComp;
    double time_taken;
    int n, rank, size, localcounter = 0;
    int totalCount = 0;
    int counter = 0;
    int *counts, *displacement;
    int *localarr = NULL;

    //Initialise the MPI environment and processes
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); //get current rank
    MPI_Comm_size(MPI_COMM_WORLD, &size); //get all the processes num
    //only root process
    if (rank == 0){
        //ask for user input for n
        printf("Enter an integer larger than 10000000: ");
        fflush(stdout);
        scanf("%d", &n);
        printf("Compute:\n");

        //counts for gathering all the count of local prime result in an array
        counts = (int *)malloc(size * sizeof(int));

        //displacement for gathering the displacement of the each result array from each processes
        displacement = (int *)malloc(size * sizeof(int));

    }
    else{
        counts = NULL;
        displacement = NULL;
    }

    //broadcast n value to other processes
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

    //localarr to store local prime number
    localarr = (int *)malloc(n * sizeof(int));
```

```c
// Get current clock time to time the computational time
clock_gettime(CLOCK_MONOTONIC, &start);

//using round robin work distribution
//use mod to distribute task -> more evenly distributed
for (int i = 2; i < n; i++){
    if (i % size == rank){
        if (isprime(i)){
            //store in each local array of processes
            localarr[localcounter] = i;
            localcounter++;
        }
    }
}

//wait for other processes to done before proceeding
MPI_Barrier(MPI_COMM_WORLD);

//Gather all the local count to an array eg {23,43,15,4}
MPI_Gather(&localcounter, 1,MPI_INT, counts, 1,MPI_INT, 0, MPI_COMM_WORLD);

if (rank == 0){
    displacement[0] = 0;
    totalCount = counts[0];
    for (int i = 1 ;i < size; i++){
        //add all the counts from each processes to total count
        totalCount += counts[i];
        //calculate the displacement of each processes
        //using the counts to know each array have how many values
        //and then set the displacement
        //for the program to know where to start
        displacement[i] = displacement[i - 1] + counts[i - 1];
    }
}
if (rank == 0){
    arr = (int *)malloc(totalCount * sizeof(int));
}
//gather all local array into a single array using counts and displacement
MPI_Gatherv(localarr, localcounter, MPI_INT, arr, counts, displacement, MPI_INT, 0, MPI_COMM_WORLD);
//sort the prime array using compare function and qsort
qsort(arr, totalCount, sizeof(int), compare);

//computational time end
clock_gettime(CLOCK_MONOTONIC, &end);
if (rank == 0){
    //print all the output
    for(int i = 0; i < totalCount; i++){
        printf("%d ", arr[i]);
    }
}
    //get the time
    if(rank == 0){
    time_taken = (end.tv_sec - start.tv_sec) * 1e9;
    time_taken = (time_taken + (end.tv_nsec - start.tv_nsec)) * 1e-9;
    printf("Overall time: %f sec \n", time_taken);
    }
    MPI_Finalize();
    return 0;
}


// Comparison function for qsort
int compare(const void *a, const void *b) {
    //if it is positive number means a is larger
    //if is negative number means b is larger
    //deferencing to access the integer value
    return (*(int *)a - *(int *)b);
}
```

```
bool isprime(int number){

    if (number <= 1){
        return false;
    }

    if (number == 2 || number == 3){
        return true;
    }

    if (number % 2 == 0){
        return false;
    }

    if (number % 3 == 0){
        return false;
    }

    //loop until sqrt of n as p*q = n so loop until one of them is sufficient
    for (int i = 5; i <= sqrt(number); i++ ){
        if (number % i == 0){
            return false;
        }
    }
    return true;
}
```

POSIX

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <stdbool.h>
#include <string.h>
#include <pthread.h>
#define THREADS 8

bool isprime(int number);

//define global variable
size_t num_elem = 10000000;
int *arr;
int counter = 0;
pthread_mutex_t mutex;

//call prime number function
void *calprime (void *arg){

    //indicate start and end
    int start = *((int*)arg);
    int end = *((int*)arg + 1);

    //looping through the numbers to check prime
    for (int i = start; i < end ; i++){
        if (isprime(i)){

            //if is prime then lock the threads so that one thread can update each time
            //to prevent race condition
            pthread_mutex_lock(&mutex);
            arr[counter] = i;
            counter++;
            pthread_mutex_unlock(&mutex);
            //unlock after done updating
        }
    }
}

int main ()
{
    //declaration and initialisation
    pthread_mutex_init(&mutex, NULL);
    struct timespec start, end, startComp, endComp;
    double time_taken;
    pthread_t threads[THREADS];
    int thread_args[THREADS][2];
    pthread_mutex_t mutex;
    arr = (int *)malloc(num_elem * sizeof(int));

    //get step for each threads
    int step = 10000000 / THREADS;

    // Get current clock time.
    printf("Compute:\n");
```

```c
    clock_gettime(CLOCK_MONOTONIC, &start);

    for(int i = 0; i < THREADS; i++){
        //set each thread for its start and end in a 2d array
        thread_args[i][0] = i * step;
        if (i == THREADS - 1){
            thread_args[i][1] = 10000000;
        }
        else{
            thread_args[i][1] = (i + 1) * step;
        }
        //create the threads using i
        pthread_create(&threads[i], NULL, calprime, &thread_args[i]);
    }

    //join the threads together
    for (int i = 0; i < THREADS; i++){
        pthread_join(threads[i], NULL);
    }

    pthread_mutex_destroy(&mutex);
    clock_gettime(CLOCK_MONOTONIC, &end);
    //print out the updated array-> all the prime number
    for(int i = 0; i < counter; i++){
        printf("%d ",arr[i]);
    }
    //get the clock time and get the time that the application run
    time_taken = (end.tv_sec - start.tv_sec) * 1e9;
    time_taken = (time_taken + (end.tv_nsec - start.tv_nsec)) * 1e-9;
    printf("Overall time: %f sec \n", time_taken);

}

//determine prime number
bool isprime(int number){

    if (number <= 1){
        return false;
    }

    if (number == 2){
        return true;
    }

    if (number % 2 == 0){
        return false;
    }

    //loop until sqrt of n as p*q = n so loop until one of them is sufficient
    for (int i = 3; i <= sqrt(number); i++ ){
        if (number % i == 0){
            return false;
        }
    }
    return true;

}
```

MONASH University

Task 1 Q&A – Observations, results and explanation

I have 8 cores available in my machine

Base speed:	3.20 GHz
Sockets:	1
Cores:	8
Logical processors:	16
Virtualization:	Enabled
L1 cache:	512 KB
L2 cache:	4.0 MB
L3 cache:	16.0 MB

Number of cores = 8

Serial code

```
Overall time: 5.135101 sec
```

The serial code used 5.1351 second to run

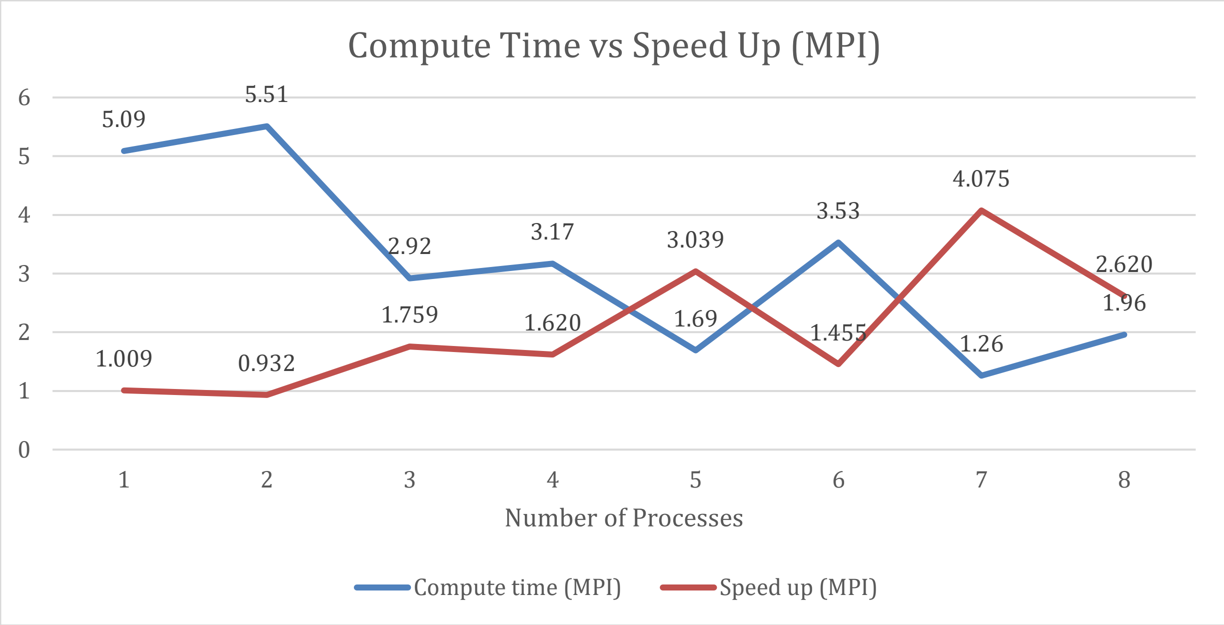
a) MPI Process overall time taken and speed up for each number of processes

The table below presents the total time taken to execute the program using up to 8 processes with the corresponding speed up. The input size is N = 10000000.

Parallel code (MPI)

Number of processes	Overall time	Speed Up
1	Overall time: 5.090669 sec	1.0089
2	Overall time: 5.509445 sec	0.9320
3	Overall time: 2.924571 sec	1.7586
4	Overall time: 3.168973 sec	1.6199
5	Overall time: 1.692404 sec	3.0385
6	Overall time: 3.531077 sec	1.4547
7	Overall time: 1.258780 sec	4.0755
8	Overall time: 1.964361 sec	2.6199

The parallel code used 1.964361 second to run
The speed up is 5.1351s /1.964361s = 2.6199
Theoretical speed up is 8 times



Higher of MPI processes always yield larger speed-ups?

According to the table and plot above the number of MPI processes doesn't always yield larger speed up. By creating 7 processors it will have the fastest computational time with 4.0755 times of speed up. Creating 6 processors the program will run slower than 3, 4, 5. All number of processor slower than the theorical speed up this may due to communication overhead and memory access.

Why speed up may not be same as theoretical speed ups

1) Communication overhead, increasing the number of processes can reduce execution time due to communication overhead. Due to the number of processes increases the communication overhead between processes can also increase. This doesn't exist in the theoretical models thus the actual speed up is slower

2) Amdahl's law, the theoretical speed up assumes that the entire program can be parallelized however the speed up is limited by the portion of the code that cannot be parallelized. So no matter how many processes we created it eventually will hit the limit of speed up.

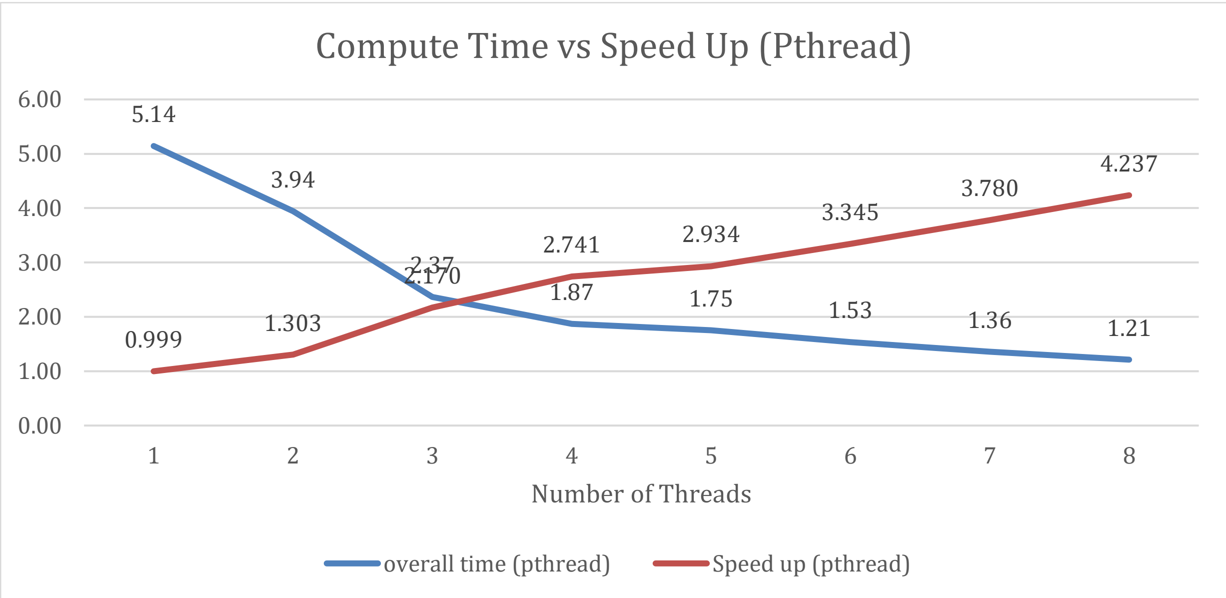
3) Memory bandwidth and cache constraint, when more processes running in parallel it consumes more memory bandwidth or CPU caches thus it results in reduces of performance.

b) POSIX thread overall time and speed up calculations

The table below presents the total time taken to execute the program using up to 8 threads with the corresponding speed up. The input size is N = 10000000.

Parallel code (pthread)

Number of Threads	Overall time	Speed Up
1	Overall time: 5.142748 sec	0.998513
2	Overall time: 3.941305 sec	1.302894
3	Overall time: 2.366826 sec	2.169615
4	Overall time: 1.873313 sec	2.741187
5	Overall time: 1.750478 sec	2.933542
6	Overall time: 1.534935 sec	3.345484
7	Overall time: 1.358583 sec	3.779748
8	Overall time: 1.211979 sec	4.236955

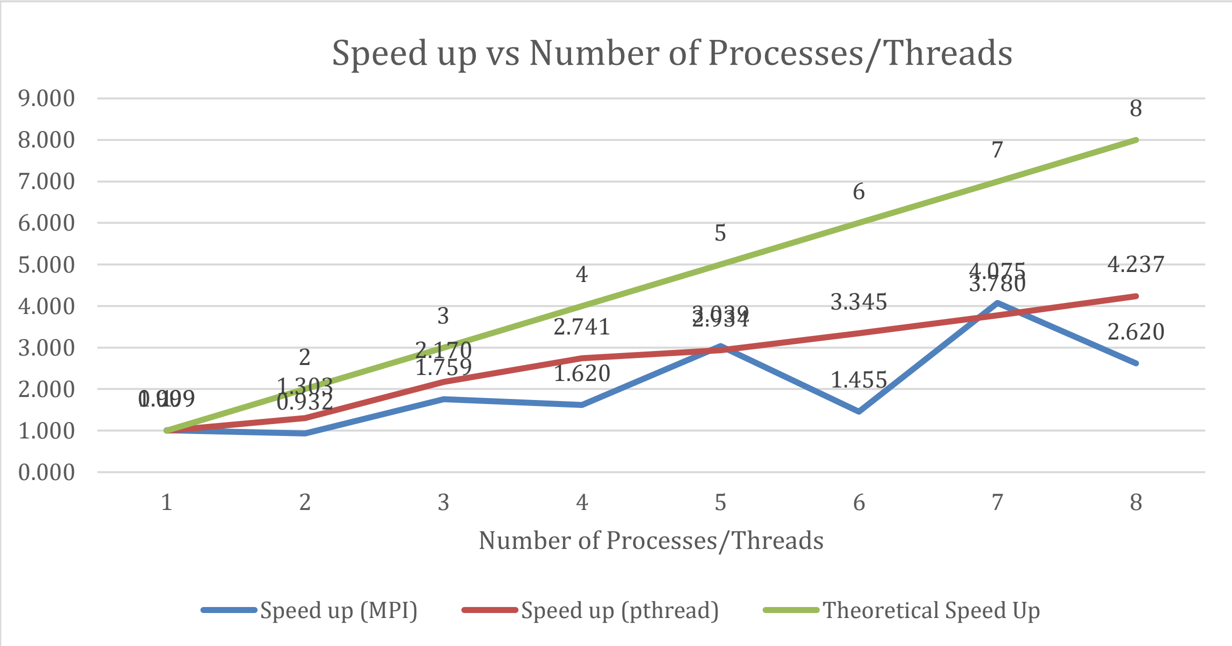


Why actual speed up is slower than theoretical speed up

1) Synchronization overhead, in a multi thread programs they are sharing resources. Thus, it will wait for one thread done with updating or accessing the resources before it accesses. Which reduce the speed up as they need to wait when accessing resources to prevent race condition

2) Thread creation and scheduling, creating threads introduce overhead like spending time to initialise threads, allocating resources and scheduling to prioritise which threads etc. It can introduce overhead and slow down the performance which the theoretical speed up does not cover.

POSIX vs MPI speed up



The chart presents a comparison of speedup between Pthreads, MPI, and the theoretical speedup. Pthreads have a greater speedup than MPI, with the maximum speedup for Pthreads reaching 4.237, while MPI achieves a peak speedup of 3.780. The speedup for Pthreads increases linearly as the number of processors increases from 1 to 8. In contrast, MPI shows that for some cases, using more processors results in slower performance compared to using fewer processors. This shows that same number of processes and threads doesn’t produce the same speed up.

c) Why MPI is slower than POSIX thread implementation

1) Communication overhead, in MPI it runs on separate nodes so it requires message passing to communicate. This introduces overhead for data transmission time. While in pthreads it using shared memory environments so it allows thread to directly access shared data without need to message passing. So communication between threads is faster because they can access the shared memory directly.

2) Data transfer cost, for MPI when one process needs to communicate with other processes they need to send through message passing. When data is sent the process needs to packed, sent and unpacked when transmitted between processes. It slows the performance especially with a large data. Whereas pthread using shared memory so they don’t need to pack and unpack.

Task 2 – CAAS codes and results
Codes OpenMPI

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <stdbool.h>
#include <string.h>
#include <mpi.h>

//function prototype
bool isprime(int number);
int compare(const void *a, const void *b);

int main (int argc, char *argv[])
{
    //function declaration and initialisation
    int *arr = NULL;
    struct timespec start, end, startComp, endComp;
    double time_taken;
    int n = 10000000, rank, size, localcounter = 0;
    int totalCount = 0;
    int counter = 0;
    int *counts, *displacement;
    int *localarr = NULL;

    //Initialise the MPI environment and processes
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); //get current rank
    MPI_Comm_size(MPI_COMM_WORLD, &size); //get all the processes num
```

```c
    //only root process
    if (rank == 0){
        printf("Compute:\n");

        //counts for gathering all the count of local prime result in an array
        counts = (int *)malloc(size * sizeof(int));

        //displacement for gathering the displacement of the each result array from each processes
        displacement = (int *)malloc(size * sizeof(int));

    }
    else{
        counts = NULL;
        displacement = NULL;
    }

    //localarr to store local prime number
    localarr = (int *)malloc(n * sizeof(int));

    // Get current clock time to time the computational time
    clock_gettime(CLOCK_MONOTONIC, &start);

    //using round robin work distribution
    //use mod to distribute task -> more evenly distributed
for (int i = 2; i < n; i++){
    if (i % size == rank){
        if (isprime(i)){
            //store in each local array of processes
            localarr[localcounter] = i;
            localcounter++;
        }
    }
}

//wait for other processes to done before proceeding
MPI_Barrier(MPI_COMM_WORLD);

//Gather all the local count to an array eg {23,43,15,4}
MPI_Gather(&localcounter, 1,MPI_INT, counts, 1,MPI_INT, 0, MPI_COMM_WORLD);

if (rank == 0){
    displacement[0] = 0;
    totalCount = counts[0];
    for (int i = 1 ;i < size; i++){
        //add all the counts from each processes to total count
        totalCount += counts[i];
        //calculate the displacement of each processes
        //using the counts to know each array have how many values
        //and then set the displacement
        //for the program to know where to start
```

```c
            displacement[i] = displacement[i - 1] + counts[i - 1];
        }
    }
    if (rank == 0){
        arr = (int *)malloc(totalCount * sizeof(int));
    }
    //gather all local array into a single array using counts and displacement
    MPI_Gatherv(localarr, localcounter, MPI_INT, arr, counts, displacement, MPI_INT, 0, MPI_COMM_WORLD);
    //sort the prime array using compare function and qsort
    qsort(arr, totalCount, sizeof(int), compare);

    //computational time end
    clock_gettime(CLOCK_MONOTONIC, &end);
    if (rank == 0){
        //print all the output
        for(int i = 0; i < totalCount; i++){
            printf("%d ", arr[i]);
        }
    }
```

```c
    //get the time
    if(rank == 0){
    time_taken = (end.tv_sec - start.tv_sec) * 1e9;
    time_taken = (time_taken + (end.tv_nsec - start.tv_nsec)) * 1e-9;
    printf("Overall time: %f sec \n", time_taken);
    }
    MPI_Finalize();
    return 0;
}

// Comparison function for qsort
int compare(const void *a, const void *b) {
    //if it is positive number means a is larger
    //if is negative number means b is larger
    //deferencing to access the integer value
    return (*(int *)a - *(int *)b);
}
```

```c
bool isprime(int number){

    if (number <= 1){
        return false;
    }

    if (number == 2 || number == 3){
        return true;
    }

    if (number % 2 == 0){
        return false;
    }

    if (number % 3 == 0){
        return false;
    }

    //loop until sqrt of n as p*q = n so loop until one of them is sufficient
    for (int i = 5; i <= sqrt(number); i++ ){
        if (number % i == 0){
            return false;
        }
    }
    return true;
}
```

Serial code (Batch number 23925)

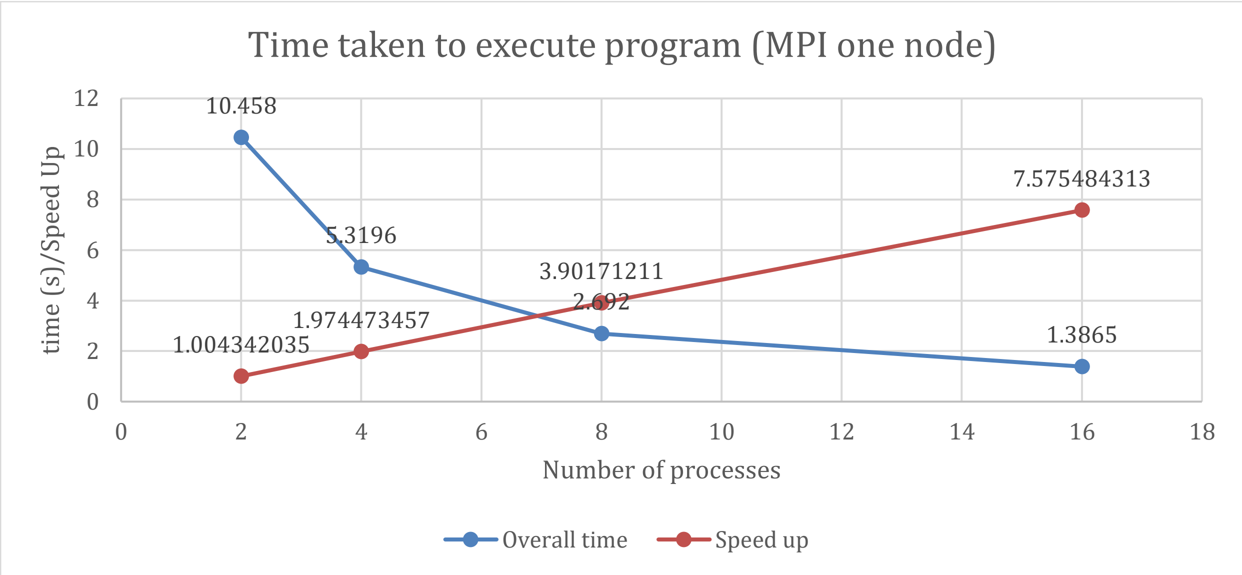
```
Overall time: 10.503409 sec
```

MPI with one compute node

MPI with one compute node is utlised, n = 10000000

Number of processes	Batch code	Overall time
2	23771	Overall time: 10.458045 sec

4	23786	Overall time: 5.319619 sec
8	23794	Overall time: 2.692036 sec
16	23803	Overall time: 1.386517 sec

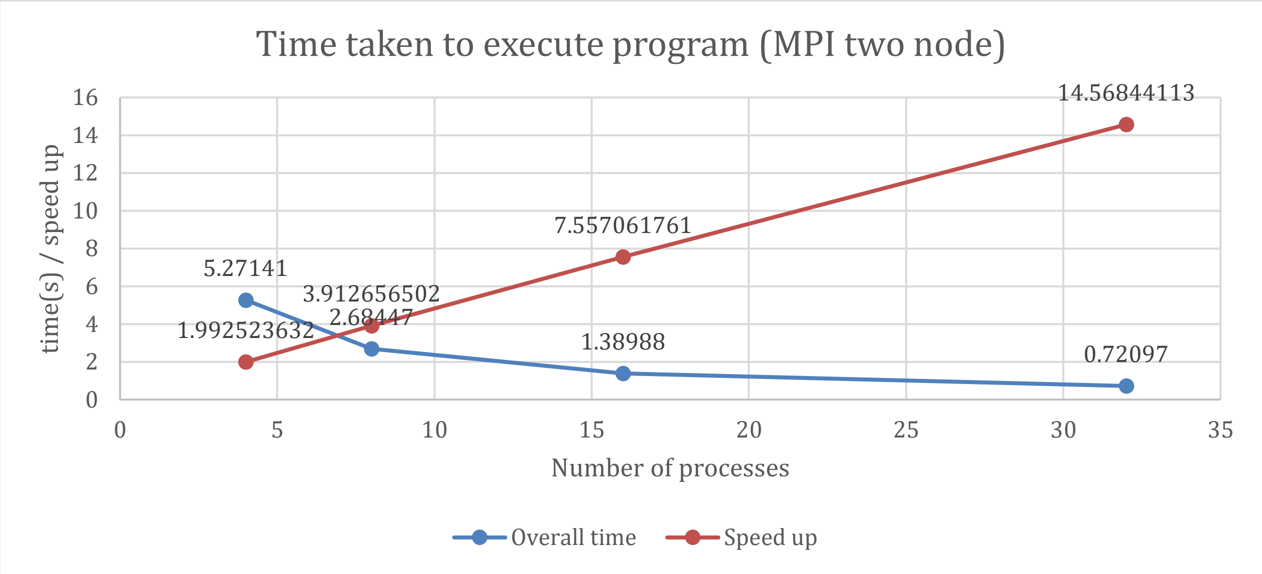


Above shows the graph representing MPI with one compute node. The speed up increases linearly when the number of processes increase. The highest speed up is 7.575 when 16 processes is created. However it is still less than the theoretical speed up which is 16. The shortest time taken is 1.3865 seconds for the program to execute finish when it is using 16 processes.

MPI with 2 compute node

MPI with 2 compute node is utilised, n = 10000000

Number of processes	Batch code	Overall time
4	23805	Overall time: 5.271416 sec
8	23806	Overall time: 2.684468 sec
16	23818	Overall time: 1.389880 sec
32	23822	Overall time: 0.720974 sec

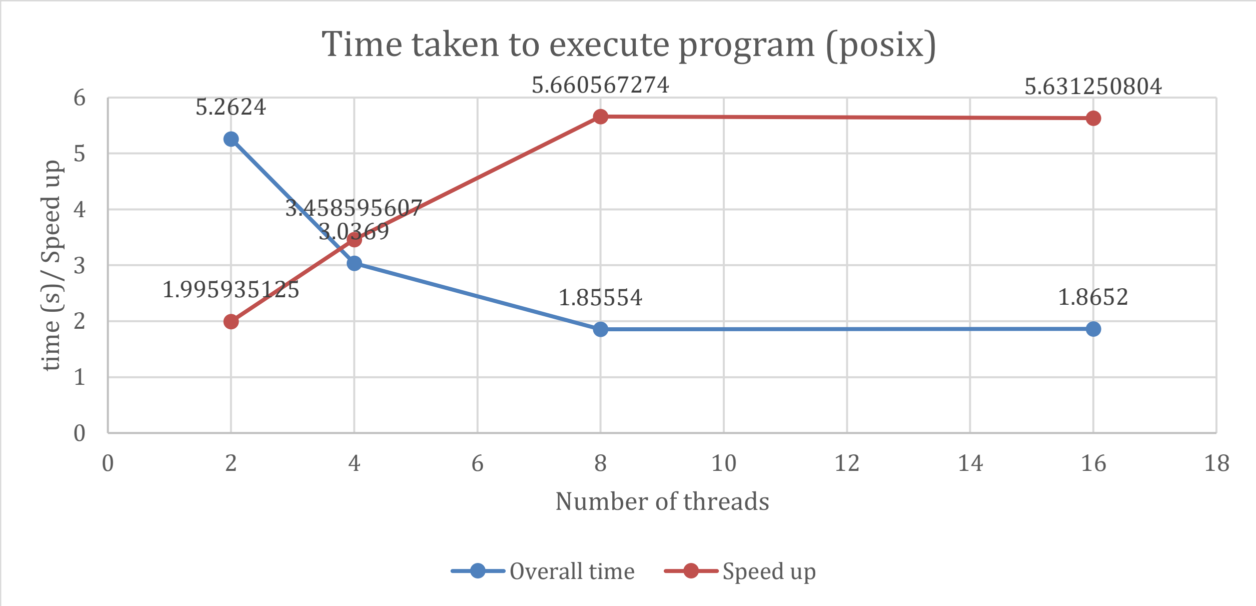


Above graph representing MPI executing the program with 2 compute nodes. The highest speed up is 14.5684 when 32 processes with 2 compute nodes is utilised. However it is still less than the theoretical speed up which is 32 times.

POSIX

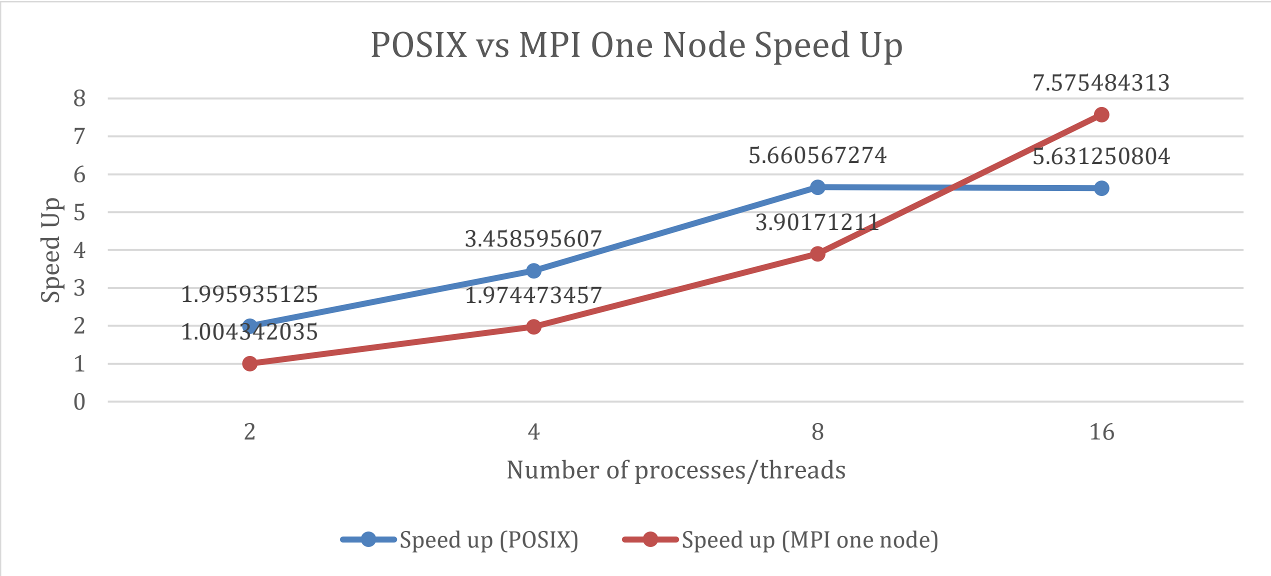
N = 10000000

Number of threads	Batch code	Overall time
2	23904	Overall time: 5.262403 sec
4	23908	Overall time: 3.036916 sec
8	23909	Overall time: 1.855540 sec
16	23914	Overall time: 1.865238 sec

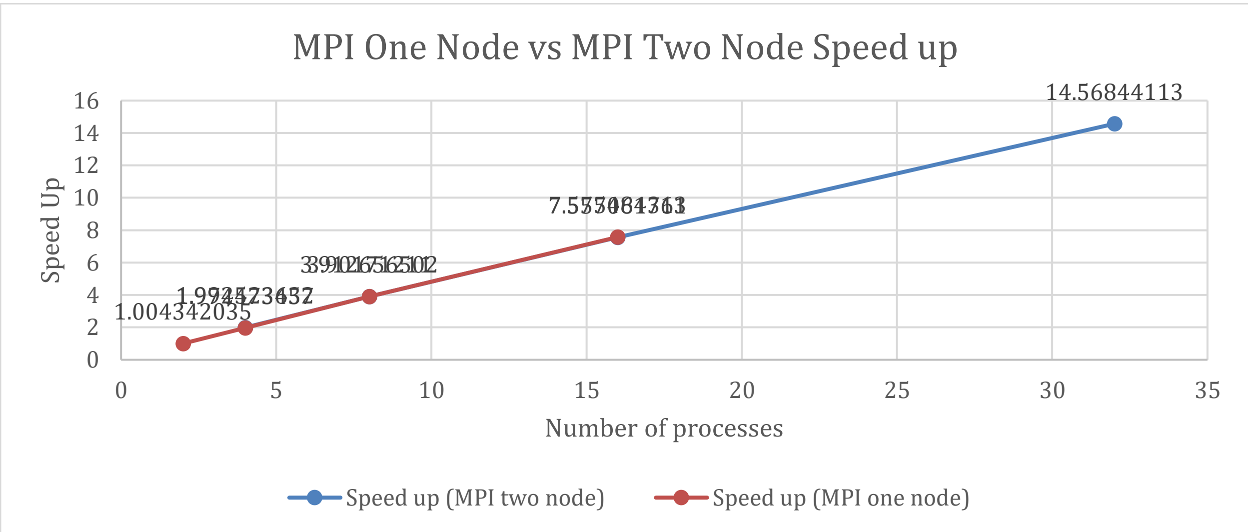


Above graph representing POSIX. The highest speed up is 5.6313 when 16 threads with 2 created. However it is still less than the theoretical speed up which is 16 times.

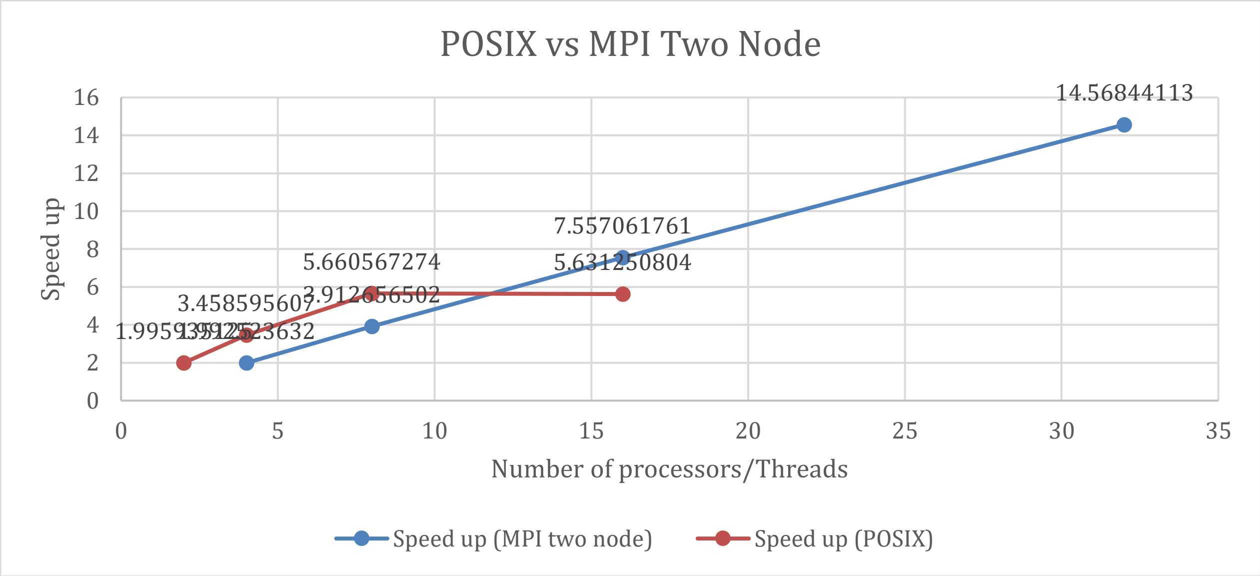
Comparison



Graph above representing the comparison of POSIX vs MPI one computational node speed up. MPI has a higher speed up when 16 processors is created. Whereas POSIX's speed up is always higher than MPI before 8 threads.



Graph above representing MPI one computational node vs MPI two computational node. In theoretical model, MPI with two computational node speed up should be way more higher than one computational node. But in actual results they are almost the same, some even have higher speed up than MPI two computational node. For instance, when 16 number of processors is created MPI with one computational node uses 1.3865s while MPI with two computational node uses 1.389880s.This might due to overhead when the nodes are communicating with one another.



Above graph represents POSIX vs MPI with two computational node. MPI with two computational node has higher speed up than POSIX starting from creating 16 processors/threads. Whereas POSIX highest speed up is 5.6605 when it creates 8 threads. However both of these results are less than the theoretical speed up.

Task 2 CAAS Q&A – Observations, results and explanation

Will CAAS be running faster on the same number of threads/processes against your machine?

It might take more time, as CAAS platforms often allocate resources from a shared pool. So if more users are accessing the platform at the same time, the resources will be shared among all the users. Moreover, if more users are running their tasks it may introduce additional overhead as it will schedule and allocate resources fairly among users. Thus, reduce performance.

From the result above the result of utilising two nodes to compute the prime number is almost the same as using one node. Not to mention, I have discovered there are tasks from other users in the queue when I check using “squeue”. So when multiple users access CAAS the network might congested and for MPI it requires communication between nodes. So this can lead to latency as suppose the two nodes result should have obvious differences than one node.

Will CAAS giving you a better speedup (against your machine) when the number of threads/processes become large? Why?

Yes, due to high performance of CPUs and GPUs. In CAAS monash provide us with high performance CPUs and GPUs that are way better than my machine. With more cores and more computational power. It can handle more computational tasks and process them faster. Thus, when the number of threads and processes become large it has more threads and processes help to process the data faster. Hence, it gives better speed up

Apart from that, CAAS uses high performance networks that offer very low latency and high bandwidth. So it minimises the latency when parallel processing is occurring like MPI message passing.

More memory allocation, in CAAS it allocates me with more RAM compared to what is available on my machine. More memory so that large datasets can load into the memory so reducing the need to swap data between RAM and storage. Moreover, CAAS can allocate more space for caching frequently accessed data. So the data that is repeatedly used can be stored in faster memory locations.