|  |
| --- |
| Coventry University |
| 6004CEM |
| Parallel Distributed Programming |

|  |
| --- |
| Gurvinder Nagra  SID: |

Contents

[Parallelism 1](#_Toc130815240)

[Distribution 2](#_Toc130815241)

[Parallel A 3](#_Toc130815242)

[Parallel B 4](#_Toc130815243)

[Parallel C 4](#_Toc130815244)

[Distributed A 5](#_Toc130815245)

[Distributed B 6](#_Toc130815246)

[Distributed C 6](#_Toc130815247)

[Bibliography 6](#_Toc130815248)

# Parallelism

Here I will be discussing the different types of granularities of parallelism and their effect in the real world, parallelism “is the use of multiple processing elements simultaneously for solving any problem” (universedecoder, 2021) . An example being using 4 cores over a single core to improve execution time for a program.

**Coarse and fine grain parallelism**

For coarse grained parallelism a problem is typically split into few parts this is so fewer processors will be required to complete the problem, this in turn can help to reduce the initial cost for system and communication needed between all the different nodes. Though bear in mind since fewer processors are utilised it will mean more clock cycles to complete each task and leads to a longer runtime. Since it is consistent of fewer components it is normally easier to implement into existing problems and as such easier to code for. (Wikipedia, 2023)

Fine grained parallelism however splits a problem into much smaller components allowing more processors to work on it in as few clock cycles as possible, this process will be much quicker though it will have a higher cost due to more nodes and communication required to be quick and efficient. (Wikipedia, 2023)

**The limits of parallel Programming**

Though parallel programming will help with great improvement for runtime and performance of a program it still possesses limitations on the capability of the process.

The lower limit of parallel computing depends on how much a program can be parallelised such programs which take basic inputs into a database, this can’t have any significant improvement in time because of the sequential method used to program which the code works with.

The upper limit for parallel programming in terms of core count and how many processing cores are able to be utilised effectively without changing the code is quite heavily influenced by managing the threads that come when utilising multiple processor cores. A program that uses more cores will need half of it to just handle threading as opposed to the actual task the program is meant to do. This lines up with Amdahl’s law which states that because not every single portion of the program can be distributed and as such won’t ever be linear meaning there will be diminishing returns at one point, since “ This effective limit, regardless of how well you write your program. is at around 32 cores.” (Pridgeon, 2021) All this means is a program needs to be built in a specific way to ensure that it utilises processors as best as it can because if only half of the program can be distributed then the runtime won’t be efficient and lead to worse diminishing returns. This could be an issue as it means data for certain programs will need to be distributed between nodes and that can only be done on the head node as it needs to send as well as collect processed data between all other nodes leaving processing overhead to be utilised.

# Distribution

In distributed programming we desire for performance to scale linearly though this isn’t always possible due to inefficient programs or lack of communication overhead. There are many different laws that can be referenced when scaling distribution such as:

Gustafson’s law, which is a revision of Amdahl’s Law which “says that increase of problem size for large machines can retain scalability with respect to the number of processors” (University of Minnesota) , this is based upon the initial execution time of the program with a single processor with more processors adding better yield for speed. The law shows that speedup is linear when nodes get properly utilised by a program showing unlike Amdahl’s law there can be a significant speedup when programs are written to use processors effectively rather than utilising all nodes as in Amdahl’s law.

Gunther’s universal scalability law, this law relates to purely speed loss when communicating with different nodes. If all nodes needed to talk with one another the number of links between them will exponentially increase. This can lead the program becoming harder to manage and leading to diminishing returns as well as losses because of how much communication a node may need to do within a single instance. Though this can be reduced and even avoided by using a head node to distribute data it will require the node to be much better at processing compared to other nodes. Instead of merely increasing communication between nodes this will reduce it to 1 to 1 communication so when more nodes get added to the distribution there will be more than a singular communication channel opened but if the head node can’t send data fast enough for all the nodes to function then the distribution will slow down overall. (University of Minnesota)

# Parallel A

A screenshot of a computer

Description automatically generated with medium confidence

#include "omp.h"

#include <iostream>

#include <cmath>

#include <cstdlib>

#include <string>

#include "stdio.h"

#include <list>

#include <chrono>

**using** **namespace** std;

**int** **main**(**void**){

#pragma omp parallel

{

**int** i=**0**;

**for**(**int** j=**0**; j<**10**; j++)

i=i+j;

}

**return** **0**;

}

# Parallel B

# Text Description automatically generated

#include "omp.h"

#include <iostream>

#include <cmath>

#include <cstdlib>

#include <string>

#include "stdio.h"

#include <list>

#include <chrono>

**using** **namespace** std;

// Show the vectors at the steps 0, 25 and 50

**void** **checkSteps**(**int** vectors[**10**][**3**], **int** i){

string stepString;

**if** (i == **0** || i == **24** || i == **49**) { // check if we are 50 steps in

stepString += "Steps in: " + to\_string(i+**1**) + "**\n**";

**for** ( **int** i = **0**; i < **10**; i++ ) **for** ( **int** j = **0**; j < **3**; j++ ) { //Iterate over the current matrix

stepString += to\_string(vectors[i][j]) + " ";

**if** (j == **2**){

stepString += "**\n**"; // create a new line per row of matix

}

}

cout << stepString + "**\n**"; //Split 25 and 50

}

}

// Get the average of all the particles for part b

**int** **getTargetX**(**int** vectors[**10**][**3**]){

**int** x = **0**;

**int** y = **0**;

**int** z = **0**;

**for** ( **int** i = **0**; i < **10**; i++ ){ //Iterate the over generated numbers

x += vectors[i][**0**]; //Add the x values

y += vectors[i][**1**]; //Add the y values

z += vectors[i][**2**]; //Add the z values

}

//Get the average of the x,y and z values

x = x/**10**;

y = y/**10**;

z = z/**10**;

// Display the average position of all the particles

cout << "**\n**Aiming for**\n**X: " + to\_string(x) + "**\n**Y: " + to\_string(y) + "**\n**Z: " + to\_string(z) + "**\n**" << endl;

**return** x;

}

// Get the average of all particles for part b

**int** **getTargetY**(**int** vectors[**10**][**3**]){

**int** x = **0**;

**int** y = **0**;

**int** z = **0**;

**for** ( **int** i = **0**; i < **10**; i++ ){ //Iterate over the generated numbers

x += vectors[i][**0**]; //Add the x values

y += vectors[i][**1**]; //Add the y values

z += vectors[i][**2**]; //Add the z values

}

//Get the average of the x,y and z values

x = x/**10**;

y = y/**10**;

z = z/**10**;

**return** y;

}

// Get the average of all particles in part b

**int** **getTargetZ**(**int** vectors[**10**][**3**]){

**int** x = **0**;

**int** y = **0**;

**int** z = **0**;

**for** ( **int** i = **0**; i < **10**; i++ ){ //Iterate over the generated numbers

x += vectors[i][**0**]; //Add the x values

y += vectors[i][**1**]; //Add the y values

z += vectors[i][**2**]; //Add the z values

}

x = x/**10**; //Get the average of the x values

y = y/**10**; //Get the average of the y values

z = z/**10**; //Get the average of the z values

**return** z;

}

// Display the particle position alteration variable

**void** **displayPPAV**(**int** ppav){

cout << "particle position alteration variable: " + to\_string(ppav) << endl;

}

**void** **getDistance**(**int** targetX, **int** targetY, **int** targetZ, **int** vectors[**10**][**3**]){

**int** distanceCalculation;

**for** ( **int** p = **0**; p < **10**; p++ ){

distanceCalculation = **0**;

//cout << distanceCalculation << endl;

distanceCalculation = ((targetX - vectors[p][**0**])^**2** + (targetY - vectors[p][**1**])^**2** + (targetZ - vectors[p][**2**])^**2**)^**1**/**2**;

**if** (distanceCalculation < **0**){

distanceCalculation = distanceCalculation\*-**1**;

}

cout << "Vector " + to\_string(p+**1**) << " (" << vectors[p][**0**] << " " << vectors[p][**1**] << " " << vectors[p][**2**] << ")" << endl;

cout << to\_string(distanceCalculation) + " away from the centre point**\n**" << endl;

}

};

**int** **main**(**void**){

**int** ppav = **2**; //Setting the particle position alteration variable to 2

**int** vectors[**10**][**3**] = { //Defining the vectors used

{**5**, **14**, **10**},

{**7**, -**8**, -**14**},

{-**2**, **9**, **8**},

{**15**, -**6**, **3**},

{**12**, **4**, -**5**},

{**4**, **20**, **17**},

{-**16**, **5**, -**1**},

{-**11**, **3**, **16**},

{**3**, **10**, -**19**},

{-**16**, **7**, **4**}

};

**for** (**int** i=**0**; i<**50**; i++){ //Iterate 50 times through i

// Modify the vectors randomly

#pragma omp parallel **for**

**for** ( **int** i = **0**; i < **10**; i++ ) **for** ( **int** j = **0**; j < **3**; j++ ) { //Iterate over the current matrix

**int** direction = rand() % **2**; //Generate a random value to see the direction of ppav

**if** (direction == **1**){ //If switchDirection is 1 then switch if ppav is positive or negative

vectors[i][j] = vectors[i][j] + ppav; //Add ppav to the current particle position

}

**else**{

vectors[i][j] = vectors[i][j] - ppav; //Subtract ppav to the current particle position

}

}

string stepString;

**if** (i == **24** || i == **49**) { //If we're 50 steps in

stepString += "Steps in: " + to\_string(i+**1**) + "**\n**";

**for** ( **int** i = **0**; i < **10**; i++ ) **for** ( **int** j = **0**; j < **3**; j++ ) { //Iterate over the current matrix

stepString += to\_string(vectors[i][j]) + " ";

**if** (j == **2**){

stepString += "**\n**"; //New line per row of matix

}

}

cout << stepString + "**\n**"; //Split 25 and 50

}

ppav = ppav + rand() % **5** + **1**; //Increase ppav

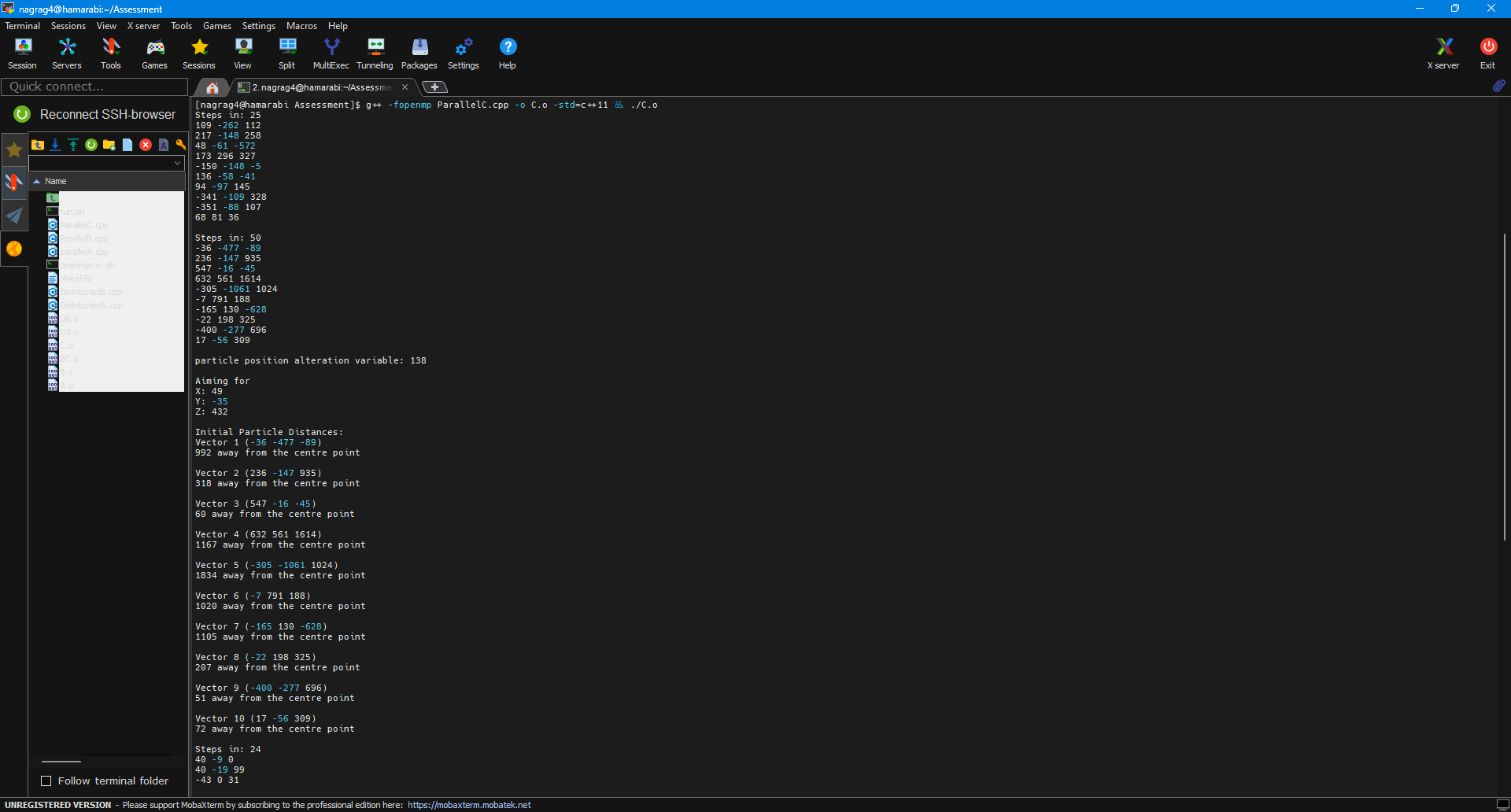
}

displayPPAV(ppav);

**return** **0**;

}

# Parallel C



A screenshot of a computer

Description automatically generated with medium confidence

#include "omp.h"

#include <iostream>

#include <cmath>

#include <cstdlib>

#include <string>

#include "stdio.h"

#include <list>

#include <chrono>

**using** **namespace** std;

// Show the vectors at the steps 0, 25 and 50

**void** **checkSteps**(**int** vectors[**10**][**3**], **int** i){

string stepString;

**if** (i == **0** || i == **24** || i == **49**) { // check if we are 50 steps in

stepString += "Steps in: " + to\_string(i+**1**) + "**\n**";

**for** ( **int** i = **0**; i < **10**; i++ ) **for** ( **int** j = **0**; j < **3**; j++ ) { //Iterate over the current matrix

stepString += to\_string(vectors[i][j]) + " ";

**if** (j == **2**){

stepString += "**\n**"; // create a new line per row of matix

}

}

cout << stepString + "**\n**"; //Split 25 and 50

}

}

// Get the average of all the particles for part b

**int** **getTargetX**(**int** vectors[**10**][**3**]){

**int** x = **0**;

**int** y = **0**;

**int** z = **0**;

**for** ( **int** i = **0**; i < **10**; i++ ){ //Iterate the over generated numbers

x += vectors[i][**0**]; //Add the x values

y += vectors[i][**1**]; //Add the y values

z += vectors[i][**2**]; //Add the z values

}

//Get the average of the x,y and z values

x = x/**10**;

y = y/**10**;

z = z/**10**;

// Display the average position of all the particles

cout << "**\n**Aiming for**\n**X: " + to\_string(x) + "**\n**Y: " + to\_string(y) + "**\n**Z: " + to\_string(z) + "**\n**" << endl;

**return** x;

}

// Get the average of all particles for part b

**int** **getTargetY**(**int** vectors[**10**][**3**]){

**int** x = **0**;

**int** y = **0**;

**int** z = **0**;

**for** ( **int** i = **0**; i < **10**; i++ ){ //Iterate over the generated numbers

x += vectors[i][**0**]; //Add the x values

y += vectors[i][**1**]; //Add the y values

z += vectors[i][**2**]; //Add the z values

}

//Get the average of the x,y and z values

x = x/**10**;

y = y/**10**;

z = z/**10**;

**return** y;

}

// Get the average of all particles in part b

**int** **getTargetZ**(**int** vectors[**10**][**3**]){

**int** x = **0**;

**int** y = **0**;

**int** z = **0**;

**for** ( **int** i = **0**; i < **10**; i++ ){ //Iterate over the generated numbers

x += vectors[i][**0**]; //Add the x values

y += vectors[i][**1**]; //Add the y values

z += vectors[i][**2**]; //Add the z values

}

x = x/**10**; //Get the average of the x values

y = y/**10**; //Get the average of the y values

z = z/**10**; //Get the average of the z values

**return** z;

}

// Display the particle position alteration variable

**void** **displayPPAV**(**int** ppav){

cout << "particle position alteration variable: " + to\_string(ppav) << endl;

}

**void** **getDistance**(**int** targetX, **int** targetY, **int** targetZ, **int** vectors[**10**][**3**]){

**int** distanceCalculation;

**for** ( **int** p = **0**; p < **10**; p++ ){

distanceCalculation = **0**;

//cout << distanceCalculation << endl;

distanceCalculation = ((targetX - vectors[p][**0**])^**2** + (targetY - vectors[p][**1**])^**2** + (targetZ - vectors[p][**2**])^**2**)^**1**/**2**;

**if** (distanceCalculation < **0**){

distanceCalculation = distanceCalculation\*-**1**;

}

cout << "Vector " + to\_string(p+**1**) << " (" << vectors[p][**0**] << " " << vectors[p][**1**] << " " << vectors[p][**2**] << ")" << endl;

cout << to\_string(distanceCalculation) + " away from the centre point**\n**" << endl;

}

};

**int** **main**(**void**){

**int** ppav = **2**; //Setting the particle position alteration variable to 2

**int** vectors[**10**][**3**] = { //Defining the vectors used

{**5**, **14**, **10**},

{**7**, -**8**, -**14**},

{-**2**, **9**, **8**},

{**15**, -**6**, **3**},

{**12**, **4**, -**5**},

{**4**, **20**, **17**},

{-**16**, **5**, -**1**},

{-**11**, **3**, **16**},

{**3**, **10**, -**19**},

{-**16**, **7**, **4**}

};

**for** (**int** i=**0**; i<**50**; i++){ //Iterate 50 times through i

// Modify the vectors randomly

#pragma omp parallel **for**

**for** ( **int** i = **0**; i < **10**; i++ ) **for** ( **int** j = **0**; j < **3**; j++ ) { //Iterate over the current matrix

**int** direction = rand() % **2**; //Generate a random value to see the direction of ppav

**if** (direction == **1**){ //If switchDirection is 1 then switch if ppav is positive or negative

vectors[i][j] = vectors[i][j] + ppav; //Add ppav to the current particle position

}

**else**{

vectors[i][j] = vectors[i][j] - ppav; //Subtract ppav to the current particle position

}

}

string stepString;

**if** (i == **24** || i == **49**) { //If we're 50 steps in

stepString += "Steps in: " + to\_string(i+**1**) + "**\n**";

**for** ( **int** i = **0**; i < **10**; i++ ) **for** ( **int** j = **0**; j < **3**; j++ ) { //Iterate over the current matrix

stepString += to\_string(vectors[i][j]) + " ";

**if** (j == **2**){

stepString += "**\n**"; //New line per row of matix

}

}

cout << stepString + "**\n**"; //Split 25 and 50

}

ppav = ppav + rand() % **5** + **1**; //Increase ppav

}

displayPPAV(ppav);

// PART C

ppav = **2**; //Reset the particle position alteration variable

**int** x = getTargetX(vectors);

**int** y = getTargetY(vectors);

**int** z = getTargetZ(vectors);

cout << "Initial Particle Distances:" << endl;

getDistance(x, y, z, vectors);

**for** ( **int** i = **0**; i < **10**; i++ ){ //Iterate over generated numbers

x += vectors[i][**0**]; //Add the x values

y += vectors[i][**1**]; //Add the y values

z += vectors[i][**2**]; //Add the values

}

x = x/**10**;

y = y/**10**;

z = z/**10**;

**for** (**int** i=**0**; i<**50**; i++){ //Iterate 50 times through i

string stepString;

**int** target = **0**;

/\* Modify the vectors towards centre point \*/

#pragma omp **for**

**for** ( **int** i = **0**; i < **10**; i++ ) **for** ( **int** j = **0**; j < **3**; j++ ) { //Iterate over the current matrix

**switch**(j){ //Select which coordinate to target

**case** **0**:

{**int** target = x;}

**case** **1**:

{**int** target = y;}

**case** **2**:

{**int** target = z;}

}

//cout << target << endl;

**if** (vectors[i][j] > target){

vectors[i][j] = vectors[i][j] - ppav; //Add ppav to the current particle position

}

**else**{

**if** (vectors[i][j] == target){

;

}

**else** {

vectors[i][j] = vectors[i][j] + ppav; //Subtract ppav to the current particle position

}

}

}

**if** (i == **24** || i == **49**) { //If we're 50 steps in

stepString += "Steps in: " + to\_string(i) + "**\n**";

**for** ( **int** i = **0**; i < **10**; i++ ) **for** ( **int** j = **0**; j < **3**; j++ ) { //Iterate over the current matrix

stepString += to\_string(vectors[i][j]) + " ";

**if** (j == **2**){

stepString += "**\n**"; //New line per row of matix

}

}

cout << stepString + "**\n**"; //Split 25 and 50

}

ppav = ppav + rand() % **5** + **1**; //Increase ppav

}

displayPPAV(ppav);

cout << "Final Particle Distances:" << endl;

getDistance(x, y, z, vectors);

**return** **0**;

}

# Distributed A

Text

Description automatically generated

#include <iostream>

#include "mpi.h"

#include <cstring>

#include <unistd.h>

#include <thread>

#include <cpuid.h>

#include <sys/sysinfo.h>

**using** **namespace** std;

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

**struct** sysinfo memInfo;

sysinfo (&memInfo); //Get the memory info

// Initialize the MPI environment

MPI\_Init(NULL, NULL);

**char** node\_name[MPI\_MAX\_PROCESSOR\_NAME];

**int** rank,size, namelen;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

MPI\_Get\_processor\_name(node\_name, &namelen);

memset(node\_name+namelen,**0**,MPI\_MAX\_PROCESSOR\_NAME-namelen);

**const** **auto** processor\_count = std::**thread**::hardware\_concurrency();

**long** **long** **int** memory = (**long** **long** **int**)memInfo.totalram \* memInfo.mem\_unit / **1024** / **1024**; //Get memory size from sys and convert to MB

sleep(rank); //Sleep for the rank of the node which causes it to display in order and in a nicer format

cout << " " << endl;

cout << "Rank: " << rank << " | Name: " << node\_name << " | Comm Size: " <<size << "| Cores: " << processor\_count << "| RAM: " << memory << "MB"<< endl;

system("lscpu | egrep 'CPU max MHz|CPU min MHz|CPU MHz'"); //Use system call to display the CPU speed

MPI\_Finalize();

}

# Distributed B

Text

Description automatically generated

#include <iostream>

#include <string>

#include <cstring>

#include <list>

#include "mpi.h"

**typedef** **struct** {

**float** list[**15**];

**int** size;

**char** letter;

**double** number;

**int** x;

**int** y;

} hostStruct;

MPI::Datatype createMpiWordType(){

//how many data types in the struct

**const** **int** count = **6**;

//type of every different block of data

MPI::Datatype typesInStruct[count] = {MPI::FLOAT,MPI::INT,MPI::CHAR,MPI::DOUBLE,MPI::INT};

//how many elements per block

**int** arrayBlockLengths [count] = {**15**,**1**};

//Now we need to specify starting memory location of each block, \*relative to the start of the struct only\*

// it's a bit of a tedious process, but neccesary.

MPI::Aint objAddress, address1,address2,address3,address4,address5,address6;

MPI::Aint arrayDisplacements[count];

hostStruct sbuf;//Just has to be a struct instance but not

// the one you're actually sending, since we only need those locations,

// which, since this is a static struct, will always be the same

objAddress = MPI::Get\_address(&sbuf);

address1 = MPI::Get\_address(&sbuf.list);

address2 = MPI::Get\_address(&sbuf.size);

address3 = MPI::Get\_address(&sbuf.letter);

address4 = MPI::Get\_address(&sbuf.number);

address5 = MPI::Get\_address(&sbuf.x);

address6 = MPI::Get\_address(&sbuf.y);

arrayDisplacements[**0**] = address1 - objAddress;

arrayDisplacements[**1**] = address2 - objAddress;

arrayDisplacements[**2**] = address3 - objAddress;

arrayDisplacements[**3**] = address4 - objAddress;

arrayDisplacements[**4**] = address5 - objAddress;

arrayDisplacements[**5**] = address6 - objAddress;

// now we create the MPI equivilent datatype using the data we just collected

MPI::Datatype mpiHostStruct;

mpiHostStruct = MPI::Datatype::Create\_struct(count,arrayBlockLengths,arrayDisplacements,typesInStruct);

// and commit it the the Communicator, so it can be used accross the entire cluster

mpiHostStruct.Commit();

**return** mpiHostStruct;

}

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

**int** com\_size, rank, namelen;

memset(receive.list, **0**, MPI\_MAX\_PROCESSOR\_NAME);

MPI\_Init(NULL,NULL);

**char** node\_name[MPI\_MAX\_PROCESSOR\_NAME];

MPI\_Comm\_size(MPI\_COMM\_WORLD, &com\_size);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

memset(node\_name, **0**, MPI\_MAX\_PROCESSOR\_NAME);

MPI\_Get\_processor\_name(node\_name, &namelen);

memset(node\_name+namelen,**0**,MPI\_MAX\_PROCESSOR\_NAME-namelen);

//Create an instance of the datatype you built

MPI::Datatype mpiHostStruct1 = createMpiWordType();

// fill it with dummy data (this part is overly verbose)

strcpy(list,"noName");

strcpy(mpiHostStruct1.list,list);

mpiHostStruct1.id = -**1**;

// nodes to use

**int** source = **0**;

**int** destination = **5**;

**if** (world\_rank == source){

MPI\_Send(&mpiHostStruct1,**1**, mpiHostStruct, destination, **0**, MPI\_COMM\_WORLD);

}

**if** (world\_rank == destination){

// make another copy for the destination node,

// calling it here so it's certain not to exist on the source node.

// no need to call a create method, as MPI will use

// this to duplicate the one it's transferring

mpiStruct local\_mpiHostStruct2;

**char** list[**15**];

strcpy(hostname,"noName");

std::cout << "> hello from "<< node\_name<< std::endl; //standard debugging code, remove later

MPI\_Recv(&local\_mpiHostStruct2, **1**, mpiHostStruct, source, **0**, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

std::cout << "> host structure Received by "<< node\_name<< std::endl;

// change the information in the data structure

strcpy(local\_mpiHostStruct2.list,node\_name);

local\_mpiHostStruct2.size,rank;

std::cout << "> host structure hostname changed to: " << local\_mpiHostStruct2.list << std::endl;

std::cout << "> host structure id changed to: " << local\_mpiHostStruct2.size << std::endl;

// now it's been changed we need to send it back to the source node,

// or rather, put it back into the communicator addressed to the head node ready to be collected

MPI\_Send(&local\_mpiHostStruct2, **1**, mpiHostStruct, source, **0**, MPI\_COMM\_WORLD);

}

// and now we repet the receive code, but this time for the source node

**if** (world\_rank == source){

// make a final copy for the source node so we are certain this is

// an altered version

mpiHostStruct local\_mpiHostStruct3;

std::cout << "> hello from "<< node\_name<< std::endl; //standard debugging code, remove later

MPI\_Recv(&local\_mpiHostStruct3, **1**, mpiHostStruct, destination, **0**, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

std::cout << "> host structure Received by "<< node\_name<< std::endl;

std::cout << "> host structure hostname changed to: " << local\_mpiHostStruct3.list << std::endl;

std::cout << "> host structure id changed to: " << local\_mpiHostStruct3.size << std::endl;

}

MPI\_Finalize();

**return** **0**;

}

# Bibliography

Pridgeon, C. (2021). Retrieved from https://github.coventry.ac.uk/pages/ab0475/6004CEM\_21-22\_Open/lectures/2\_limits.html#/slide-sec-

universedecoder. (2021, June 4th). *Introduction to parallel computing*. Retrieved from geeksforgeeks: https://www.geeksforgeeks.org/introduction-to-parallel-computing/

University of Minnesota. (n.d.). *Multi processor laws.* Retrieved from https://www.d.umn.edu/~tkwon/course/5315/HW/MultiprocessorLaws.pdf

Wikipedia. (2023, January 16th). *Granularity (parallel computing)*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Granularity\_(parallel\_computing)