Seminar Report

ZombieSim HPC Project Report

Abdullah Amawi
MatrNr: 11849696

Maaike Bierenbroodspot
MatrNr: 21622215

Supervised by: Prof. Dr. Julian Kunkel

Georg-August-Universität Göttingen
Institute of Computer Science

September 30, 2022

Abstract

In recent years, High-Performance-Computing(HPC) has been, and still is one of the top fields of computing in both research and industry due to its importance of utilization in many other fields that it supports, such as genomics, astrophysics, machine learning, big data and analysis, weather and climate sciences, and much more. This results that HPC is a highly important field in utilizing and researching it due to this importance of its usage in many crucial fields that needs HPC support for its own advancement. Because of the importance of the field of HPC, we aimed to study it through a practical manner in which we materialize in our project ZombieSim. In order to study HPC, researchers are faced with the crucial task of studying and understanding parallelization and the field of parallel computing in order to utilize it for use in HPC. In this report, we will introduce ZombieSim, the idea behind it, and how it would help in understanding and learning more about parallelization that would help us in obtaining some skills to use in HPC. On the other hand, we will note that the methodology and idea of our project revolves around making our project parallelized and serves the learning process of parallelism and HPC usage. Our implementation consists of three parts, ZombieSim program itself, a supplied configuration file that serves as a tool to make our program plug and play and very easy for researchers and hobbyists alike to be able to manipulate and see different simulations, and it incorporates a game engine that would help us to visualize our simulation in order to give it a real feel and makes it easier to understand and very easy to demonstrate changes to it. Then we would present out performance analysis to the program and its accompanying game engine. And even though that our journey in this practical project had its own challenges that we will also mention, it did still result in a successfully parallelized program that is also visualized with a game engine, and our program demonstrated good results in parallelism with minimal additional overheads using C++ and Boost library.

Contents

List of Tables

List of Figures

Listings

List of Abbreviations

HPC	High-Performance Computing
DL	Deep Learning

1 Introduction

In this report we will be introducing ZombieSim project; ZombieSim is a Zombie simulator to investigate how infections spread based on multiple factors to try to simulate how humans get infected with diseases based on factors such as behavior, environment, and biological factors. The goal of the project is to learn more about parallel computing and how to analyze parallel efficiency. On the other hand, we intend to make the factors in our simulator very easy to add, in order to allow the freedom of research and for the ease of manipulating factors. What led us to investigate parallel computing and to learn more about it is the fact that in recent years HPC systems have been gaining more and more traction in usage, not only due to the previous scientific computing use, but even further to recent success and huge amount of work in multiple fields such as in deep learning(DL) for example that also now utilizes HPC systems, and the challenge here is that in order to utilize an HPC system effectively and efficiently, we have to understand and learn more about parallelization, such mixture of usage of HPC and deep learning are applied to try to solve many important problems, such as weather analysis, high energy physics, and cosmology[Jia+18]. Regardless of the goals of any researcher on how they would want to use an HPC system, they have to start by learning a lot more about parallelization so they can employ their newly gained skills that they have to obtain in order to use an HPC system effectively, which is why we are trying to learn more about paralellization and sharing our project for researchers to be able to use it for similar goals.

2 Methodology

When it comes to the methodology that we used to approach solving the problem that we intended to be a way of understanding more about parallelism and why we need HPC systems by simulating a country that has multiple cities, and each one of those cities has a population that commute between the residential areas where they live to the business areas where they work. Our demo program simulates this in a setup that has three cities in one country, where each city has a specific amount of its own population, we also intended to make a configuration file that makes it very easy to adjust the setup of the cities, population, and all the other factors that we will introduce here. On the other hand, we will further explain our configuration file that we have, that we will supply with the demo to make it possible for researchers to modify and adjust how they see fit, our configuration file is very easy that even hobbyists with limited knowledge can adjust and see the change themselves in our demo that has its own visualization. More about the configuration file and how to use it will be listed in the implementation section.

After discussing the methodology that we will be using to approach how to make our simulation work, we were discussing how to decide on the general flow and structure of the program itself, and after a good time of consideration we concluded that the flow should start with initializing the variables, create the objects that we need for the simulation, such as the country, city, humans and their attributes. Then the next step would consist of running the simulation itself, such that the human objects can move around between their work environment and their homes, we would also have to track the living status of the humans, and if they get sick, we have to process that in a way that are they sick, until when, and also the mortality rate; On the other hand, we have to take care of the visualization using the olcPixelGameEngine [OLC]. as for the structure of the country, we have the country itself with three separate cities that would have the humans inside them, but as we noted earlier that the country attributes and the number of the cities and their attributes are easily changeable through the configuration file that we supply with the executable. Fig. 1 demonstrates ZombieSim flow and the structure of the country and cities.

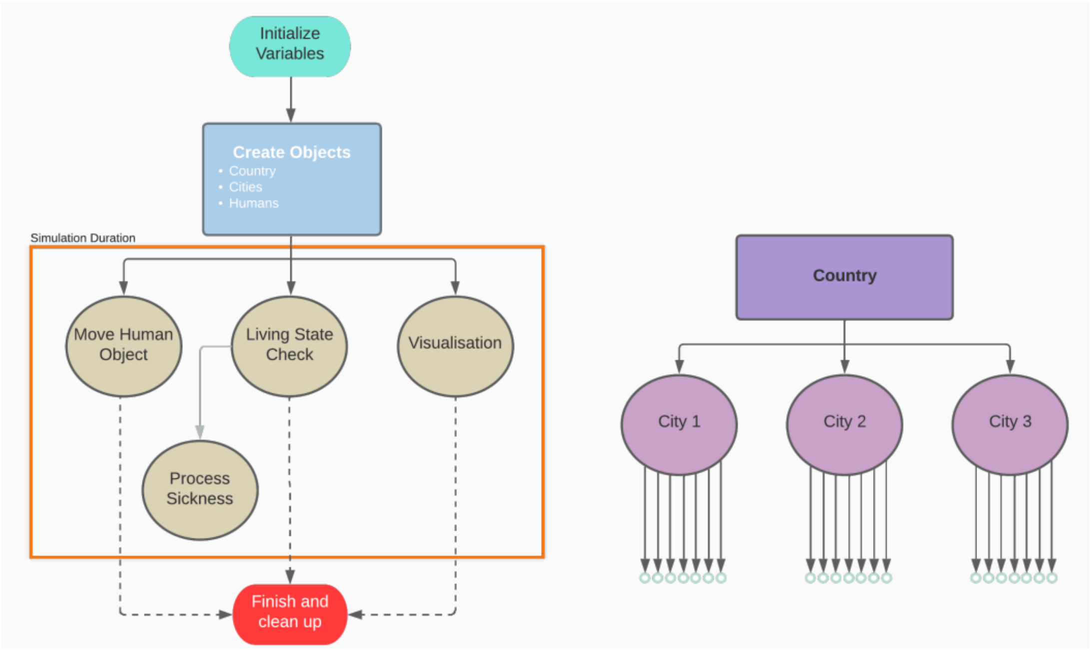

Figure 1: ZombieSim flow & structure

3 Implementation

Our implementation consists of three main parts, the program itself(ZombieSim), the supplemented configuration file, the physical game engine used for the visualization, we will explain them in the opposite order starting with the physical game engine. We will explain them in the opposite order so we can start with the easier part to understand, going up to the main, larger part.

3.1 The physical game engine

We used the olcPixelGameEngine[OLC] in our project to visualize the simulation that we implemented. Not only because this game engine serves the purpose that we intended for its use, but also for the simplicity of integrating it in our program. Using a game engine allowed us to visualize our program and see ourselves how our simulation acts and to verify that the intended purpose of the simulation is met in regards that we can visualize the cities inside a country, and we can clearly see that we have objects that signify the humans do indeed work properly and move in between their homes and work, and on the other hand, we can see that the infection simulation demonstrates what it was intended to do by visually observing that the human objects demonstrate that some are healthy and appear in green color, and some are sick and appear in red color, and how they do indeed infect each other by the intended rate.

3.2 The configuration file

Our configuration file gives a very easy way for researchers and anyone who uses our simulation a way to be able to adjust the parameters of the country, cities, humans, and the thread parallelism on the fly without the need of any knowledge of the specifics of the implementation of the C++ code, so it serves both as an abstraction for the implementation, and a plug and play tool for easy adjustments, we will list here the different inputs that can be adjusted with comments on what do they exactly change. The configuration file includes the following inputs that can be adjusted:

The Country.

```
[Country]

Size=500
Population=5000

StartSick=100   #How many are sick at the start.

CycleDuration=10 #In Seconds, half of the time population is in work
    state and other half is in home state recommended to at least 10.

#cycle completion = 1 day.

RateOtherCities=0 #rate of percentage of humans that have a different "
    work" address. The main city is where they "live"

FamilyUnit=4 #How many "humans" have for sure the same home "address".

MovingSpeed=5 #How smooth the Humans move from Work and Home (
    independent on frame rate); The higher the faster
```

Listing 1: Country in configuration file

The City

```
[Country]

CitiesAmount=3

Names=RaccoonCity,DimitrescusCastle,LosIluminados #just an example, it's
     comma delimeted'

Sizes=100,100,100
```

Listing 2: City in configuration file

The Humans

```
InfectionRate=5 #this is in percentage

DeathRate=1 This is in percentage

```

```
SickDurationTimesCycle=4 #How many cycles do the people stay sick.
```

Listing 3: Humans in configuration file

The Parallelism

```
ThreadBase=10 #How many threads are running over the human object loop
    they exist till the simulation is over

ThreadOverlapCheck=10 #How many threads are made by the ThreadBase
    threads, to check if different human objects overlap. They are remade
     after every overlap check set (based on the size of the human object
     vector)

SimulationDuration=120 #in seconds
```

Listing 4: Humans in configuration file

And we note that the configuration file will have to be placed with the executable file in order to be able to run the simulation. In order for anything to be adjusted as required, the user should only adjust the configuration file text to the state that they need or prefer, then run the executable that will use the configuration file with the added adjustments.

3.3 The program implementation

When it comes to the program implementation, this part is much larger than the previous two subsections, and that leads us to try to include the most crucial parts in here to explain how they work, we will structure this subsection starting with the entities that were mentioned in the configuration file, meaning, the country, city, humans, and the threads. We will demonstrate how they were implemented in C++.

Starting with the country code sample alongside its corresponding code comments for further explanation, and please note this is a sample for the sake of explanation, and the full code can be found in the provided repository, to that extent many details and libraries will be ignored, but can be found in the code repository.

```
//Inside the country.cpp,  we would also find the #include for the
    Country.h that has the city class with its own variables that would
    materialize what we explained earlier, such as the population, size,
    number of healthy and unhealthy humans, and so on. And all the
    Getters and Setters needed.

// When it comes to the Country.cpp functionality, this is a non-
    exhaustive sample for the major functions.

// We start with the Country itself:

long CCity::SetCountry(CCountry* pCountry)
  {
    if (pCountry == nullptr)
      return 1; //Todo Error message
    m_pCountry = pCountry;
    return ERR_NOERROR;
  }

```

```
CCountry * CCity::GetCountry() { return m_pCountry; }
long CCity::GetCountry(CCountry** ppCountry)
{
  if (m_pCountry == nullptr)
    return 1; //ToDO Error Message

  *ppCountry = m_pCountry;
  return ERR_NOERROR;
}


//then the SetCityName function to set the names of the cities, they can be manipulated in the config file as mentioned earlier.

long CCity::SetCityName(char * pCityName, size_t sizeToCopy)//init function
{
    if (m_pCityName)
        delete m_pCityName;

    if (sizeToCopy <= 0)
        return 1; //ToDo Error handling

    m_pCityName = new char[sizeToCopy + 10];
    memset(m_pCityName, '\0', (sizeToCopy + 10)); //zero terminate the string
    memcpy(m_pCityName, pCityName, sizeToCopy); //Copy the data into the buffer

    return ERR_NOERROR;

}

//After that we have to set the position and size of the city in SetCitypositions function.

long CCity::SetCityPositions(float fCitySize, float px, float py) //Init function doesn't have to be thread safe
{
  m_mutex.lock();
    if (fCitySize <= 0.0)
        return 1; //ToDO ErrorHandling

    if(px < 0.0)
        return 1; //ToDO ErrorHandling

    if(py < 0.0)
        return 1; //ToDO ErrorHandling

    m_fCitySize = fCitySize;
    m_px = px;
    m_py = py;
  m_mutex.unlock();

    return ERR_NOERROR;
}

// then the details in the code for getting and setting each city size
```

```
and name and so on.

// then we get the healthy, sick population that they were preset, and their corresponding functions.

//now we use SetPopulation function, which is a init function that is not thread related, since there is a lot of usage of the different variables and functions, we will list the complete SetPopulation function.

long CCountry::SetPopulation()
 {
  long lReturn = ERR_NOERROR;

  //Set population size
  size_t sizeHumanPopulation = 0;;
  lReturn = glb.propertyBag.GetPopulationSize(&sizeHumanPopulation);
  m_sizePopulation = (unsigned)sizeHumanPopulation; //population can never be negative.
  if (lReturn != ERR_NOERROR)
    return GET_POPULATION_SIZE;

  long lCities;


  //Set healthy group
  m_sizeHealthy = m_sizePopulation;
  m_sizeSick = 0;
  m_sizeDeath = 0;

  //Now the population has to be divided over the amount of cities ( for now we are just going to equally divide it.
  //In the future we reduce the city size amount to increase the population density.

  //Get the amount of cities based on the vector. If vector size is = 0 call Set Cities
  //Call it again af it is again 0, return error message.

  size_t sizeCities = m_vecCities.size();
  if (sizeCities <= 0)
    lReturn = SetCities();

  if (lReturn != ERR_NOERROR)
    return lReturn;

  if (sizeCities <= 0)
    return 1; //ToDo Error handling


  /*
  Now we are going to fill the human object vectorand set the City* within them.
  We are going to get the amount of humans per city Humans/City.
  Where the last city gets the remaining humans when we have a fraction.

  */
```

```

 bool bFraction = false;

 double dFraction = (double)(sizeHumanPopulation/ sizeCities);
 double dFractionSubstract = dFraction - static_cast<long long>(dFraction);
 if (dFraction > 0.0)//if it's a fraction or not!
   bFraction = true;

 long long llDivide = static_cast<long long>(dFraction);
 CHuman* pHuman = nullptr;
 CCity* pCity = nullptr;

 //Going to create Human Objects and set the City Pointers.
 for (size_t idx = 0; idx < sizeCities; idx++)
 {
   pCity = m_vecCities.at(idx);
   for (long long idxHuman = 0; idxHuman < llDivide; idxHuman++)
   {
     pHuman = new CHuman;
     lReturn = pHuman->SetCityPointer(pCity);
     if (lReturn != ERR_NOERROR)
       return lReturn;
     m_vecHumans.push_back(pHuman);
     pCity->PushBack(pHuman); //this is just for easy access/calculations.
   }
 }

 //if we have a fraction we have some remaining humans they are added to the last one
 if (bFraction == true)
 {
   size_t sizeHumanPushed = (size_t)llDivide * sizeCities;
   size_t sizeRemainingToBePushed = sizeHumanPopulation - sizeHumanPushed;

   pCity = m_vecCities.at(sizeCities - 1);

   for (size_t idxHumanToBePushed = 0; idxHumanToBePushed < sizeRemainingToBePushed; idxHumanToBePushed++)
   {
     pHuman = new CHuman;
     lReturn = pHuman->SetCityPointer(pCity);
     if (lReturn != ERR_NOERROR)
       return lReturn;
     m_vecHumans.push_back(pHuman);
     pCity->PushBack(pHuman); //this is just for easy access/calculations.
   }
 }

 //So at this point we have all the cities and all the humans.
 //But the humans do not have a home address or a work address time to set those.


 //Step 1, Set the Home Address and work address in the same City.
```

```
 //First we have to know how big a human family unit is. Aka how many humans live for sure on the same spot.
 long lHumanFamilyUnit = 0;
 lReturn = glb.propertyBag.GetFamilyUnit(&lHumanFamilyUnit);
 if (lReturn != ERR_NOERROR)
   return lReturn;

 if (lHumanFamilyUnit <= 0)
   lHumanFamilyUnit = 2; //Just as a default number in case someone added something weird.

 //Now we do the same thing as we did for the cities. We need to know if we have a fraction or not. The remaining family will be Smaller. (lucky them!
//Get the Respective pCity, since we have to know how many family units we have, so we round it up
//eg we have 9 humans, family unit = 4. Divide is 2.25 We will have 3 family units. 4,4,1

 size_t sizeHumanVectorInCity = 0;
 size_t sizeRoundUpFamilyUnit = 0;
 size_t idxHumanVector = 0;

 float fCityX = 0.0;
 float fHumanHomeX = 0.0;
 float fHumanWorkX = 0.0;
 float fCityY = 0.0;
 float fCityY2 = 0.0;
 float fHumanHomeY = 0.0;
 float fHumanWorkY = 0.0;
 float fCitySize = 0.0;

 std::random_device RNG;  // Will be used to obtain a seed for the random number engine
 std::mt19937 Seed(RNG()); // Standard mersenne_twister_engine seeded with RNG()


 for (size_t idxCity = 0; idxCity < sizeCities; idxCity++)
 {
   idxHumanVector = 0; //Reset it

   pCity = m_vecCities.at(idxCity);
   sizeHumanVectorInCity = pCity->GetHumanVectorSize();
   if (sizeHumanVectorInCity <= 0)
     return 1; //ToDo Error Handling.

   fCitySize = pCity->GetCitySize();
   fCityX = pCity->GetCityX();
   fCityY = pCity->GetCityY();
   fCityY2 = fCityY + (fCitySize * 0.5);

   std::uniform_real_distribution<> XCoordinateHome(fCityX, (fCityX + fCitySize));
   std::uniform_real_distribution<> XCoordinateWork(fCityX, (fCityX + fCitySize));
   std::uniform_real_distribution<> YCoordinateHome(fCityY, fCityY2);
   std::uniform_real_distribution<> YCoordinateWork(fCityY2, (fCityY
```

```
+ fCitySize));

   sizeRoundUpFamilyUnit = (size_t)std::round(((long double)sizeHumanVectorInCity / (long double)lHumanFamilyUnit));
   for (size_t idxFamilyUnit = 0; idxFamilyUnit < sizeRoundUpFamilyUnit; idxFamilyUnit++)
   {

     fHumanHomeX = XCoordinateHome(Seed); //stay for every family unit the same;
     fHumanHomeY = YCoordinateHome(Seed);

     for (long lFamilyUnitMember = 0; lFamilyUnitMember < lHumanFamilyUnit; lFamilyUnitMember++)
     {
       if (idxHumanVector < sizeHumanVectorInCity)
       {
         pHuman = pCity->GetHuman(idxHumanVector);
         fHumanWorkX = XCoordinateWork(Seed);
         fHumanWorkY = YCoordinateWork(Seed);

         lReturn = pHuman->InitPositions(fHumanHomeX, fHumanHomeY, fHumanWorkX, fHumanWorkY, fHumanHomeX, fHumanHomeY);
         if (lReturn != ERR_NOERROR)
           return lReturn;

         idxHumanVector++;
       }
       else
         break;

     }
   }



 }

 //We have no set the positions of all our human objects.
 //Now we have some randomization that some humans go visit other cities.
 //Step 2 Set % of humans to work in other cities;
 long lRateOfDifferentCityWorkAddress = 0;
 lReturn = glb.propertyBag.GetRateOfDifferentWorkAddress(&lRateOfDifferentCityWorkAddress);
 if (lReturn != ERR_NOERROR)
   return lReturn;

 //So every city have x% members that work in a different city. We round it down
 //With 1 City -> no randomization
 //With 2 City -> City 1 and City 2 swap.
 //With 3 cities -> City1 -> City 2, City 2 -> City 3 and City 3 -> City 1. Etc.

 size_t sizeCityLoopRuns = 0;

 CCity* pMoveToCity = nullptr;
```

```
    size_t idxCityToMove = 0;
    size_t idxCityToMovePos = 0;
    size_t sizeRateOfDifferentCityWorkAddress = (size_t)lRateOfDifferentCityWorkAddress;

    if (sizeCities >= 2)
    {
      for (size_t idxCity2 = 0; idxCity2 < sizeCities; idxCity2++)
      {
        idxCityToMovePos = 0;
        pCity = m_vecCities.at(idxCity2);
        if (idxCity2 + 1 == sizeCities)
          idxCityToMove = 0;
        else
          idxCityToMove = idxCity2 + 1;

        pMoveToCity = m_vecCities.at(idxCityToMove);
        sizeCityLoopRuns = std::round((float) pCity->GetHumanVectorSize() / (float)lRateOfDifferentCityWorkAddress);

        fCitySize = pMoveToCity->GetCitySize();
        fCityX = pMoveToCity->GetCityX();
        fCityY = pMoveToCity->GetCityY();
        fCityY2 = fCityY + (fCitySize * 0.5);

        std::uniform_real_distribution<> XCoordinateWork2(fCityX, (fCityX + fCitySize));
        std::uniform_real_distribution<> YCoordinateWork2(fCityY2, (fCityY + fCitySize));

        for (size_t idxCityLoop = 0; idxCityLoop < sizeCityLoopRuns; idxCityLoop++)
        {
          pHuman = pCity->GetHuman(idxCityToMovePos);
          fHumanWorkX = XCoordinateWork2(Seed);
          fHumanHomeY = YCoordinateWork2(Seed);

          lReturn = pHuman->SetWork(fHumanWorkX, fHumanWorkY);

          idxCityToMovePos = idxCityToMovePos + sizeRateOfDifferentCityWorkAddress;
        }
      }
    }

    //Done
    return lReturn;
  }



//After that we move on to SetCities function to set the cities.

long CCountry::SetCities()
{
  long lReturn = ERR_NOERROR;

  long lCitiesAmount = 0;
```

```
lReturn = glb.propertyBag.GetCitiesAmount(&lCitiesAmount);
if (lReturn != ERR_NOERROR)
  return 0; //ToDo Error Message

CCity* pCity = nullptr;
for (size_t idx = 0; idx < lCitiesAmount; idx++) //create city objects
{
  pCity = new CCity;
  m_vecCities.push_back(pCity);
}

std::string strCityNames;
lReturn = glb.propertyBag.GetCitiesName(&strCityNames);
size_t sizeString = strCityNames.length();

long lCount = 0; //Keep track of how many names there are;
char* pPos;

char* szString = &strCityNames.at(0);
char* pEnd = szString + sizeString -1 ;
pPos = szString;
while (pPos != pEnd && pPos) //While pPos is not equal to PEnd and pPos exist, should never reach pass the pEnd, but just to be sure!
{
  if (*pPos == ',')
  {
    if (pPos == pEnd)//in case the user used a comma at the end of the line
      break;

    pCity = m_vecCities.at(lCount);
    lReturn = pCity->SetCityName(szString, pPos - szString);
    if (lReturn != ERR_NOERROR)
      return 1; //ToDo Error Message;

    lCount++;
    if (lCount == lCitiesAmount)
      break;


    pPos++; //Set it 1 past the ,
    szString = pPos; //Set it to the
  }
  pPos++;
}

if (pPos == pEnd) //get the last item
{
  pCity = m_vecCities.at(lCount);
  if(*pPos == ',')
    lReturn = pCity->SetCityName(szString, pPos - szString);
  else
    lReturn = pCity->SetCityName(szString, pPos - szString+1);
  if (lReturn != ERR_NOERROR)
    return 1; //ToDo Error Message;

  lCount++;
}
```

```

//Check if all cities are filled if not add default name
std::string strCityNameDefault;
if (lCount < lCitiesAmount)
{
  while (lCount < lCitiesAmount)
  {
    pCity = m_vecCities.at(lCount);
    strCityNameDefault = "CityName" + std::to_string(lCount+1);
    lReturn = pCity->SetCityName(&strCityNameDefault[0], strCityNameDefault.length());
    lCount++;
  }
}

//Now lets determine the sizes of the cities and the positions!
//check how we have to divide the space of the screen based on our number of cities.
//Since the screen is a perfect square we can determine how much cities have to fit vertically and horizontally
float fSpaceDivide = ceil(sqrt((float)lCitiesAmount)); //we have to round it up not down. 8 cities will result in a square of 3 x 3 cities, with one space being unoccupied but that's fine

float fScreenSize = 0.0;

lReturn = glb.propertyBag.GetCountrySize(&fScreenSize);
if (lReturn != ERR_NOERROR)
  return 1; //TodoErrorHandling

float fCitySize = round(fScreenSize / fSpaceDivide) - 10; //We round it down for quick calculation AND give a gap of 10 so that the cities are not touch one another.

float fStartPosX = 0.0;
float fStartPosY = 0.0;

//Fill the positions and sizes in the cities;
lCount = 0;
for (float idxPosY = 0; idxPosY < fSpaceDivide; idxPosY++)
{
  for (float idxPosX = 0; idxPosX < fSpaceDivide; idxPosX++)
  {

    pCity = m_vecCities.at(lCount);
    lReturn = pCity->SetCityPositions(fCitySize, ((fCitySize * idxPosX) + (10 * idxPosX)), ((fCitySize * idxPosY) + (10 * idxPosY)));
    if (lReturn != ERR_NOERROR)
      return lReturn;

    lCount++;
    if (lCount == lCitiesAmount)
      break;
  }
}


return lReturn;
```

```cpp

}


//SetLivingStatuses function specifies the default status of the humans and the sickness spread, but on the other hand please note that have the config file to change those parameters.

long CCountry::SetLivingStatuses()
{
  long lReturn = ERR_NOERROR;

  long lDefaultSick = 0;
  lReturn = glb.propertyBag.GetStartSick(&lDefaultSick);
  if (lReturn != ERR_NOERROR)
    return 1; //ToDo Error Handling

  if (lDefaultSick == 0)
    lDefaultSick = 1; //As a default

  //Have to be spread equally;
  size_t SizeSpreadRate = (size_t) std::round(m_vecHumans.size() / (size_t)lDefaultSick); //we round it down to make sure we don’t try to access non existing vector objects.

  size_t sizeSpreadPos = 0;
  CHuman* pHuman = nullptr;
  for (size_t idx = 0; idx < (size_t)lDefaultSick; idx++)
  {
    pHuman = m_vecHumans.at(sizeSpreadPos);
    pHuman->SetLivingStatus(LIVINGSTATUS::SICK);
    sizeSpreadPos = sizeSpreadPos + SizeSpreadRate;

  }

  return lReturn;
}

After initializing all of this, at this point the thread starts by getting the population, the rest of the details can be found in Country.cpp file in the project.
```

Listing 5: Country file in C++

When it comes to the City implementation, as we demonstrated, the city class is inside the Country.h, and the City name and other needed functionality is inside the Country.cpp functions. And now we will move to the implementation of the humans in the project.

```cpp
//As for the Humans, you will find a similar structure of both Humans.cpp and Humans.h files in the project, with the Humans class and other pointers in the Humans.h file, and the functionality inside the Humans.cpp file in the project.

//After including Humans.h and Threads.h, the main functionality in Humans.cpp is to process the sickness of the humans, the related functions, and also to deal with their positioning and movement as explained earlier in the design, that the humans move from their homes to work and vice versa, and of course along that they infect
```

```
each other.


// The function to process the sickness of the humans
void CHuman::ProcessSickness(std::vector<CHuman*>* pvecHumans)
{
  //First we check if the human actually died or not
  static std::random_device RNG;  // Will be used to obtain a seed for the random number engine
  static std::mt19937 Seed(RNG()); // Standard mersenne_twister_engine seeded with RNG()
  static std::uniform_real_distribution<> disDeathRate(0, 100);
  static float fDeathRate = (float)glb.propertyBag.GetDeathRate() * 0.1;
  float fDeathSet = disDeathRate(Seed);


  if (fDeathSet <= fDeathRate)
  {
    Kill(); //Human died :(
    return;
  }


  //Now we have to find the overlapping Healthy objects
  static uint64_t u64CycleSicktime = (uint64_t) glb.propertyBag.GetCycleTime()* 3;

  m_mutex.lock();
  static size_t sizeThreadOverlapCheck = (size_t) glb.propertyBag.GetThreadOverlapCheck();
  static long SizeHumanVector = pvecHumans->size();
  m_mutex.unlock();

  size_t sizeDivideWorkload = (size_t)round(((float)SizeHumanVector / (float)sizeThreadOverlapCheck));

    size_t sizeStartPos = 0;
    size_t sizeEndPos = 0;
    CZombieSimThreadSickness* pThread = nullptr;
    std::vector<CZombieSimThreadSickness *> vecThreads;


    for (size_t idxThreadPos = 0; idxThreadPos < sizeThreadOverlapCheck; idxThreadPos++)
    {
        sizeStartPos = idxThreadPos * sizeDivideWorkload;
        sizeEndPos = sizeStartPos + sizeDivideWorkload - 1;
    if (idxThreadPos == (sizeThreadOverlapCheck - 1))
    {
      sizeEndPos = SizeHumanVector - 1;
    }
        //Start Thread

        pThread = new CZombieSimThreadSickness(sizeStartPos, sizeEndPos, this);
        pThread->SetThreadProc(new boost::thread(ThreadProcessSickness, pThread));
    ASSERT(pThread->GetThreadProc());
```

```

  //Give is time to initialize
  boost::this_thread::sleep_for(boost::chrono::milliseconds(250));

  //And add the info to the vector of running threads
    vecThreads.push_back(pThread);
  }

#ifdef _DEBUG

 printf_s("Now we wait till all processes are done %s", EOL);

#endif

 WaitForAllThreads(vecThreads);
  //Now we delete them
  ClearVector<CZombieSimThreadSickness>(&vecThreads);

 //SetHealthyOrNot
 boost::posix_time::ptime SickCurrenttime = boost::posix_time::second_clock::local_time();
 boost::posix_time::time_duration SickDuration = SickCurrenttime - this->GetSickStartTime();

 if (((uint64_t) SickDuration.total_seconds() )> u64CycleSicktime)
  SetLivingStatus(GETHEALTHY);


}

// Then we have the functions that get the humans and their living status for further use later on, We will skip their details here for the same of simplicity, everything can be found in the same sequence of the report structure, meaning that they can be found after the ProcessSickness function.

// Then we initialize the humans positions as in their home, work, current so we can move them to work and back to their homes.

// Movement, starting with moving to work:
void CHuman::MoveToWork()
{

 m_mutex.lock();
 //First check if the person is not dead, otherwise we have to do all this code for nothing;
 if (m_eLivingStatus == DEAD)
 {
  m_mutex.unlock();
  return;
 }

 static float fMovingSpeed = glb.propertyBag.GetCycleTime() * 0.05; // 5% of your current cycle time.
 float fElapsedTimeObject = glb.fElapsedTimeProperty;

 float fxDistanceToWork = this->GetDistanceWorkX();
 float fyDistanceToWork = this->GetDistanceWorkY();
```

```

  boost::posix_time::ptime CurrentTime = boost::posix_time::second_clock
    ::local_time();
  boost::posix_time::time_duration TimeDiff = CurrentTime -
    m_MoveStarttime;

  //m_fxCurrentPos = m_fxCurrentPos + ((abs(m_fxHome - m_fxCurrentPos) *
     (fElapsedTimeObject / 2.0)));
  m_fxCurrentPos = m_fxCurrentPos + (-1 * fMovingSpeed * (m_fxCurrentPos
     - m_fxWork) * fElapsedTimeObject);
  //m_fyCurrentPos = m_fyCurrentPos + ((abs(m_fyHome - m_fyCurrentPos) *
     (fElapsedTimeObject / 2.0)));
  m_fyCurrentPos = m_fyCurrentPos + (-1 * fMovingSpeed * (m_fyCurrentPos
     - m_fyWork) * fElapsedTimeObject);

  //m_fxCurrentPos = fxDistanceToWork * (
  // (float)TimeDiff.total_seconds() / fMovingSpeed);
  //m_fyCurrentPos = fyDistanceToWork * ((float)TimeDiff.total_seconds()
     / fMovingSpeed);
  //m_fxCurrentPos = fxDistanceToWork * (fElapsedTimeObject /
    fMovingSpeed);
  //m_fyCurrentPos = fyDistanceToWork * (fElapsedTimeObject /
    fMovingSpeed);

  if(((float)TimeDiff.total_seconds()) >= fMovingSpeed)
  {
    m_fxCurrentPos = m_fxWork;
    m_fyCurrentPos = m_fyWork;
    m_eMoveStatus = STATIC;
  }

  m_mutex.unlock();


}

// Then the humans will go back to their homes. Please note as mentioned
     multiple times, the time cycle and all time-related variables can be
     easily manipulated in the configuration file that we mentioned in
    the start of the implementation section so the reader can easily
    connect it to the program implementation.

void CHuman::MoveToHome()
{

  m_mutex.lock();
  //First check if the person is not dead, otherwise we have to do all
    this code for nothing;
  if (m_eLivingStatus == DEAD)
  {
    m_mutex.unlock();
    return;
  }

  static float fMovingSpeed = glb.propertyBag.GetCycleTime() * 0.05; //
    5% of your current cycle time.
  float fElapsedTimeObject = glb.fElapsedTimeProperty;

```

```cpp
  float fxDistanceToHome = this->GetDistanceHomeX();
  float fyDistanceToWork = this->GetDistanceHomeY();

  boost::posix_time::ptime CurrentTime = boost::posix_time::second_clock
    ::local_time();
  boost::posix_time::time_duration TimeDiff = CurrentTime -
    m_MoveStarttime;

  //m_fxCurrentPos = m_fxCurrentPos + ((abs(m_fxWork - m_fxCurrentPos) *
     (fElapsedTimeObject / 2.0)));
  m_fxCurrentPos = m_fxCurrentPos + (-1 * fMovingSpeed * (m_fxCurrentPos
     - m_fxWork) * fElapsedTimeObject);
  //m_fyCurrentPos = m_fyCurrentPos + ((abs(m_fyWork - m_fyCurrentPos) *
     (fElapsedTimeObject / 2.0)));
  m_fyCurrentPos = m_fyCurrentPos + (-1 * fMovingSpeed * (m_fyCurrentPos
     - m_fyWork) * fElapsedTimeObject);

  //m_fxCurrentPos = fxDistanceToWork * (
  // (float)TimeDiff.total_seconds() / fMovingSpeed);
  //m_fyCurrentPos = fyDistanceToWork * ((float)TimeDiff.total_seconds()
     / fMovingSpeed);
  //m_fxCurrentPos = fxDistanceToWork * (fElapsedTimeObject /
    fMovingSpeed);
  //m_fyCurrentPos = fyDistanceToWork * (fElapsedTimeObject /
    fMovingSpeed);


  //m_fxCurrentPos = fxDistanceToHome * (fElapsedTimeObject /
    fMovingSpeed);
  //m_fyCurrentPos = fyDistanceToWork * (fElapsedTimeObject /
    fMovingSpeed);

  if(((float)TimeDiff.total_seconds()) >= fMovingSpeed)
  {
    m_fxCurrentPos = m_fxHome;
    m_fyCurrentPos = m_fyHome;
    m_eMoveStatus = STATIC;
  }

  m_mutex.unlock();


}

// The mentioned functions above are the main functions in the Humans.
    cpp implementation, but of course, there are a lot of other smaller
    functions that deal with the details, that we mentioned, and more
    such as for example how to deal with the death toll, the percentage
    and how some humans will die due to sickness, provided by the death
    toll percentage. All those details can be found at the end of the
    file after the movement functions.
```

Listing 6: Humans C++

4 Performance analysis

When it comes to the performance analysis we utilized Hotspot[Hot] tool that is used in Linux that provides a GUI for performance analysis. The main functionality of Hotspot is to visualize the performance data graphically in what the creators call "Flame Graph", flame graph basically shows the timeline of the program that is running with all its subparts, on the other hand, the size of the bar in the Hotspot flame graph reflects the percentage of time, and cycle costs for each part. For example, if the main program runs has a lot more cycles than the parallelization library in use, it means that we have a good parallelism, on the other hand, if we have the parallelization library taking a lot more cycles than the main program itself, it means that we have bad parallelism. Fig. 2 demonstrates the frame graph for our ZombieSim program runtime.

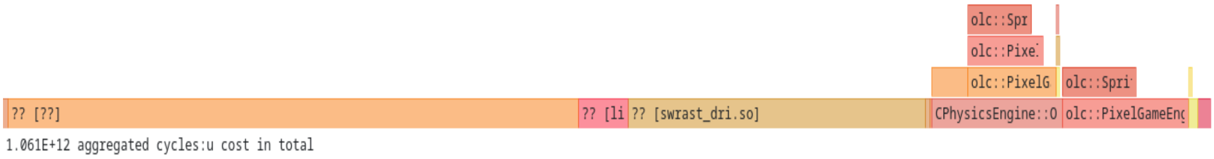

Figure 2: ZombieSim Hotspot flame graph

What we can observe here in Fig. 2 is that the flame graph demonstrates what it seems like a horizontally stacked bars, starting from left hand side to right hand side, the first and second bars correspond to our ZombieSim main simulation, so it is a good sign that it has the largest bar, meaning most of the cycles are spent on the program itself, and not much overhead is added by the usage of the parallelization library, the third bar corresponds to the visualization game engine, which we can't decide on what should be the norm for its percentage, but considering we have most of the cycles used on ZombieSim and we still have more cycles cost used on the game engine physics, it looks like we have a good result; The rest of the bars on the right hand side that their names start with ether PhysicsEngine or olc are both corresponding to the game engine in use for the visualization. This means that the overhead for adjusting the program for parallelization is minimal, which is the desired result.

To make things easier to read, Hotspot also provides tabled results with the corresponding percentages of the cycles used for the program runtime. Fig. 3 demonstrates the tabled results for our ZombieSim that corresponds to what we saw earlier in Fig. 2.

Binary	cycles:u (incl.)	cycles:u (self)
	47.1%	47.1%
swrast_dri.so	24.5%	24.5%
ZombieSim64_Arch.out	10.8%	0.223%
ZombieSim64_Arch.out	10.5%	4.36%
libc.so.6	4.13%	4.13%
libc.so.6	1.02%	1.02%
ZombieSim64_Arch.out	0.762%	0.499%
	0.521%	0.521%
ZombieSim64_Arch.out	0.419%	0.382%
ZombieSim64_Arch.out	0.0951%	0.0644%
libc.so.6	0.0593%	0.0593%
ZombieSim64_Arch.out	0.0295%	0.0295%
libc.so.6	0.0127%	0.0127%
ZombieSim64_Arch.out	0.0117%	0.0067%
libglapi.so.0.0.0	0.0105%	0.0105%
libc.so.6	0.00816%	0.00816%
libGLX.so.0.0.0	0.0063%	0.0063%
libX11.so.6.4.0	0.00574%	0.00574%
libGLX_mesa.so.0.0.0	0.00518%	0.00518%
libgcc_s.so.1	0.00428%	0.00428%
libxcb.so.1.1.0	0.00383%	0.00383%
libc.so.6	0.00375%	0.00375%

Figure 3: ZombieSim Hotspot tabled graph

We observe that ZombieSim useage tops the cycles with 47.1%, followed by "swrast_dri.so" with 24.5%, which is the graphical library in the game engine, then more into ZombieSim outputs, which means that ZombieSim has the most cycles cost, which demonstrates good parallelism.

5 Challenges

Many challenges were faced in our project, which in some cases were tiring, but on the other hand it made it a valuable learning experience, in this section we will split those challenges in regards of their corresponding area.

5.1 Compilation.

Unfortunately, we had issues compiling ZombieSim on the designated HPC system; This was due to the fact that the HPC system uses Scientific Linux which is based on Red Hat Enterprise Linux or RHEL in short. For ZombieSim to compile C++ version 17 (C++17) is needed, because the visual layer (OlcPixelGameEngine) requires it. C++17 can be used on Windows, Debian GNU/Linux and Arch Like systems, but not on RHEL systems. Any other version (GNU12-14 or C++11-C++14) will result in library errors, especially within the Boost library that is used throughout the whole program. Which means that using any version of C++ besides the C++17 version isn't compatible within ZombieSim to the best of our knowledge.

5.2 Performance analysis.

We initially intended to use VAMPIR[Vam] for the performance analysis, but due to the problems faced during the compilation and that we had to compile our program on different systems other than the designated HPC system, we tried to utilize VAMPIR on different systems but that was not possible due to the fact that VAMPIR requires a license, our goal was to create VAMPIR tracefiles that can be reused later on, but it was not possible. On the other hand, we also tried to resort to LIKWID[Lik], installation was successful and easy, had very good documentation a lot of functionality, many of the basic functions ran good, especially LIKWIDs hardware analysis functions, which was nice, but then we were faced with the fact that both our systems were unsupported and incompatible for using the function "likwid-perfctr", which according to the documentation has the type of performance analysis that we need. Moreover, it was strange that our systems CPUs were listed as supported, but after the failure of usage, and making our own research, seems like a lot of specific models were unsupported and were reported in LIKWID forums, but no solution till the time of report. At the end we decided to use Hotspot for the analysis, it does a decent job, but unfortunately it also has its own problems, such as that it does not recognize the name of our program nor the different processes within the program, but the program as a whole, but this still serve the intended purpose, but with limited insights in comparison to what we have hoped for, and to an extent of limited, not very detailed performance analysis.

6 Conclusion

In our course of study in the parallel computing project we learned a lot more about the importance of parallel execution importance and how it can exponentially increase the performance of the programs that we may implement or use, but on the other hand, we had a lot of challenges that were an eye opener that we may have had underestimated in some cases and had to work a lot more on in order to overcome them, but it was what it made it an authentic learning experience that benefited us in many ways, on the other hand we are happy about the state of our program and the simplicity of usage and not only that it is easy to modify but also that it is visualized to give anyone that uses it a real feel of what is going on under the hood in the program, so it makes it much more easier to grasp and be more interactive.

References

[Hot] Hotspot. *https://github.com/KDAB/hotspot*.

[Jia+18] Zihan Jiang et al. “HPC AI500: a benchmark suite for HPC AI systems”. In: *International Symposium on Benchmarking, Measuring and Optimization*. Springer. 2018, pp. 10–22.

[Lik] Likwid. *https://github.com/RRZE-HPC/likwid*.

[OLC] OLC. *https://github.com/OneLoneCoder/olcPixelGameEngine*.

[Vam] Vampir. *https://vampir.eu/*.

A Work sharing

In this section we list our corresponding contributions and work sharing. Both team members contributed in all the phases and output of the project, but with each member being responsible over a different part with the inclusion of major input of the other team member, alongside having weekly meetings at the minimum.

A.1 Abdullah Amawi

Mainly responsible for the report and its structure, with Maaike having her input in the main sections of implementation, analysis, and code related segments.

A.2 Maaike Bierenbroodspot

Mainly responsible for the program code with Abdullah helping in parts of it but with the guidance of Maaike due to her vast experiance in C++ and Boost library.

B Code & other material

We provide the full code for ZombieSim alongside an executable version with its corresponding configuration file.

https://github.com/amawi/ZombieSim.git

On the other hand, an explanation of each file in the code of our ZombieSim project corresponds to:

BB_PropertyBag: A header file that creates the Propertybag, stores all variables that's read from a propertybag

BB_General: A header file containing standard functions used within the rest of the BB files and sometimes the ZombieSim files.

BB_LinuxDefs: By default programs are normally developed with a Windows based by Maaike. She created function definitions to match the C++ version of Windows within Linux.

BB_DirList: Basically get a list of files with extension <xx> from directory defined by the user. Is used in PropertyBag to get the configuration file.

City: All the class relevant function and variables related to the City Objects.

ClearVector: A template to clear out vectors and destroy (delete) it's objects.

Country: All the class relevant function and variables related to the Country Object.

CreateObjects: Basically contains the functions to create the different playfield objects gets called by the main

Helper: Just includes standard macros and also the CGlobal class which contains the propertybag and the management objects. Defines global object variable CGlobal glb.

Humans: All the class relevant function and variables related to the Human Objects.

Initialize: Contains the ParseCommandLine function and the InitProgramVariables. That initialize all the variables used within ZombieSim

Management: Basically a class that handles the error messaging.

olcPixelGameEngine: The Visual layer lib used within ZombieSim. Made by OneLoneCoder.com

PhysicalEngine: Class that contains what the program actually has to do in the visual layer. Uses olc::PixelGameEngine as a derived (polymorphism) class for the actual drawing of the playfield

RunThreads: All the different thread functions used within ZombieSim

Stdafx: Contains all standard libs used throughout the program. In windows this is of course a precompiled header.

Thread: Contains the different thread classes used within ZombieSim

ZombieSim.conf: The configuration file of ZombieSim

ZombieSim.cpp: The main of ZombieSim

ZS_PropertyBag: Uses CPropertyBag (BB_PropertyBag) as a derived class for storing variables from the command line or configuration file. But in itself has getters and setters towards these specific variables. BB_Propertybag stores everything the user adds into the configuration file. But ZS_PropertyBag makes them accessible throughout the program.