2022

Verslag – Circuit Satisfiability Problem

CEYHUN ÇAKIR | 1784480

Inhoud

Inleiding

In deze opdracht moest ik met behulp van openMPI de programma Circuit Satisfiability sneller maken door gebruik te maken van meerdere processen. Dit zou betekenen dat de programma uiteindelijk geparallelliseerd word. We kunnen denken aan bijvoorbeeld dat een bepaald stukje lists gereduceerd worden voor de specifieke proces wat uiteindelijk weer tot een global count terug gezet wordt. Na dat alles geparallelliseerd is moest ik de uitslagen opslaan in een csv bestand wat later weer getoond moet worden in een lineplot.

Wat is openMPI

OpenMPI is een open source message passing interface wat je kan gebruiken om programma's sneller te kunne draaien door meerdere processen op een bepaalde functie te zetten.

MPI_INIT

Mpi init is een functie om initialisatie te maken van de MPI execution environment. De twee arguments die we er in moeten zetten zijn (argc, argv). Argc is een pointer naar de hoeveelheid van argumenten. Argv is de pointer voor de vector argument. Ook is het zo dat alleen de main thread de functie MPI init mag oproepen.

MPI_Comm_rank

Mpi comm rank bepaalt de rank van de calling process binnen de communicator. Dit functie mag door meerdere threads opgeroepen worden wat het dus thread save maakt.

MPI_Comm_size

Mpi comm size bepaald de groep groote die geassocieerd is met de communicator. Ook wel de size die als argument meegegeven wordt is de hoeveelheid nummer van processen in de groep van communicatie. Ook dit functie mag door meerdere threads opgeroepen worden wat het dus thread save maakt.

MPI_Reduce

Mpi reduce zorgt er voor dat de waardes die de mpi processen sturen tot een waarde komen. En dit totale waarde wordt weer opgeslagen in een grote global count. Dit kunnen we bijvoorbeeld op de root zetten. In dit functie worden meerdere argumenten gegeven zoals (send_buffer, receive_buffer, count, datatype, operation, root, communicator). Uiteindelijk geeft de return van het functie een mpi succes en dat wilt zeggen dat de routine succesvol is afgehandeld.

MPI_Finalize

Mpi finalize zorgt er voor dat de mpi environment getermineerd wordt. Het is zo dat alle process dit routine moeten oproepen voordat ze getermineerd worden. Dit functie word ook alleen maar door de zelfde thread als die van de mpi init opgeroepen.

Hoe heb ik het gerealiseerd?

Ik ben begonnen met het downloaden van het (.cpp) bestand die meegegeven was van canvas. Na een code analyse heb ik de programma een beetje aangepast met de code om tijd op te slaan in een variabelen. Ik keek na de aanpassing of de programma werkte zoals het op canvas was beschreven. Hieronder in de afbeelding is te zien wat de aanpassing was voordat ik begon met realiseren van het parallellisatie.

```
double startTime = 0.0, totalTime = 0.0;
startTime = MPI_Wtime();
```

```
totalTime = MPI_Wtime() - startTime;
```

Ik ging toen verder met het aanpassen van het programma zodat de programma wel parallel gedraaid kon worden met meerdere processen. Ik begon met het bekijken van de openMPI documentatie waardoor ik al inzicht kreeg van hoe openMPI werkte. Ik zag al dat ik meerdere functies moest gebruiken genaamd (MPI_INIT, MPI_Comm_rank, MPI_Comm_size, MPI_Reduce, en MPI_Finalize.). Door elk van de functie koppen te lezen die in dit document staan zien we ook waarom we deze functies nodig hebben om dit opdracht te kunnen realiseren.

Hieronder in de afbeelding is te zien hoe ik de volgende 3 functies (MPI_init, MPI_Comm_rank, MPI_Comm_size) initialiseer

```
MPI_Init(&argc, &argv);

MPI_Comm_rank(MPI_COMM_WORLD, &id);
MPI_Comm_size(MPI_COMM_WORLD, &numProcesses);
```

We zien hier zoals ik had uitgelegd dat ik voor de mpi init als argumenten argc en argv er door heen passeer. Verder zet de id en de geselecteerde process in de toebehorende functies MPI_Comm_rank, en MPI_Comm_size.

```
int chunkSize = UINT_MAX / numProcesses;
start = id * chunkSize;
stop = start + chunkSize;
```

Verder creëer ik een variabelen om de chunksize te vinden voor de hoeveelheid processen. Ook creëer twee variabele om alle chunks te vinden voor de specifieke process. Dit noem ik op het moment start en stop.

```
for(combination = start; combination < stop; combination++){
      for (i = 0; i < SIZE; i++)
         v[i] = EXTRACT_BIT(combination, i);

      count += checkCircuit(id, v);
}
```

Verder pas ik alleen de for loop aan om voor de specifieke process alle combinaties te krijgen en voor elke combinatie de bits te extracten en het door de checkcircuit functie heen te halen. De verandering is te zien door de onderstaande afbeelding te bekijken.

```
MPI_Reduce(&count, &global_count, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

totalTime = MPI_Wtime() - startTime;

MPI_Finalize();
```

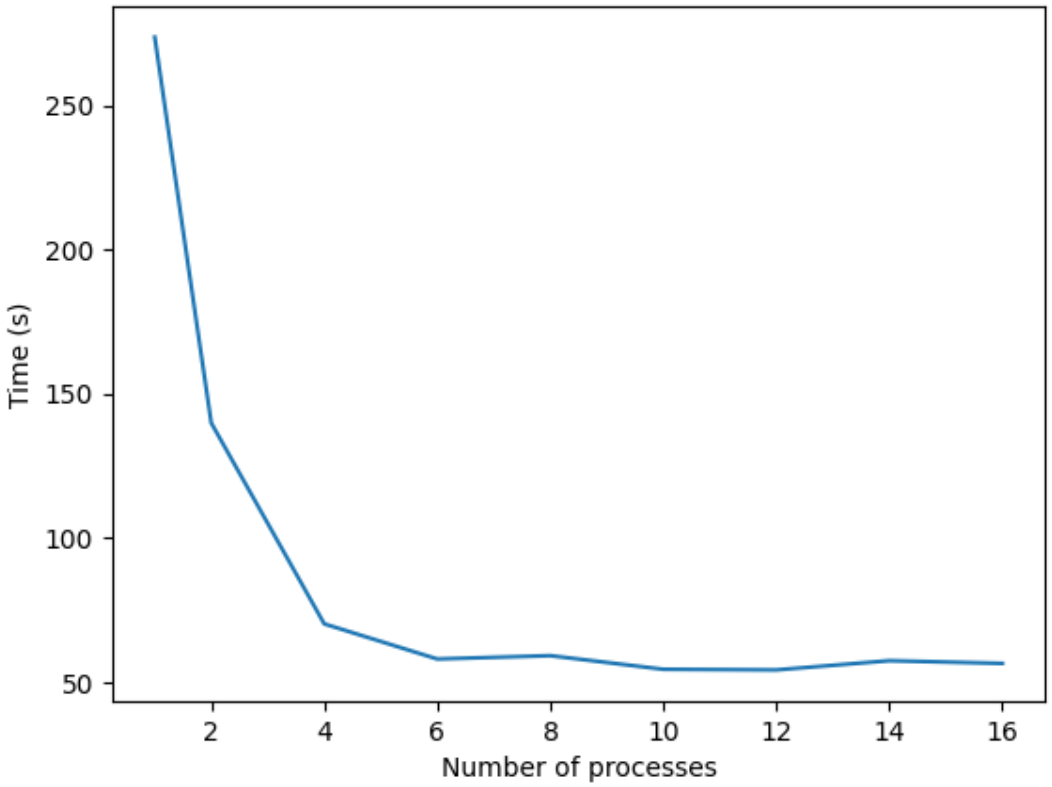
Tot slot gebruiken we de reduce functie om de waarde terug te schrijven naar de global count. We sluiten de mpi process af door de functie mpi_Finalize te gebruiken.

```
if(id == 0) {
  std::ofstream myFile;
  myFile.open("spreadsheet/data.csv", std::ios::app);
  myFile << numProcesses << "; " << totalTime << "; " << global_count << std::endl;
  myFile.close();

 cout << "global_count " << global_count << endl;
}

cout << endl;
cout << "Process " << id << " finished in " << totalTime << " seconds." << endl;

return 0;
```

Om dit data uiteraard te gebruiken kijken we of we op de root process zitten. Wanneer we op de root process zitten schrijven we de data uit naar een Excel sheet die we later kunnen gebruiken om grafieken te tonen waardoor we meer inzicht krijgen in hoe de tijd verschilt door meerdere processen te gebruiken.

Uitkomsten

Van de gegenereerde data die opgeslagen is in de csv bestand kunnen we ook een line plot er van maken. In de onderstaande afbeelding kunnen we zien wat de tijd is met de gegeven processen.



We zien bijvoorbeeld dat wanneer je de programma executeerd en de aantal processes verhoogt dat de tijd afneemt wat blijkt dat het dus sneller klaar is. Dit klopt ook aangezien je met 4 processen sneller klaar bent dan als je met 1 process er mee bezig bent,