4. CIRCUIT SATISFIABILITY

Joey Welvaadt

HOGESCHOOL UTRECHT 1734098

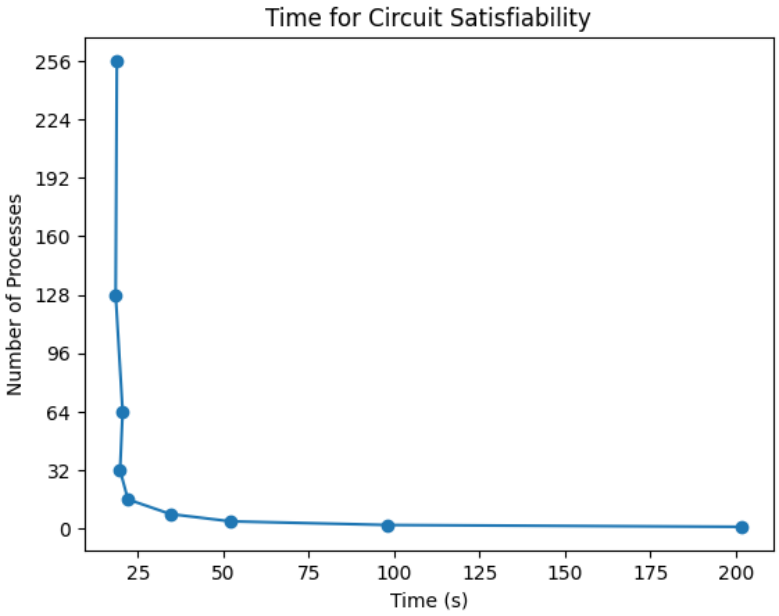
https://github.com/JoeyWelvaadt1999/HPPv2

1. Code

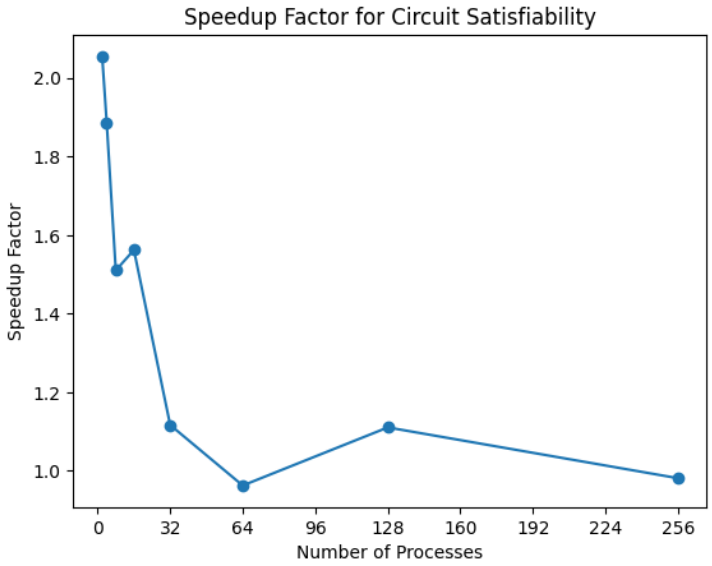
```
int main (int argc, char *argv[])
{

   const uint64_t REPS = UINT_MAX; // number of combinations to check
   uint64_t i, combination; // loop variable (32 bits)
   int id = -1, numProcesses = -1; // process id, number of processes
   uint64_t start = 0, stop = 0; // start and stop of loop
   int count = 0, globalCount = 0;  // number of solutions, global number of solutions

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &id);
   MPI_Comm_size(MPI_COMM_WORLD, &numProcesses);

   uint64_t chunkSize = REPS / numProcesses;
   uint64_t remainder = REPS % numProcesses;
   start = id * chunkSize + (id < remainder ? id : remainder);
   stop = start + chunkSize + (id < remainder ? 1 : 0);



   bool *v = (bool *)malloc(sizeof(bool) * SIZE); /* Each element is one of the 32 bits */
Ctrl+L to chat, Ctrl+K to generate
   cout << endl << "Process " << id << " is checking the circuit..." << " (" << start << " - " << stop << ")" << endl;

   double startTime = MPI_Wtime();

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

      count += checkCircuit(id, v);
   }


   double totalTime = MPI_Wtime() - startTime;
   double maxTime = 0.0;

   MPI_Reduce(&count, &globalCount, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
   MPI_Reduce(&totalTime, &maxTime, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);

   if (id == 0) {
      cout << "\nTotal solutions found: " << globalCount << endl;
      cout << "Max execution time (across all processes): " << maxTime << " seconds\n" << endl;
   }

   free(v);
   MPI_Finalize();
   return 0;
}
```

2. Results

Er worden in dit hoofdstuk 2 verschillende grafieken weergegeven: de tijd die het met elk aantal processen heeft gekost en wat daar de speed up factor voor was (van bijvoorbeeld 1 naar 2 processen).



Figuur 1 Processen vs time



Figuur 2 Speedup factor vs processen

Wat mij is opgevallen bij het verzamelen van deze data is: tot ongeveer 32 processen wordt de tijd dat het kost voor het programma geleidelijk verlaagt, maar van 32 naar 64 processen stijgt het weer een beetje. Het werk wat elk proces moet uitvoeren wordt steeds kleiner, de *relatieve overhead* stijgt. Het komt er op neer dat er een punt is dat het aanmaken van processen en het opdelen van het werk meer tijd kost dan het uiteindelijk uitvoeren van het programma. Zo is ook al te zien bij de

speed up factor, dat van 16 naar 32 die 1.5 is. En van 32 naar 64 1.1, vervolgens van 64 naar 128 gaat deze onder 1 (zie tabel 1).

Aantal Processen	**Tijd (s)**	**Speedup Factor**
1	201.835	0.00
2	98.2361	2.05456
4	52.0982	1.88564
8	34.4968	1.51024
16	22.0829	1.56221
32	19.7724	1.11681
64	20.5361	0.96281
128	18.489	1.11066
256	18.8386	0.98144

Tabel 1 Uitvoertijd en speedup factor bij verschillende aantallen processen