CS 140 - Paralell Scientific Computing

Parallel Sampling and Shortest Path in the Configuration Space of a 3 Link Robot Manipulator on a Distributed Memory Computer System using MPI

by

Sverre Kvamme & Simen Andresen

University of California, Santa Barbara

Abstract

This paper was written as a final project in the course CS 140 Parallel Scientific Computing at the University of California, Santa Barbara in March 2013. The objective of this project was to make a program for parallelizing the procedure of sampling the configuration space of a 3 Link Robot Manipulator.

When controlling robots in arbitrary environment it is necessary to sample the environment and check at which points the robot collides with obstacles, and and then construct a feasible path between points a start and goal point . This procedure is very time consuming for a robot with 3 degrees of freedom.

A solution was therefore to parallelize the procedure of sampling and path generation. This yielded a close to linear speed-up for large sampling spaces.

Contents

1 Introduction

In the field of robotics, motion planning is an important topic with many interesting challenges. One of which are the problem of deciding where a robot can move without violating constraints such as colliding with obstacles in the environment of the robot. This cannot be done analytically in closed form, and sampling of the environment must be done to check where the robot can move in order to comply with these constraints. When it is decided where the robot can move, one can use a shortest path algorithms to decide how to move from a desired start point to a desired goal point. This procedure can be very computational intensive, and using parallel computation would therefore increase the running time, allowing for more samples, and thus higher accuracy and efficiency of the robot motion.

This paper addresses this problem for a specific scenario of a 3 link robot manipulator shown in Fig. 1 where the procedure of sampling, collision detection and shortest path are written in the C programming language using Message Passing Interface (MPI) for parallelizing.

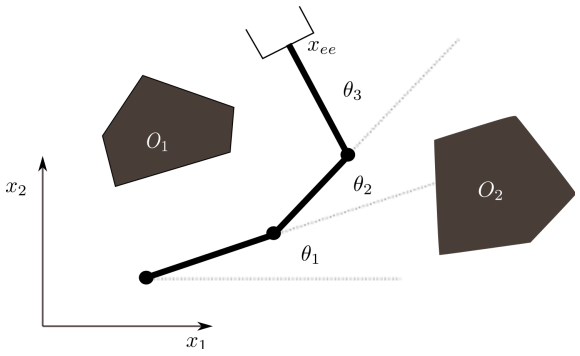

Figure 1: Illustration of a 3 link robot manipulator moving in a two dimensional space

2 Preliminaries and Nomenclature

Further some specifications and notes on notation can be useful.

- The **Workspace W** is the set of all $\mathbf{x} = \begin{bmatrix} x_1 & x_2 \end{bmatrix}^\mathbf{T}$ coordinates in the physical environment of the robot, and is defined as

$$W \in \mathbb{R}^2 \tag{1}$$

- The **Configuration Space C** is the set of all coordinates $\boldsymbol{\theta} = \begin{bmatrix} \theta_1 & \theta_2 & \theta_3 \end{bmatrix}^T$ and belongs to the three dimensional space

$$C \in \mathbb{S}^1 \times \mathbb{S}^1 \times \mathbb{S}^1 = \mathbb{T}^3 \tag{2}$$

  Where $\mathbb{T}^3$ denotes that the configuration space lies on a 3-torus. Since a 3 torus is difficult to visualize, the configuration space will be visualized as a cube as illustrated in Fig. 2. It should further be noted that since $\theta_i \in \mathbb{S} = [-\pi, \pi)$ and $-\pi = \pi$ each side of the cube is the same as it's opposite side.

- The **Free Configuration Space** $\mathbf{C}_f$ is the subset of points in C which does not cause collision between any link of the robot and an obstacle in the workspace.
- The links and obstacles O_1 and O_2 are modelled as convex polygons, where the property of convexity facilitates the collision detection.

Below is a short list of symbols used throughout this text.

nprocs Number of processors

$\mathbf{C}_{f,t}$ Total free configuration space

$\mathbf{C}_{f,i}$ Configuration space on processor i

$\mathbf{n}_{c,t}$ Total size of free configuration space.

$\mathbf{n}_{c,i}$ Size of free configuration space on processor i

$\mathbf{n}_s$ Total number of points in the sample list.

$\mathbf{AdjTab_t}$ Total adjacency table

$\mathbf{AdjTab_i}$ Adjacency table on processor i

$\mathbf{n}_{a,t}$ Total size of adjacency table.

$\mathbf{n}_{a,i}$ Size of adjacency table on processor i

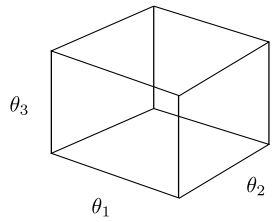

Figure 2: Illustration of the configuration space of the 3-link robot visualized as a cube.

3 Overview of the Program

The program is divided into 7 modules each with it's own purpose. The modules and their respective member functions are illustrated in Fig. 3. Having a look at the program running on one processor, one can divide the program into 4 steps as illustrated in Fig. 4 and the program will be explained in this top-to-bottom order.

4 The Program

In this section each of the different main procedures will be explained.

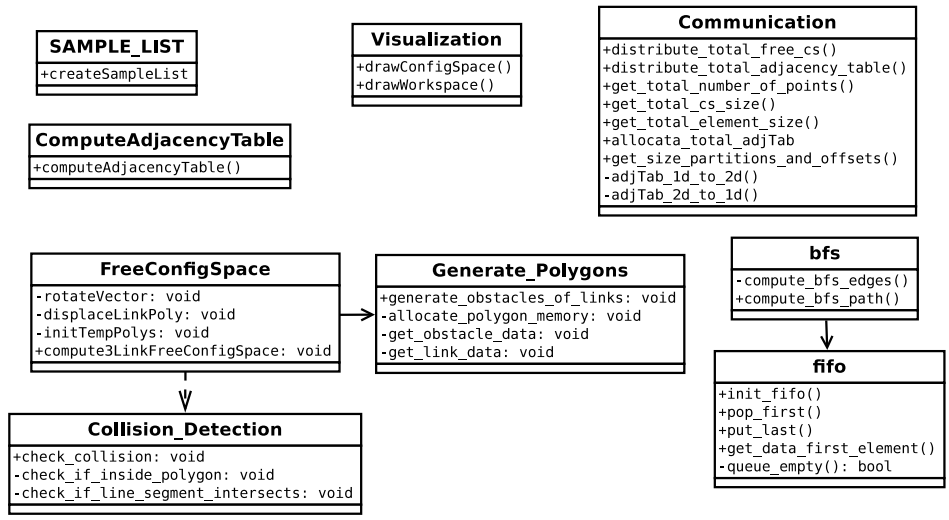

Figure 3: Diagram showing the different modules and dependencies

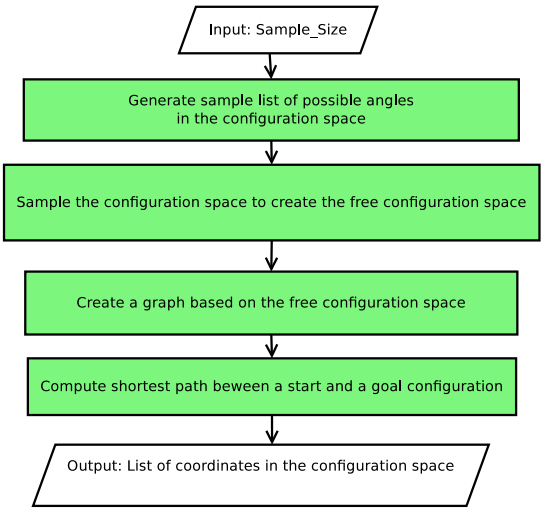

Figure 4: Float chart showing the sequential run of the program

4.1 Create Sample List

In order to check collisions at different configurations a list of configurations must be created. In this project we have chosen to use the Halton sequence which is a pseudo random sequence, giving a deterministic sampling that appear to be random, but with low discrepancy compared to a random sampling, meaning it covers the whole space better.

When running on n processors the data has to be split into nearly equal sized blocks, even when the sample size i not divisible by the number of processors. This is done by the following:

```
sample_size_per_processor = floor( total_sample_size / nprocs );
if(myrank==0){
        sample_size_per_processor +=  total_sample_size %  nprocs;
```

```
}
```

Each processor except from the first one gets an equal sample size, given by the division as given above, while processor 0 gets the same size plus the remainder from the division.

4.2 Compute Free Configuration Space

After each processor has generated a subset of the total configuration space as explained in the previous section, each subset is run through a collision detection procedure that creates a free configuration space. It should be noted that, so far, each processor is working with separate data blocks without any communication. Before a collision detection procedure is run, the workspace is initialized by loading data from a json file describing the polygons making up the links and obstacles, as well as their relative and absolute positions in the workspace.

The collision detection is performed by checking each point in the sample list for collision. The collision detection wont be explained in detail here as it is somewhat outside the scope of this text, but it can be summarized in short:

- The collision detection only works for convex obstacles.
- For a given configuration each vertex in the link polygons are checked if inside or outside any obstacle
- For a given configuration each segment in any link polygon is checked for intersection with any segment in any obstacle

4.3 Distributing Total Configuration Space

To generate a graph based on the total free configuration space $C_{f,t}$ each processor has to have access to $C_{f,t}$. This is done by gathering the data $C_{f,i}, i = \{1, 2..nprocs\}$ to processor 0. Since each partition of the free configuration space is of different size the MPI procedure MPI_Gatherv() is used. In order to use MPI_Gatherv() to get all the data to processor 0, each processor has to send their respective $n_{c,i}$ to processor 0. From this data two arrays are generated:

Offsets An $nprocs \times 1$ array were at index j is the number $\sum_{k=1}^{j-1} n_{c,j}$

Size_per_partition is a $nprocs \times 1$ array where at index i is the number $n_{c,i}$

The two arrays above is then used in the MPI_Gatherv() functon in order to gather the total configuration space $C_{f,t}$ on processor 0, which are then distributed to all the processors.

4.4 Create Adjacency Table

In order to run graph algorithms such as BFS a graph has to be generated based on the free configuration space. This is by checking the distance from each point in in the $C_{f,t}$ to all other points. If the distance between two points is below a certain threshold called $maxAdjRadius$ an edge is created. $MaxAdjRadius$ is constructed as follows:

$$maxAdjRadius = \frac{2.2\pi}{\sqrt[3]{n_s} - 1} \tag{3}$$

In the parallelized version of this, each point in $C_{f,i}$ is checked against the total $C_{f,t}$, and thus creating partitions of the total graph on each processor. It should be noted that the configuration lies on a 3-torus so that the procedure of checking the distance therefore includes some extra arithmetic.

Further, each partition of the adjacency table is distributed gathered on processor 0 and distributed to all other processors in the manner as described in Section 4.3.

4.5 Finding Shortest Path in the Configuration Space

To find the shortest path, a sequential breadth first search (BFS) is applied to the adjacency table. The input to this function is the start and goal node. The BFS algorithm is implemented with a linked list FIFO-queue. Initially the BFS algorithm was planed to be parallelized. However, the time used to find the shortest path is negligible compared to the total runtime of the program and therefore the algorithm remains sequential. Since the edges of the graph is not weighted, the shortest path will consist of the path with the least amount of nodes between start and goal.

5 Where is the Data

To give a short summary of how the data is distributed throughout the run of the program an illustration was made and can be seen in Fig. 5

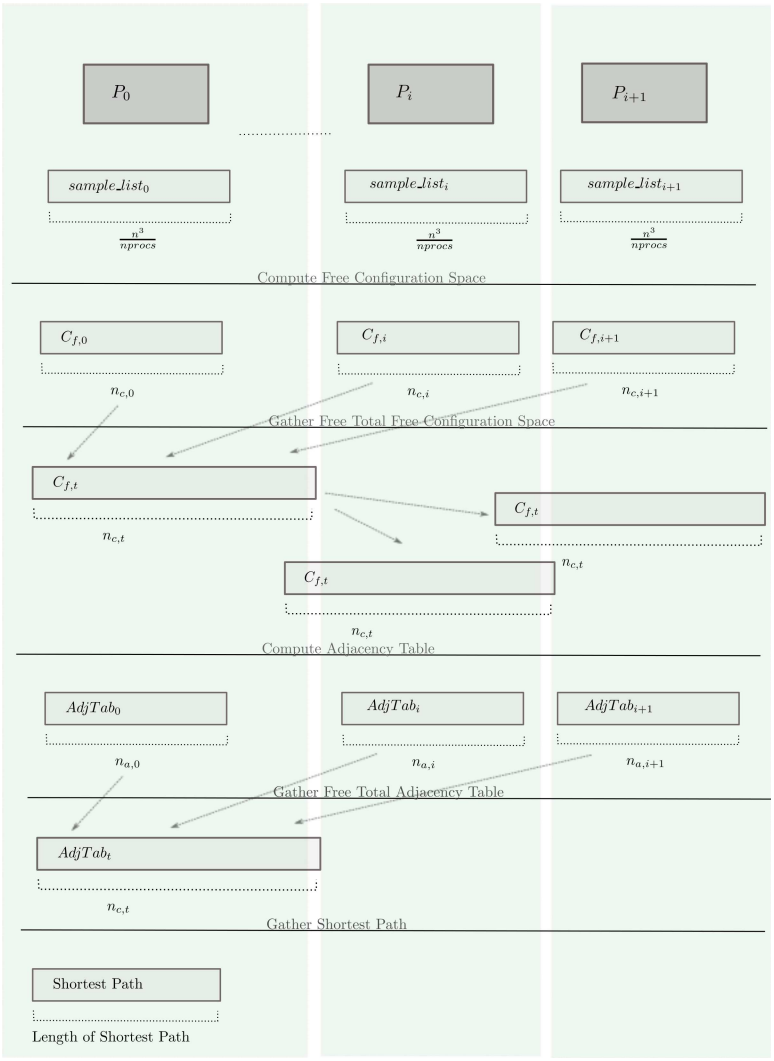

Figure 5: Illustration of how the data is distributed throughout the run of the program

6 Performance

The performance of the program yielded a close to linear speedup for large sampling spaces. This is no surprise as the computation load of the adjency table is rather large and independent, i.e. the problem can be divided into large chunks of work for each processor with minimal communication volume between the processors. Fig. 6 illustrates speedup when timing the whole program. Fig. 7 illustrates the corresponding parallel efficiency. Up to the point where nproc passes 30 the efficiency is above 0.9, which indicates good parallelization. From Table 1 we can also conclude that the main

time-consuming activity is the computation of the adjacency table. When the number of sampling points become significant large, the other modules in the program (sampling and BFS) becomes negligible compared. Even though the computation time of the sampling is much smaller than the computation of the adjency table, the sampling is parallelizated and have a good speedup for nprocs less than 10, as showed in Fig. 8. The tables 2, 3, and 4 shows the runtime with respect to the input n for 1, 8 and 16 processors respectively, and Fig. 9 illustrates the parallel efficiency with respect to n for the whole program.

Table 1: Running times [s] vs nprocs. n=40

nprocs	Sampling	Adjacency	BFS	Totale
1	0.666704	54.267254	0.045985	54.979953
2	0.33628	27.075784	0.045498	27.457603
3	0.22772	18.148102	0.045721	18.421665
4	0.173505	13.570408	0.045746	13.789829
5	0.140272	10.875004	0.045016	11.061219
6	0.122115	9.144176	0.04649	9.313183
7	0.104067	7.876923	0.04649	8.029002
8	0.096887	6.879652	0.04575	7.023167
10	0.079547	5.581816	0.047921	5.709573
12	0.0729	4.634902	0.047505	4.755729
16	0.061942	3.487535	0.046545	3.597782
20	0.058173	2.799534	0.044608	2.90437
26	0.056877	2.16388	0.046477	2.269197
32	0.069046	1.858496	0.046813	1.975175
40	0.064668	1.49332	0.045581	1.605897
50	0.058054	1.191751	0.047265	1.297975
64	0.056982	1.026737	0.044454	1.130737

Table 2: Running times [s] vs n on one processor

n	Sampling	Adjacency	BFS	Totale
15	0.034675	0.153544	0.002176	0.190401
20	0.082417	0.85519	0.005061	0.942674
25	0.161021	3.27004	0.01001	3.441077
30	0.279251	9.732613	0.017938	10.029809
35	0.448	24.459491	0.028328	24.935827
40	0.670897	54.173144	0.044997	54.889046
45	0.952543	110.080956	0.063599	111.097107
50	1.320307	206.835204	0.087269	208.242789
55	1.742765	366.557014	0.118995	368.418783

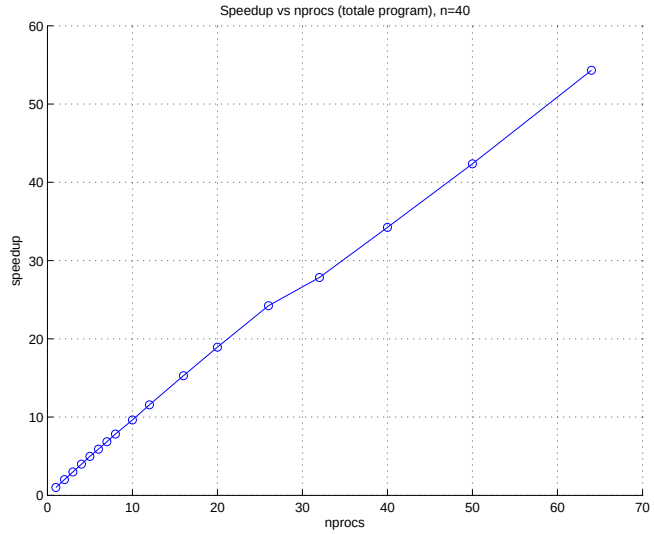

Figure 6: The speedup of the totale runtime with respect to nprocs

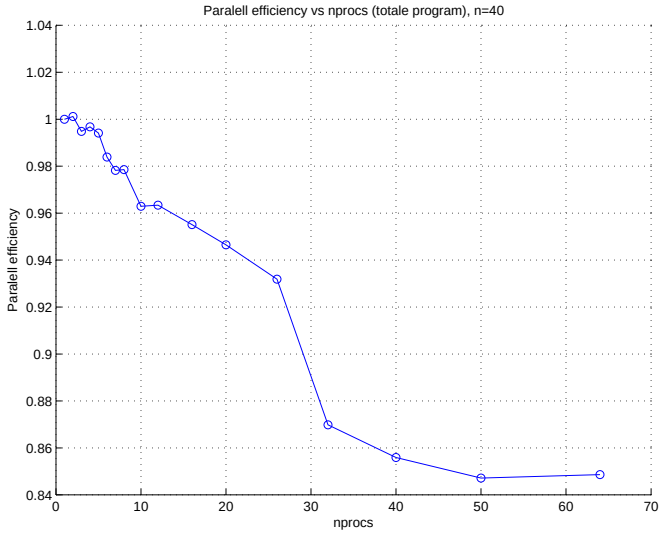

Figure 7: Parallel efficiency of the whole program with respect to nprocs

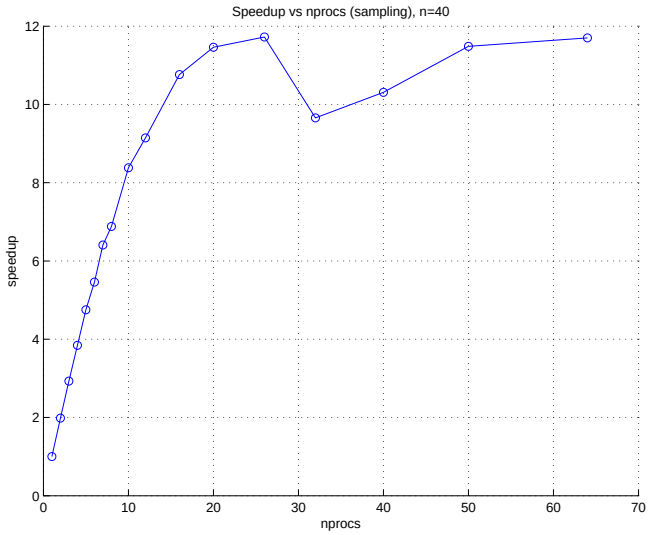

Figure 8: The speedup of the sampling module with respect to nprocs

Table 3: Running times [s] vs n on 8 processor

n	Sampling	Adjacency	BFS	Totale
15	0.008055	0.020849	0.002163	0.03116
20	0.016545	0.115837	0.005195	0.137743
25	0.022748	0.410435	0.00997	0.443438
30	0.04352	1.219338	0.018319	1.281424
35	0.066294	3.085242	0.029837	3.181954
40	0.097265	7.355196	0.046719	7.50003
45	0.133225	14.492618	0.061898	14.691187
50	0.185375	27.796559	0.085963	28.072558
55	0.246335	49.432498	0.136099	49.815521

Table 4: Running times [s] vs n on 16 processor

n	Sampling	Adjacency	BFS	Totale
15	0.008977	0.011992	0.002213	0.023344
20	0.012298	0.058336	0.005641	0.076406
25	0.021309	0.21244	0.010356	0.244682
30	0.027734	0.616534	0.018334	0.663206
35	0.042801	1.549376	0.029344	1.622798
40	0.064563	3.68557	0.046358	3.79863
45	0.079617	7.671292	0.06518	7.819164
50	0.119901	13.876635	0.092847	14.093173
55	0.147009	24.946662	0.122272	25.221188

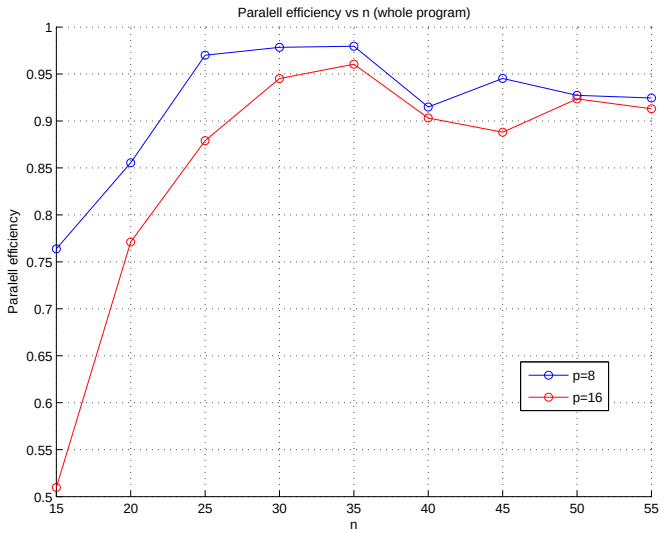

Figure 9: Parallel efficiency of the whole program with respect to n