Mobile Robot Path Planning Using Parallel Computer System

W.Shang and G.K.Egan

Laboratory for Concurrent Computing Systems
Computer Systems Engineering
School of Electrical Engineering
Swinburne Institute of Technology

Abstract: **This paper presents a fast, parallel method for mobile robot path planning. The technique is applicable to autonomous robots operating in an obstacle space populated by other similar co-operating robots, such as robot aircraft or large numbers of robots in an industrial workspace. The technique is based on visibility-Graph algorithm proposed by Lozano-Perez. It has been extended and applied to multiple robots with particular priorities. The parallel algorithm takes advantage of the MIMD shared memory machine. Each robot's path planning can be executed simultaneously because each processor can work independently. The results on Encore-Multimax multiprocessor system show that the computation time of the parallel algorithm is much less than that of the sequential algorithm.**

Introduction: This paper presents a path planning algorithm for multiple robots which has been implemented on the multi-processor system. The basic idea which makes this algorithm possible derives from Lozano-Perez's Visibility-Graph (V-G) algorithm, where the space is represented in terms of configuration space, the obstacles are represented as Cspace obstacles while the robot is shrunk as a point. The free space is visualized as a graph with nodes corresponding to vertexes of Cspace obstacles and links which exist only when two nodes in the graph can "see" each other. Among these nodes and links a path is searched for from a start position to a goal position without collision with any obstacles. We extended this algorithm for multiple robots by giving priority to each robot. The lower priority robot takes into account not only of these real obstacles but also the paths generated by higher priority robots when planning its own path. However, there is a disadvantage. For a single robot, the computation time for Cspace obstacles and searching time are very expensive if there are many obstacles in the workspace, which necessitates too many nodes to be visited. For multiple robots, the searching time increases N times that of a single robot where N stands for the number of robots. In order to solve this problem, we developed a parallel method which can take advantage of the shared memory machine so that the whole computation can be divided into a set of subcomputations and can be done by different processors simultaneously.

The paper is organized as follows: Section I gives background and overview; Section II describes a multiple robots path planning algorithm; Section III covers the parallel implementation on shared-memory machine and its results; and Section IV presents our conclusion and outlines further work to be carried out.

I. Background and Overview

Given a robot with an initial position, a goal position and a set of obstacles located in a workspace, the path planning problem is to find a continuous path for the robot from the initial position to the goal position which avoids collision with obstacles along the way.

A few algorithms have been proposed in this class, amongst which the most influential are the Brooks[2] and Lozano-Perez[3][1] algorithms. Brooks' method represents the free space as overlapping generalised cones and the volume swept by the robot as a function of its orientation. Conceptually, finding a collision-free path is equivalent to comparing the swept volume of the object with the sweepable volume of the free space. In a relatively uncluttered workspace, Brooks' method is fast and efficient. Its major drawback is that paths can only follow the spines of the generalised cones used to represent free space. It does not perform well in cluttered environments, for there are not sufficient generalised cones to allow for a rich choice of path. Lozano-Perez's V-G algorithm represents the space in terms of configuration space or Cspace, as it is often called, which virtually means transforming the robot into a point and enlarging the obstacles accordingly. In practical terms, a collision-free path is one that does not intersect with any of the expanded obstacles. The method consists of three steps:

1. Constructing the Cspace Obstacles;
2. Representing the Free Space; and
3. Searching for a Collision-Free Path.

Constructing the Cspace Obstacles Configuration is used to denote the degree of freedom. The configuration of a polyhedron is a set of parameters that characterise the position of every point of the polyhedron. The space of configuration for a polyhedron A is called its configuration space and it is denoted Cspace A. In Cspace A, the set of configuration of A where A overlaps obstacles B is denoted $CO_A(B)$.

$$CO_A(B) = \{ x \in Cspace\ A \mid (A)_x \cap B \neq 0 \}.$$

$CO_A(B)$ is regarded the Cspace A obstacles due to B. It is an enlarged version of obstacle B, whereas the moving object A can be represented by a point. Lozano-Perez's algorithm: $CO_A(B) = B - Ao$ works well in two-dimensional workspace, where Ao is A in its initial configuration. But in three-dimensional workspace, computation for $CO_A(B)$ is more complicated. Not only the edges of the obstacles but also the faces have to be increased significantly. An "approximate algorithm" has been proposed as an alternative to facilitate the computation of three-dimensional workspace[6]. Although the Cspace obstacles worked out with this method is an approximation and some free space may be ruled out, it works well in computation, for it is dimension-independent and the number of faces does not need to be increased. The enlarged obstacles can be obtained by increasing the length of each vertexes from the centre. See Fig.1.

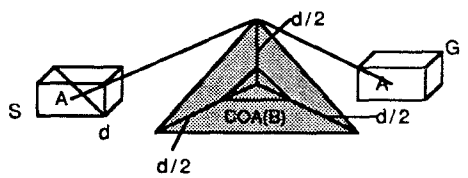


Fig.1 Moving object A and COA(B) by using approximate Method

Representing the Free Space Once the Cspace obstacle $CO_A(B)$ has been constructed, the next step is to represent the free space and obstacles. It is useful to visualise the free space as a graph consisting nodes which correspond to certain states of space, and arcs representing the relationships between states. In the V-G algorithm, a node represents an enlarged obstacle's vertex, an arc exists if any two vertexes can see each other. See fig. 2.

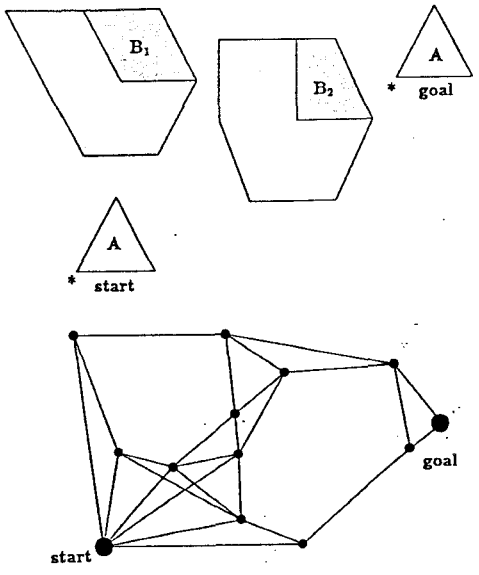


Fig.2 (a) Moving object A and obstacles B1,B2
(b) Free space graph

Searching for a Collision-Free Path Free space having now been represented by way of graph, A* algorithm[7] is used to search the graph for an optimal path, from a node containing a start position to a node containing a goal position. The cost function used is the distance travelled through the space.

II. Path Planning for Multiple Robots

Problem Formulation / Solutions For multiple robots path planning, additional requirements should be considered to avoid collision with other robots in a certain workspace. Not only collision avoidance but also robots coordination need to be taken into consideration simultaneously. The Visibility-Graph algorithm can not guarantee to find collision-free paths for multiple robots. We have developed a new algorithm on the basis of the V-G algorithm to carry out path planning for multiple robots.

In general, the structure of multiple robot path planning in terms of approaches can be classified into (1) centralized configuration and (2) decentralized configuration with prioritization[10]. Our method for multiple robot path planning is to assign a priority to each robot and then plan the path for each robot at a time. The main advantage of this approach is that it reduces the large dimensional problem into a sequence of lower dimensional subproblems. In our case, the priority of each robot depends on the size of each robot.

Our method is based on the concept of configuration space. Each robot has been shrunk into a point while the obstacles have been expanded accordingly. Free spaces are represented as graphs. Each robot has its own graph as they have different sizes. We assume all the real obstacles to be pyramids, each having four vertexes. By using the "approximate" method, we have Cspace obstacles, $CO_A(B)$, as pyramids too. If there are n obstacles in a three-dimensional workspace, there are 4*n + 2 nodes in its graph for each robot. We use S for start node and G for goal node. As the highest priority robot only needs to consider the real obstacles as its obstacles, the path planning algorithm for it is similar to that for a single robot.

Searching algorithm for the highest priority robot:

```
begin
  current = node[s];
  repeat
    count = 0;
    j = 1;
    for i = 1 to 4*n+1 do
      if current and node[i] can see each other then
        begin
          inc(count);
          node_1[j] = node[i];
        end;
    min_node = node_1[1];
    for i = 1 to count do
      if min_node > node_1[i] then
        min_node = node_1[i];
    current = min_node;
  until current = node[g] or all nodes have been visited;
end;
```

For lower priority robots, path planning algorithm is slightly different as we need to consider collision free not only from the real obstacles but also from other robots.

Searching algorithm for lower priority robots:

```
begin
  current = node[s];
  repeat
    count = 0;
    j = 1;
    for i = 1 to 4*n+1 do
      if current and node[i] can see each other then
        begin
          inc(count);
          node_1[j] = node[i];
        end;
    found = false;
    min_node = node_1[1];
    while not found do
      begin
        for i = 1 to count do
          if min_node > node_1[i] then
            min_node = node_1[i];
        if line(min_node, current ) insect any path in
        memory then
          change min_node cost
        else
          found = true;
      end;
    current = min_node;
  until current = node[g] or all nodes have been visited;
end;
```

By tracking back all these current nodes, a path is sure to be obtained. A marked shortcoming is that computation requires considerable time, which means the sum of each robot's computation time, because we cannot start path planning for another robot until that of a higher priority robot is completed. However, the processing can be significantly speeded up with the help of the added power of the parallel computer.

III. Parallel Implementation

Our implementation takes advantage of MIMD (Multiple instruction stream Multiple datastream) shared memory machine with n processors sharing the input/output subsystems and the global memory. See Fig.3. A key feature of shared-memory machine is that the access time to a piece of data is independent of the processor making the request.[8]

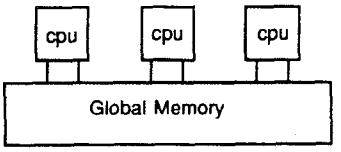


Fig.3 Shared memory machine

As mentioned above, we assume each robot to have a priority. The higher priority robots store their data which can be used by lower priority robots into the global memory, and lower robots have access to it when planning their own paths. The higher priority robots can store their data at any time and these lower priority robots can receive it at any time.

We rely on the programming language to describe how parallelism is to be incorporated. There are two kinds of language support: implicit as well as explicit. Parallelization is implicit when the complier can recognize potentially concurrent portions of a sequential program and generate parallel code. Implicit parallelization requires extensive analyses of the dependencies among data items and cannot guarantee an optimal solution. Parallelization becomes explicit when the programmer must specify the nature and extent of concurrent activities through language constructs. Three mechanisms have been used to support parallel capabilities[9]:

1. incorporating parallel features as integral parts of a language's design;
2. adding parallel extension to an existing sequential language, and
3. providing high-level interfaces to parallel routines stored in a system library.

We chose the second mechanism, language extension, due to the availability of compliers, familiarity of sequential langrage and easing with sequential program. The extensions used in our case are: *fork / join, barrier, spinlock / spinunlock. Fork/ join* is one of the common ways to divide up the work in a shared-memory machine. In this way, a process spawn subprocesses, *fork*, and wait for them to finish, a *join*. The number of subprocesses varies with applications.

In fork / join program, a restriction for accessing code is needed. A program can contain two major sections. Critical section contains code that gets executed by all processors one at a time. Serial section is code to be executed by only one processor and skipped by all others. It is usually used to initialize global data.

The program as discussed before, has been divided into a set of its subprograms. Each subprogram is allocated in different processes. For each robot, its work can be divided into a certain set of subworks. Critical sections and serial sections are also involved in each robot's subprogram. In some sections, these subworks need to be carried out by different processes simultaneously. In this case, synchronisation is needed. *Barrier or fbarrier* are used, for all processes waiting for the last one to arrive. It influences speedup because it may spend a long time waiting for a process and a deadlock may arise. On the other hand, *Spinlock-spinunlock* is used to protect the serial region, no other processes can execute the code in the protect section. Fig 4. shows our parallel program structure with three robots.

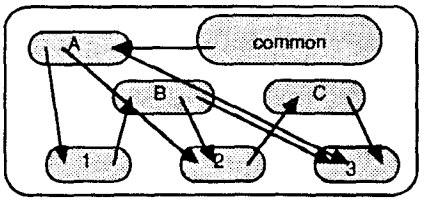


Fig.4 Parallel program structure

common: is a common part which does the program initialization. Global data generated by it is stored in A.

A,B,C: stand for different parts of global memory. The whole data which can be read by every processor is stored in A.B is used for storing data generated by robot[1]'s subprogram, it can be read by 2 and 3. As above, C is used to store data from 2. For 3, both B and C are needed to be considered when planning paths.

1,2,3: stands for each robot's subprogram, which is individual and independent. We assume 1 has the highest priority, 3 the lowest.

Algorithm There are two ways to run parallel programs in *fork / join* style, SPMD and MPMD. With SPMD, each subprocess runs the same program but executes different data depending on its processor id, or data in shared memory. With MPMD, each subprocess runs different programs and executes different data.

SPMD and MPMD are combined in our parallel program. SPMD is preferred for the computation of Cspace obstacles as all processors run the 'expanding procedure' but deal with different obstacles. The program is as follows:

```
repeat
 spinlock( );
     i = share_1;
     share_1 = succ( share_1 );                    program 1
  spinunlock( );
  if i <= no_obstacles then
     expanding obstacle[i];
until i > no_obstacles;
fbarrier( barr );
   .
   .
   .
```

SPMD also helps to check visibility between nodes. The program is as follows:

```
repeat
  spinlock( );
     i = share_1;
     share_1 = succ (share_1);
  spinunlock( );
  if i <= 4*n +1 then                              program 2
     visible[i] = visit( current, node[i]);
until i > 4*n + 1;
fbarrier( barr[i] );
   .
   .
   .
```

The number of subprocesses that we spawn is the multiple of the number of robots. Each robot has the same number of processes as each robot's path planning work is similar. The id number for each robot's subprogram can be obtained from the following:

```
subnprocs = ( number of process ) div (number of robots );

if subnprocs <= 1 then
   id = ( number of robots ) mod ( number of process )
else if subnprocs > 1 then
   (( id - (i-1)*subnprocs ) >= 0)  and (( id - i*subnprocs ) < 0);
```

If we have six processes and three robots,then robot[1] has id = 0 / 1; robot[2] has id = 2 / 3; and robot[3] has id = 4 / 5. Robot[1]'s subprogram is run by subprocesses whose id are 0 and 1. Robot[2]'s subprogram is run by subprocesses whose id are 2 and 3. Robot[3]'s subprogram is run by subprocesses whose id are 4 and 5. I f we have six robots, then robot[1] has id = 0; robot[2] has id = 1; ...; robot[6] has id = 5. If we have 12 robots, then robot[1] and robot[7] use same process whose id = 0; robot[2] and robot[8] use process whose id = 1; etc.

As critical and serial sections are involved in our program,

synchronisation is needed. The mechanism we used is *fbarrier* , all processors wait for the last processor to arrive.

procedure fbarrier_init (var barr:bar; count: integer; var d integer);

procedure fbarrier (var barr:bar);

Count in procedure fbarrier_init indicates how many processes sychronising at the barrier. In the whole program[see program 1], it equals the number of subprocesses that we 'fork'. d can be used for id number. when program is running, it wait until d decrease from (count -1) to 0.

For robot's subprogram[see program 2], count equals each robot's subprocess. As id number need to be available from count_1 - 1 to 0, we use a pseudo-id which is got from

id_ps[i] = id - (i-1)*subnprocs

So, fbarrier_init and fbarrier becomes:

fbarrier_init (barr[i],count,id_ps[i]);

fbarrier(barr[i]);

Robots Coordination As robots are given different priorities, robot[1], who is given the highest priority, starts its path planning first. When robot[2] starts, robot[1]'s data should be taken into account. When robot[3] starts, both robot[1] and robot[2]'s data should be taken into account, etc.. Generally speaking, robot[i] need to consider data from robot[1] to robot[i-1]'s if we assume the priority is from 1 to i. However, the problem is that, as each robot does its own work independently, we cannot guarantee that a higher priority robot stores its data in the global memory before the lower priority robot gains access to it. In order to solve this problem we built up "switches" used to control the order of each robot's work. By simply using *while* statement we reached our goal. Each subprogram[i] has its "switch", which is initialized as zero. The subprogram[i] does not start its program until start[i] becomes 1.

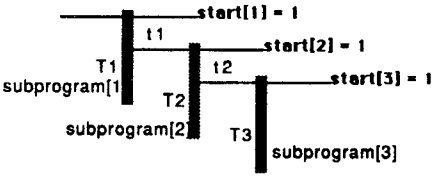
for robot[i]' subprogram:

```
while start[i] <> 1 do
   wait;
subprogram[i];
  begin
     .
     .
     .
  start[i+1] = 1;
     .
     .
     .
end;
```

for robot[i+1]' subprogram:

```
while start[i+1] <> 1 do
   wait;
subprogram[i+1];
  begin
     .
     .
     .
  start[i+2] = 1;
     .
     .
     .
end;
```

It is obvious that subprograms are nested control structure. Higher priority robot's subprogram controls the start of a lower priority robot's subprogram through the "switch" start[i] which is inside higher priority robot's subprogram. They work in this order:



Results The table and graph below show the time and speedup for different number of processor as run for three robots.

speedup = Ts / Tp;

where Ts stands for time required for the nonparallel program execution and Tp for time required for parallel version execution[5].

robots / nprocs	3		6		12	
	time	speedup	time	speedup	time	speedup
1	24.6	1	36.6	1	66.9	1
2	15.3	1.61	21.7	1.69	38.2	1.75
3	10.9	2.20	16.1	2.27	27.0	2.48
6	8.1	3.04	10.3	3.55	17.9	3.74

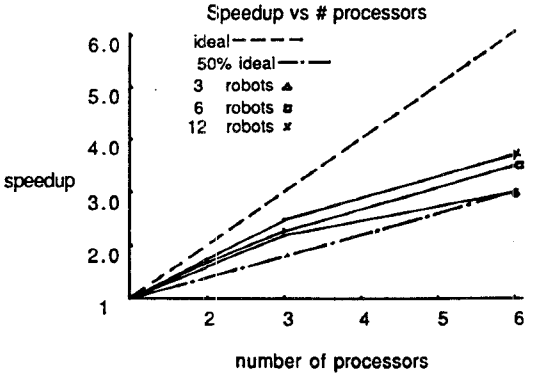


Fig.5 Speedup curves

V. Conclusion and Further Work

A technique for planning the paths for multiple co-operating autonomous robots and its implementation on MIMD shared memory multiprocessor have been described. It can be seen that this is a simple, explicit approach that can guarantee an optimal collision-free path to be found in two-dimensional workspace and a near optimal collision-free path in three-dimensions. The technique can be effectively implemented on the shared-memory multiprocessor and the study has shown that it is possible to obtain a good speedup, especially when an increased number of robots are involved in the computation.

Further work includes using implicit programming language to develop parallel methods and, their implementation on message-passing multiprocessor.

Acknowledgments

We wish to thank all members of the Laboratory for Concurrent Computing Systems at Swinburne Institute of Technology for their support.

Reference

[1] Tomas Lozano-Perez, "Spatial Planning: A Configuration Space Approach", *IEEE transactions on Computing*, Vol. C-32, No.2, pp. 108-119, Feb 1983.

[2] Rodney A. Brooks, "Solving the Find-Path Problem by Good Representation of FreeSpace ", *IEEE Transactions on System, Man and Cybernetics*, Vol.SMC-13, No.3,pp. 190-197, March / April 1983.

[3] Tomas Lozano-Perez, "Automatic Planning of Manipulator Transfer Movements", *IEEE Transactions on System, Man and Cybernetics*, Vol. SMC-11, No.10, pp. 681- 698, Oct 1981.

[4] G.S. Almasi and A.Gottlieb, *Highly parallel computing*, The Benjamin / Cummings Publishing Company, Inc., 1989.

[5] S.Brawer, *Introduction to Parallel Programming*, Academic Press, Inc., 1989. pp. 75.

[6] K.S.Fu, R.C.Gouzalez and C.S.Lee, *Robotics, Control, Sensing, Vision and Intelligence*, New York, McGraw-Hill Book Company, 1987, pp. 400-450.

[7] Nils. J. Nillsson, *Problem-Solving Methods in Artificial Intelligence*, New York, McGraw-Hill Book Company, 1971, pp. 43-79.

[8] Alen H. Karp, "Programming for Parallelism", *IEEE computer*, pp. 43-57, May 1987.

[9] Cherri Pancake, Donna Dergmark, "Do Parallel Languages Respond to the Needs of Scientific Programmers?", *IEEE Computer*, pp. 13-23, Dec 1990.

[10] Ching-Long Shih, J Peter.Sadler, William A.Cruver, "Collision Avoidance for Two SCARA Robots", *Proceedings of the 1991 IEEE International Conference on Robotics and Automation*, Sacramento, California, April, 1991, pp. 674-679.

[11] Hans Sima and Barbara Chapman, *Supercompilers for Parallel and Vector Computers*, ACM press, 1990.