2007 IEEE International Conference on
Robotics and Automation
Roma, Italy, 10-14 April 2007

Motion Planning of Multiple Agents in Virtual Environments on Parallel Architectures

Yi Li and Kamal Gupta
Robotic Algorithms & Motion Planning (RAMP) Lab
School of Engineering Science
Simon Fraser University
Burnaby, BC, V5A 1S6, Canada
Email: {liyi|kamal}@cs.sfu.ca

Abstract— **We proposed in a previous paper [1] a hybrid two-layered approach for motion planning of multiple agents in static virtual environments, consisting of open spaces connected by multiple narrow passages. The discrete Generalized Voronoi Diagram (GVD) of the environment is used to identify narrow passages, and plan the global path of each agent independently of other agents' global paths. As each agent moves along its global path, the agent's path is locally modified using the hybrid technique of combining steering behaviors with Coordination Graphs (CG), where coordination graphs are used for deadlock avoidance in the narrow passages. The planner in the previous paper [1] was single threaded, and it was able to plan the motions of 30 agents moving around in a simple virtual environment with 3 narrow passages. If more agents are moving in a more complex virtual environment (i.e., with more narrow passages), we may not be able to construct and process all the coordination graphs in real-time. In this paper, we parallelize the single threaded planner in a supervisor-worker paradigm with Unix processes who communicate with each other using System V Interprocess Communication (IPC) mechanism. We show that significant, scalable speedups are obtained by constructing and processing coordination graphs in parallel on a Symmetric Multiprocessing (SMP) system.**

I. Introduction

We proposed in [1] a hybrid two-layered approach for motion planning of multiple agents in static virtual environments, consisting of open spaces connected by multiple narrow passages. The hybrid two-layered approach was designed to achieve two goals: very fast pre-processing plus real-time motion coordination and obstacle avoidance. A fast pre-processing is essential for a planner that plans motions of multiple agents in virtual environments and games, because a gamer may only tolerate a few seconds of delay while the game is being loaded/started. Real-time motion coordination and obstacle avoidance are critical in computer games, because the games must be responsive to user's inputs.

During the pre-processing phase, the hybrid two-layer approach uses the discrete generalized Voronoi diagram (generalized in the sense that it is applicable to general sites such as polygons) of the static environment to identify all narrow passages and compute the global path of each agent independently of other agents' global paths. Agents' motions are coordinated at runtime and locally (e.g., when two agents want to to enter the same narrow passage), using a hybrid technique combining steering behaviors [2], [3] with coordination graphs [4]. Using the hybrid two-layered approach, the single threaded planner in [1] planned motions for 30 agents in a virtual environment with 3 narrow passages, the average frame rate was 117 fps and the percentage of all frames updated at a lower speed than 24 fps was 4.27%.

Suppose that the virtual environment has *n* narrow passage. To avoid deadlocks in all narrow passages at a certain time, we must perform *n tasks*, where each task constitutes constructing a coordination graph for a particular narrow passage and then performing a variable elimination algorithm [4] to compute an optimal joint action that avoids deadlock in that particular narrow passage. If there are more than 30 agents and more than 3 narrow passages, the single threaded planner in [1] may not be able to perform all tasks in real-time on a single processor. Because tasks associated with different narrow passages are independent of one another, they can be performed in parallel on Symmetric Multiprocessing (SMP) systems. An SMP system has two or more identical processors connected to a single shared main memory. We consider SMP systems because, first of all, the single shared main memory allows the processors exchanging/sharing data efficiently. Moreover, our work is motivated by computer game applications and all next-generation game consoles (i.e., Microsoft Xbox 360, Nintendo Wii, and Sony PlayStation 3) are SMP systems containing multi-core processers. Although multi-core x86 processors are either dual-core or quad-core, processor manufacturers are adding more cores to their multi-core processors. Sun's UltraSPARC T1 has already eight cores, and Sun plans to release a processor with sixteen cores in 2008 [5].

Some early parallel motion planning approaches (graph-based, grid-based, potential fields, mathematical programming, and ancillary algorithms) were reviewed in [6]. Parallel formulations for exhaustive enumerative search in high dimensional spaces based on a static load-balancing scheme and a dynamic load-balancing scheme were presented in [7] and [8], respectively. Recently, parallelization of probabilistic path planners, such as *Probabilistic Roadmap* (PRM) [9] and *Rapidly-exploring Random Trees* (RRT) [10], has been studied extensively. All parallel versions [11], [12], [13], [14], [15] of these two planners aim to solve high-dimensional problems and/or yield speedups by distributing work to multiple processors. However, none of these planners have real-time constraints. In this paper, we have to not only coordinate the motions of the agents online by computing their optimal joint actions in real-time, but also render the virtual environment and all agents in the environment using OpenGL at a frame rate of at least 24 fps. These real-time requirements impose strong constraints on parallelization, in particular parallel overhead must be minimal.

The two main parallel programming models are: *OpenMP* and *Message Passing Interface* (MPI). With OpenMP, the sequential code can be parallelized easily. However, OpenMP supports only loop-level parallelism, not task parallelism. As other OpenGL programs, the single threaded planner in [1] runs in an event loop accepting and handling events (e.g., display event). The planner runs also in variable frame rate mode (i.e., display events occur as fast as possible). If we were to use OpenMP to parallelize the single threaded planner, then multiple threads must be created and destroyed when a display event occurs. The resulting parallel overhead will be significant enough so that the parallelized version will not yield desired speed-up. MPI supports task parallelism on either SMP systems or distributed systems, but it requires more programming changes to go from sequential to parallel. Additionally, as a purely practical implementational issue, we found it difficult to integrate MPI with OpenGL. Instead, we implemented the task parallelism in a *supervisor-worker paradigm* with Unix processes that communicate with each other using System V Interprocess Communication (IPC) mechanism. The main advantage of this approach is that it minimizes the parallel overhead, because all processes are created only once and destroyed only once and they can communicate with each other very efficiently through the System V IPC mechanism. We show that significant, scalable speedups are obtained by constructing and processing coordination graphs in parallel on a Symmetric Multiprocessing (SMP) system.

This paper is organized as follows. We formally define the problem to be solved in section II. The hybrid two-layered approach is presented in section III. A parallel procedure for construction of multiple coordination graphs and computation of multiple optimal joint actions is presented in section IV. A simple scheduler and asynchronous message delivery are used in section IV to improve the parallel performance on multiple processors. In section V, we describe the experiments, and the results obtained. We conclude in section VI.

II. Problem Definition

Let k agents $A_1, \ldots, A_k$ be in a static two-dimensional polygonal virtual environment, where each agent is modeled by a disc of radius r with two degrees of freedom x and y. The agents' start positions $\mathbf{P}_s = \{P_{s_1}, \ldots, P_{s_k}\}$, and goal positions $\mathbf{P}_g = \{P_{g_1}, \ldots, P_{g_k}\}$ are also defined. The virtual environment consists of open spaces connected by n narrow passages, and for now we assume that no narrow passages intersect with each other. The task is to move each agent from its start position to its goal position without colliding with other agents and static polygonal obstacles.

III. The Hybrid Two-Layered Approach

First, we compute the discrete generalized Voronoi diagram using the frame buffer of the graphics hardware [16]. The distance from any point on the GVD to its nearest obstacle (called *clearance*) can be obtained in the Z-buffer. With the GVD and the distance information, we not only identify all narrow passages in the environments, but also find all agents' global paths.

A. Narrow Passage Identification

All narrow passages, openings, and expanded openings shown in Fig. 1 have been identified automatically using the GVD. Openings are drawn in darker gray (two different greens in colored version) compared to the expanded openings (yellow and orange in colored version) surrounding them.

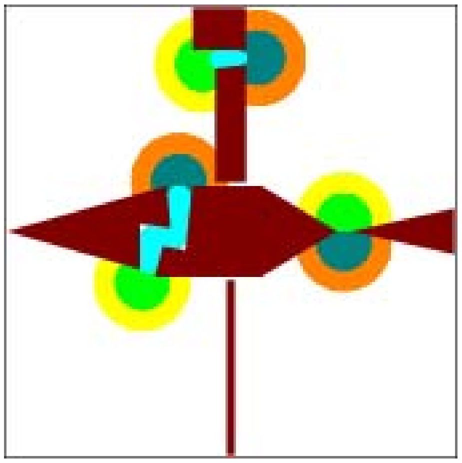

Fig. 1. A virtual environment with 3 narrow passages.

B. Motion Coordination with Coordination Graph

Because the agents are moving in a 2D virtual environment, the medial axis of the free space is a one dimensional entity; hence, the GVD can be used as a roadmap. Using the GVD, the global paths between $\mathbf{P}_s$ and $\mathbf{P}_g$ are computed sequentially and independently of each other.

Having generated a global path for each agent, we want it to follow its path while avoiding obstacles and other agents. This goal can be accomplished by steering behaviors [3]. All steering behaviors are based on local information; therefore, the agents get easily stuck if they are located in cluttered environments, such as narrow passages. To avoid deadlocks in narrow passages, we presented in [17] a hybrid technique, combining local steering behavior and an efficient AI technique for decision making and planning cooperative multi-agent dynamic systems called a *Coordination Graph* (CG) [4]. We do not discuss details here, but just present some broad essentials.

Suppose that m agents $\mathbf{A} = \{A_1, \ldots, A_m\}$ are located inside a narrow passage and its two openings at time t. These agents must coordinate their actions in order to pass through while avoiding deadlocks. All agents are acting in a space described by a set of discrete state variables, $\mathbf{X} = \{X_1, \ldots, X_n\}$. A state $\mathbf{x} = \{x_1, \ldots, x_n\}$ is an assignment to each state variable X_i. Each agent is assigned a local value function $Q_j(\mathbf{x}, \mathbf{a})$, where $\mathbf{a} = \{a_1, \ldots, a_m\} \in Dom(\mathbf{A})$ is a joint action. A local value function is basically a set of value rules, where each value rule describes a context — an assignment to state variables and actions — and a value increment. Next, a coordination graph is automatically constructed with the information in the local value functions. Each node of the coordination graph represents one agent, and two nodes are connected only if the corresponding agents must coordinate their actions; hence, the coordination graph can capture local coordination requirements between agents and the action space is reduced. Finally, all the agents' states are observed and all value rules which are not consistent with the current states are deleted. The variable elimination algorithm is then used in combination with a message passing scheme [4] to find one joint action that maximizes the total utility function $Q = \sum_j Q_j(\mathbf{x})$ called *optimal joint action*. The value rules in [17] are designed in such a way that that optimal joint actions eliminate deadlocks in narrow passages.

C. Summary

The overall procedure for the hybrid two-layered approach is given below as Procedure 1. The most expensive computation in this procedure is the computation of optimal joint actions (step 8). This computation must be done in real-time regardless of the number coordination graphs. In order to handle more agents and more narrow passages, we must speed up the computation of optimal joint actions by constructing and processing coordination graphs in parallel.

Procedure 1 The Hybrid Two-layered Approach

```
1:  Input: a static virtual environment with polygonal obstacles, agents' start positions $\mathbf{P}_s$ and goal positions $\mathbf{P}_g$
2:  Pre-processing: Compute the GVD.
3:  Pre-processing: Identify all narrow passages.
4:  Pre-processing: Compute the agents' global paths sequentially and independently.
5:  loop
6:      Wait for an OpenGL display event.
7:      Observe each agent's state.
8:      Tasks: Construct coordination graphs (once for each narrow passage). For each coordination graph, compute an optimal joint action.
9:      Compute each agent's local path using steering behaviors and the optimal joint actions.
10:     Render the virtual environment and all agents.
11: end loop
```

IV. Parallel Approach

A. Supervisor-Worker Paradigm

We implemented the task parallelism in a supervisor-worker paradigm: a single supervisor, also called master, asks multiple workers, also called slaves, to perform the tasks (i.e., constructing and processing coordination graphs).

The supervisor-worker paradigm can be implemented with either threads or processes. Threads should be used when the same complex data structures must be processed concurrently, and processes should be used instead for less tightly coupled applications [18]. Therefore, we choose processes for implementation of the supervisor-worker paradigm, because all workers perform their tasks independently of one other; consequently, no data is passed between workers. Although creating processes costs more than creating threads, note that no processes are created or destroyed during runtime. We create and destroy processes only when the supervisor and the workers initialize and terminate, respectively. Another advantage of processes over threads is that developing and testing processes separately is easier. However, each process has its own set of memory pages; hence, two processes need to use Interprocess Communication to communicate with one another. We use System V IPC for interprocess communication, in particular *message queues* and *shared memory segments*.

We divide Procedure 1 into two programs: a supervisor program and a worker program. Once the supervisor program is launched, we make a system call *fork* to create a second process. The original process is called the *parent*, and the new one is called *child*. In order to create multiple worker processes, we launch the worker program multiple times. The number of the worker processes depends on the number of available processors. The supervisor processes communicate with the worker processes through two message queues (Fig. 2): the supervisor sends messages to the workers through message queue A, and the workers send back the results through message queue B.

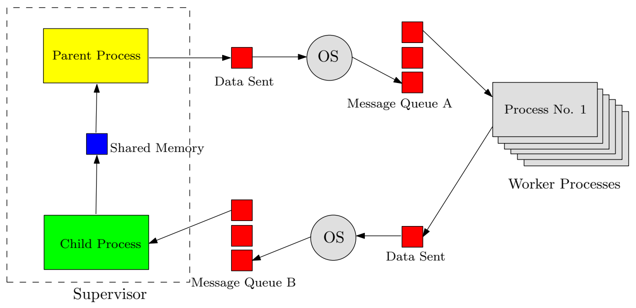

Fig. 2. The supervisor and its workers communicate through System V message queues and shared memory.

Each time a display event occurs (i.e., step 6 in Procedure 1), the parent process observes all agent's states (step 7). For each narrow passage, the parent process stores the relevant agents' states inside a message before the message is appended to message queue A. Once all messages are sent to the worker processes, the parent process continues on executing step 9 and step 10. As soon as a worker process is free, it pops the first message in message queue A. Using the data stored inside the message, the worker process performs one task from step 8, and sends back the corresponding optimal joint action to the child process of the supervisor through message queue B. When the child process pops a message in message queue *B*, it writes the data (i.e., an optimal joint action) in the message into the same memory it shares with the parent process (Fig. 2). The shared data are only written by one process (i.e., the child process); hence, there is no need to use semaphores for restricting access to the data.

We use message queues for communication between the supervisor and the workers, because small messages (100 bytes or so) can be passed between two processes quickly [18]. However, the size of a message and the number of messages in the queue are limited [18]. For example, we are able to send 40 messages to message queue *A* on an SGI UltimateVision. If there are more than 40 narrow passages in the virtual environment, shared memory can be used instead of message queue for communication between the supervisor and its workers. Shared memory should also be used for big messages [18].

B. Job Scheduling

n tasks (i.e., construct *n* coordination graphs, and then compute an optimal joint action for each one of them) are performed each time a display event occurs. A task can only be processed by one processor, and a task cannot be interrupted. All processors available for processing are identical. We want to minimize the length of time required to complete all tasks denoted by *M*. This is a classical *scheduling problem of parallel identical processors and independent jobs* (tasks). We use a fast and effective heuristic procedure for minimizing *M*, *Longest Processing Time* (LPT). The schedules produced by LPT are close-to optimal [19].

In general, processing time for a coordination graph grows with the number of nodes in it. Therefore, an LPT ordering of the tasks can be obtained by sorting the coordination graphs according to the number of nodes.

C. Asynchronous Message Delivery

With multiple processes running in parallel, once all messages are sent by the parent process (in the supervisor program) to the worker processes, the parent process continues (to step 9 and step 10 Procedure 1) without waiting for the results from the worker processes via the child process. This is called *asynchronous message delivery*. With it, we can achieve a smoother simulation (i.e., maintain at least a frame rate of 24 fps) than the single threaded program. However, on average, the worker processes still need to compute optimal joint actions in less than one-tenth of a second in order to avoid deadlocks in the narrow passages.

During the simulation, the parent process maintains a table of boolean variables, with each variable corresponding to a single agent. The size of the table is therefore equal to the number of agents. The initial values of all table entries are *true*. Whenever an optimal joint action is received, the parent process checks whether an agent is allowed to enter a narrow passage or not. If not, the agent's corresponding entry in the table is set to *false*. Each time a display event occurs, the parent process checks all entries of the table. If an entry contains value *false*, the corresponding agent makes a U-turn away from the narrow passage in front it; hence, a deadlock inside the narrow passage is prevented. Whenever a narrow passage is empty, all entries corresponding to the agents that intend to enter the narrow passage are reset to *true*.

V. Experiments and Results

A. Hardware and Software Setup

The experiments were performed on two different systems: an SGI UltimateVision and a Dell OptiPlex GX620. The SGI UltimateVision has 24 MIPS R16000 processors and runs IRIX 6.5.28. Each MIPS processor runs at 700 megahertz, and has 4 megabyte L2 cache. The system has 14 gigabytes of global shared memory, and it is visible to all 24 processors. The Dell OptiPlex GX620 has one Intel Pentium D Processor 820 with two execution cores running at 2.8GHz and each core having 1MB level 2 cache (2MB in total), one 256MB ATI Radeon X600 PCIe video card, 4 gigabytes of RAM, and runs Red Hat Enterprise Linux 4. The code was written in ANSI C/C++ and compiled using GNU GCC 3.4.3 on the Dell OptiPlex GX620 and GNU GCC 3.3 on the SGI UltimateVision.

B. Performance Analysis

1) The SGI UltimateVision: An important performance metric is *speedup S*, which is defined as

$$S(N,P) = \frac{T_{seq}(N)}{T(N,P)}, \qquad (1)$$

where N is the size of the problem, P is the number of processors, T_{seq} is the runtime of the sequential program, and $T(P)$ is the runtime of the parallel program.

To measure the parallel efficiency, we created a simplified supervisor that enables us to perform some experiments in a controlled environment using the SGI UltimateVision. The simplified supervisor generates jobs of different sizes, but it does not perform rendering, because the rendering and all agent steering behaviors are handled by a single process in the supervisor (i.e., the parent process), we found that the 700 megahertz MIPS processors in the SGI system are not powerful enough. A job consists of one or multiple tasks. The size of the job varies not only with the number of the tasks, but also with the size of each task, which is equal to the number of nodes of the coordination graph inside the task, and it ranges from 1 to 6 in our experiments. Depending on the experiment, the simplified supervisor generates a job with a certain number of tasks, where the sizes of the tasks may be equal to one another or different. To measure runtime, a timer is started before the parent process in the simplified supervisor sends messages to the workers through message queue *A*; it is stopped once the parent process receives all optimal joint actions from the child process.

First, we measured how the runtime of one task varies with the size of the task. For each job, the simplified supervisor sent 40 tasks of the same size to the workers for processing. The average runtime for 1 task is shown in Table I.

Next, we investigated how the size of a task (i.e., the number of nodes in a coordination graph) affects the speedup.

TABLE I

RUNTIME OF ONE TASK.

Task Size (No. of CG nodes)	1	2	3	4	5	6
Time in msec	2	10	20	64	107	160

Three different numbers of nodes were tested: 2, 4, and 6. For each test, the simplified supervisor sent 40 tasks of equal size to the workers. Runtimes for 1, 2, 5, 10, and 20 workers are shown in Table II and Fig. 3. The speedups for coordination graphs with 4 and 6 nodes are almost linear. However, for coordination graphs with 2 nodes, the speedup is just over 10 when using 20 workers, due to the *communication overhead*. The effect of the communication overhead is especially evident when the jobs are small, and they are distributed to many processors (e.g., when 40 coordination graphs with 2 nodes are processed by 20 workers).

TABLE II

RUNTIMES OF 40 EQUAL-SIZE TASKS.

No. of Processors (*P*)	1	2	5	10	20
Time in msec (2 nodes)	412	210	86	51	33
Time in msec (4 nodes)	2568	1279	521	268	149
Time in msec (6 nodes)	6326	3148	1272	647	338

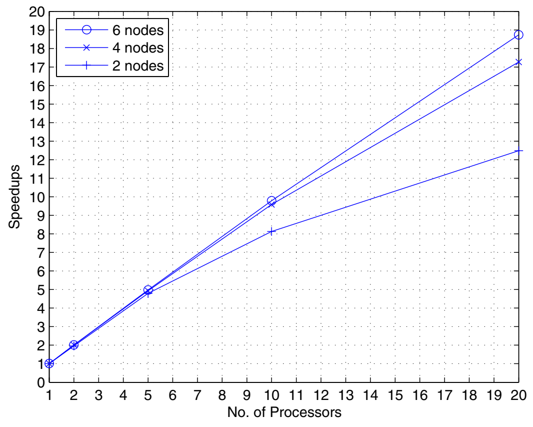

Fig. 3. Speedups for 40 equal-size tasks.

Another factor affecting speedups is the load balance between different processors. Ideally each processor should perform the same amount of work. Given multiple tasks with mixed sizes, we use the LPT scheduler to distribute tasks evenly among the available processors for a good load balance. We tested the effect of the LPT scheduler with 20, 30, and 40 tasks. The size of each task was determined randomly by the simplified supervisor, and it ranged from 1 to 6. Each task was performed first without the LPT scheduler, and then with the scheduler. Runtimes (averaged over 10 runs) are shown in Table III. Speedups are shown in Fig. 4.

The data in Table III and Fig. 4 indicates that the parallel performance improves significantly when the LPT scheduler

TABLE III

RUNTIMES OF UNEQUAL-SIZE TASKS (WITH AND WITHOUT SCHEDULING).

No. of Processors (*P*)	1	2	5	10	20
without scheduling					
Time in msec (20 tasks)	1081	569	270	171	157
Time in msec (30 tasks)	1625	842	378	231	164
Time in msec (40 tasks)	2139	1098	482	290	194
with scheduling					
Time in msec (20 tasks)	1079	525	223	155	150
Time in msec (30 tasks)	1620	812	332	184	153
Time in msec (40 tasks)	2139	1071	434	231	176

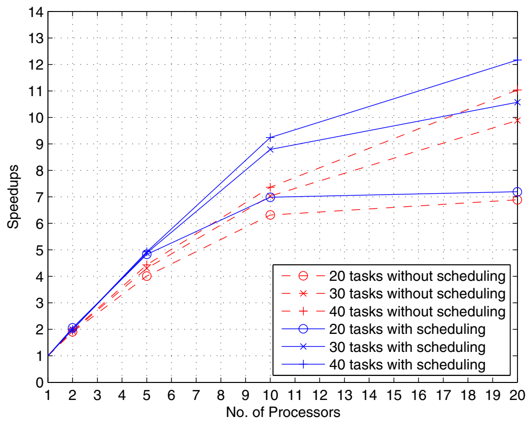

Fig. 4. Speedups for unequal-size tasks (with and without scheduling).

is used. Observe that the speedups are limited by the processing time of the biggest task (i.e., a task with size 6). Because it takes abut 160 msec (Table I) to process a task with size 6, multiple tasks can not be processed in less time than 160 msec regardless of the number of processors used to process these tasks.

2) The Dell OptiPlex GX620: Finally, we ran the supervisor-worker paradigm on the OptiPlex with a faster Pentium D processor (compared to the MIPS processors) using a full version of the supervisor, which not only observes all agents' states and distribute works to the workers, but also renders the agents and the virtual environment using OpenGL.

We used the virtual environment shown in Fig. 5 to test the runtime performance. To make sure that an agent (shown as a solid triangle) passes through at least one narrow passage while moving along its global path, each agent's start position and goal position are located in two different open spaces. Once an agent reaches its goal position, it makes U-turn and heads back to its start position.

In addition to the two supervisor processes (i.e., the parent and the child), we created two worker processes. Because the single Pentium D processor has two cores, the four processes we created are simply multiplexed over the two cores by the system. However, running with fewer processors than processes can hinder performance [20]. In order to maintain a sufficient frame rate, we gave the worker processes lower

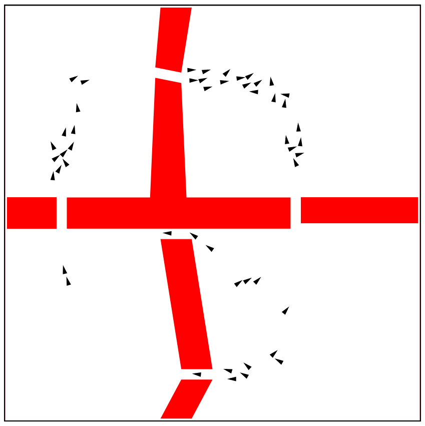

Fig. 5. A virtual environment with 5 narrow passages and 50 agents. See the accompanying short video clip.

priorities than the supervisor processes by setting worker processes' and supervisor processes' *nice values* to 39 and 20, respectively. A process with higher nice value runs at a lower priority [18].

We ran the simulation (see the accompanying short video clip) for 10 minutes. By processing two coordination graphs in parallel utilizing the dual-core processor, we can now plan motions of 50 agents (instead of 30 [1]) in a more complex virtual environment shown in Fig. 5. The average frame rate is 106 fps and the standard deviation is 54. Only 0.95% of all frames were drawn at a frame rate slower than 24 fps.

VI. Conclusions

In this paper, we parallelized a hybrid two-layered approach for motion planning of multiple agents in virtual environments we proposed in [1] on a SMP system. We implemented the task parallelism in a supervisor-worker paradigm with Unix processes who communicate with each other using System V Interprocess Communication mechanism. The supervisor not only renders, but also distributes works to the workers and receives results (i.e. optimal joint actions) from the workers. A worker takes order from the supervisor, then constructs a coordination graph and computes an optimal joint action.

The main advantage of our approach is that the parallel overhead is reduced to a minimum, because all processes are created only once and destroyed only once, and the processes can communicate with each other very efficiently using the System V IPC mechanism. Parallel efficiency of our approach is further improved by job scheduling algorithm LPT and asynchronous message delivery. The experiments show that significant, scalable speedups are obtained. By constructing and processing multiple coordination graphs in parallel on a SMP system, the hybrid two-layered approach can now handle more agents and more complicated virtual environments.

Acknowledgment

The authors would like to thank Brian Corrie at the IRMACS Centre, Simon Fraser University for his help and support. This research has been enabled by the use of WestGrid computing resources, which are funded in part by the Canada Foundation for Innovation, Alberta Innovation and Science, BC Advanced Education, and the participating research institutions. WestGrid equipment is provided by IBM, HP and SGI. This research was supported in part, by a grant from Natural Sciences and Engineering Research Council (NSERC) of Canada.

References

[1] Y. Li and K. Gupta, "A hybrid two-layered approach to real-time motion planning of multiple agents in virtual environments," in *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2006.
[2] C. W. Reynolds, "Flocks, herds and schools: a distributed behavioural model," *Computer Graphics*, vol. 21, no. 4, pp. 25–34, 1987.
[3] ——, "Steering behaviors for autonomous characters," in *Proceedings of Game Developers Conference*, 1999, pp. 763–782.
[4] C. E. Guestrin, "Planning under uncertainty in complex structured environments," 2003. [Online]. Available: http://ai.stanford.edu/ guestrin/Publications/Thesis/thesis.pdf
[5] A. Vance, "Sun cheers two sparc advances in one week (true)," January 18, 2007. [Online]. Available: http://www.theregister.co.uk/2007/01/18/sun_rock_tape/
[6] D. Henrich, "Fast motion planning by parallel processing – a review," *Journal of Intelligent and Robotic Systems: Theory and Applications*, vol. 20, no. 1, pp. 45–69, 1997.
[7] D. Henrich, C. Wurll, and H. Worn, "6 dof path planning in dynamic environments – a parallel online approach," in *Proceedings of IEEE International Conference on Robotics and Automation*, vol. 1, 1998, pp. 330–335.
[8] B. Taati, M. Greenspan, and K. Gupta, "A dynamic load-balancing parallel search for enumerative robot path planning," *Journal of Intelligent and Robotic Systems*, vol. 47, pp. 55–85, 2006.
[9] L. E. Kavraki, P. Svestka, J. C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, 1996.
[10] S. M. LaValle and J. J. Kuffner, "Randomized kinodynamic planning," in *Proceedings of IEEE International Conference on Robotics and Automation*, vol. 1, 1999, pp. 473–479.
[11] M. Akinc, K. E. Bekris, B. Y. Chen, A. M. Ladd, E. Plaku, and L. E. Kavraki, *Probabilistic Roadmaps of Trees for Parallel Computation of Multiple Query Roadmaps*, ser. Robotic Research: The Eleventh International Symposium. Springer, STAR 15, 2005, pp. 80–89.
[12] E. Plaku, K. E. Bekris, B. Y. Chen, A. M. Ladd, and L. E. Kavraki, "Sampling-based roadmap of trees for parallel motion planning," *IEEE Transactions on Robotics*, vol. 21, no. 4, pp. 597–608, 2005.
[13] S. Carpin and E. Pagello, "On parallel RRTs for multi-robot systems," in *Proceedings of the 8th Conference of the Italian Association for Artificial Intelligence*, 2002, pp. 834–841.
[14] N. M. Amato and L. K. Dale, "Probabilistic roadmap methods are embarrassingly parallel," in *Proceedings of IEEE International Conference on Robotics and Automation*, vol. 1, 1999, pp. 688–694.
[15] P. Isto, "A parallel motion planner for systems with many degrees of freedom," in *Proceedings of International Conference on Advanced Robotics*, 2001, pp. 339–344.
[16] K. E. Hoff, T. Culver, J. Keyser, M. Lin, and D. Manocha, "Interactive motion planning using hardware-accelerated computation of generalized voronoi diagrams," in *Proceedings of IEEE International Conference on Robotics and Automation*, vol. 3, 2000, pp. 2931–2937.
[17] Y. Li, K. Gupta, and S. Payandeh, "Motion planning of multiple agents in virtual environments using coordination graphs," in *Proceedings of IEEE International Conference on Robotics and Automation*, 2005, pp. 380–385.
[18] M. J. Rochkind, *Advanced UNIX Programming*. Addison-Wesley Professional, 2004.
[19] K. R. Baker, *Introduction to Sequencing and Scheduling*. John Wiley and Sons Inc, 1974.
[20] R. Chandra, L. Dagnum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel programming in OpenMP*. San Francisco, CA: Morgan Kaufmann Publishers, 2001.