Speeding Up Learning in Real-time Search through Parallel Computing

Vinicius Marques Luiz Chaimowicz Renato Ferreira
Department of Computer Science
Universidade Federal de Minas Gerais
Belo Horizonte, Brazil
{*allonman,chaimo,renato*}*@dcc.ufmg.br*

***Abstract*—Real-time search algorithms solve the problem of path planning, regardless the size and complexity of the maps, and the massive presence of entities in the same environment. In such methods, the learning step aims to avoid local minima and improve the results for future searches, ensuring the convergence to the optimal path when the same planning task is solved repeatedly. However, performing search in a limited area due to real-time constraints makes the run to convergence a lengthy process. In this work, we present a parallelization strategy that aims to reduce the time to convergence, maintaining the real-time properties of the search. The parallelization technique consists on using auxiliary searches without the real-time restrictions present in the main search. In addition, the same learning is shared by all searches. The empirical evaluation shows that even with the additional cost required to coordinate the auxiliary searches, the reduction in time to convergence is significant, showing gains from searches occurring in environments with fewer local minima to larger searches on complex maps, where performance improvement is even better.**

***Keywords*-parallel and distributed systems; algorithms; applications;**

I. Introduction

In the context of video games or robotics, there are some situations that may require fast responses, or even real-time responses. In these situations, the AI engine can be responsible for hundreds to thousands of agents traversing huge and complex maps simultaneously, making the path planning time becomes a major factor. Real-time search methods, like LRTA* [1], solve this problem without compromising time restrictions. These algorithms perform local searches, only planning a few actions per time in a limited area, therefore giving a fast response. Real-time search also has the ability to perform searches on dynamic environments, that can change their structures over time. Such ability is possible because these methods interleave planning and plan execution with a learning step, which refines a cost function (formed by heuristic estimates) that guides the upcoming actions. The learning acquired is used in subsequent searches, converging to a shortest path when solving the same planning task repeatedly [2].

However, the fact that such algorithms amortize learning over several planning episodes makes the convergence a lengthy process. In order to keep the real-time premise and give fast responses, the learning can only be applied to a delimited area. This process to convergence, also named run, is then an expensive process, that often has to be done under real-time constraints. Thus, the question discussed here is how learning can be accelerated so that fewer repeated path-finding experiences are needed before converging to an optimal path.

In this paper we exploit parallelization to speed up learning, and therefore the run to convergence. It is important to notice that the parallelization does not aim to reduce the time to a goal in a single search. The objective of the parallelization is to reduce the time spent to converge to the optimal path, *i.e.*, the amount of searches performed until the minimal path is reached.

We focus on data parallelism, performing the same search in parallel under different regions of the map. Moreover, all the searches share the same learning, which means that they have to use the same heuristic values. The parallelization is defined as a master/slave based protocol, where the main core both coordinates tasks as well as execute them under real-time constraints.

We have designed the parallelization considering a distributed memory system, with data synchronization between the cores. The implementation was made on a current console architecture, the PlayStation 3's Cell Broadband Engine Architecture [3]. The choice of this architecture is justified by the fact that real-time search algorithms have a wide application on the AI present in current games. Although Cell/B.E. has singular details that differs it from other architectures, it is important to notice that this choice does not imply in limitations on the application of the technique introduced in this paper. The parallelization proposed here fits any architecture, including a possible port to general purpose programing on graphical processing units (GPU), like CUDA[1]. In fact, this hypothesis was even considered, but discarded due to possible competition with CGI in video games, where tasks are processing intensive. We also implemented a particular task sorting technique, used to define the priority of the tasks related to the searches performed on the auxiliary cores.

The rest of the paper is divided as follows: Section II gives a brief overview about the research related to our work.

[1]See *The CUDA Programming Guide, 1.1.1*

We describe details of the parallelization methodology in Section III, also showing the task composition, distribution and how to define priority for them. Section IV describes the results of our experiments. Finally, Section V concludes the paper.

II. Related Work

In this section we present a brief overview of some related works that focus on speedup convergence of real-time search. For further analysis of general performance improvements on search algorithms, please refer to Rios & Chaimowicz [4].

Researchers have attempted to speedup the convergence of real-time search methods without increase search depth, the *lookahead*. Early methods aims speedup by sacrificing the optimality of the resulting path [5], [6], allowing suboptimal solutions, with a margin of error, to reduce the total amount of learning performed.

FALCONS (Fast Learning and Converging Search) [7] shows that tie-breaking criterion crucially influences the convergence speed, and then uses an alternatively way to select successors: instead of minimizing the estimated cost to go, it considers the estimated cost from the start (that can be outside the current local search space) to goal, via the successor state it moves to.

LRTA*(k) [8] is an LRTA* based algorithm with an alternative strategy to the propagation of changes of heuristic estimates, named bounded propagation. This strategy propagates these changes on the successors of the current state, limited by a parameter k. The LRTA*(k) records heuristic estimates that are closer to their exact values than those recorded by LRTA*, therefore converging faster.

Bulitko et al. [9] combine automatic state abstraction with learning real-time search, dynamically building state abstractions that allows the learning to generalize updates to the heuristic function. This allows more states to have the heuristic values updated, thereby speeding up learning.

The Local Search Space LRTA* (LSS-LRTA*) [10] considers the learning space as the same as local search space, determined by a bounded A* search. It also uses a different way to update heuristics by performing a Dijkstra search. A larger local search space guarantees more heuristic updates while planning. In addition, the execution of a Dijkstra search over the learning space optimizes the range of learning, speeding up the run to convergence.

All those works cited above have concentrated the efforts on modifying the LRTA* original structure to reduce the convergence time. The resulting methods do not modify the strictly sequential behavior of the original method. Our work differs form the others by introducing the paradigm of parallel programming to real-time search. In fact, it is important to note that the parallelization proposed here fits any real-time search algorithm that behaves like LRTA* and all those discussed above.

III. Parallelization Methodology

In this section we explain the methodology used for the parallelization. We based our design on a master/slave protocol, with the exception that the master not just manages the tasks, but also performs a search under real-time constraints.

The execution flow on the main core divides in three sections: task processing, task management, and synchronization with the auxiliary cores. Task processing on the main core is basically an iteration of a real-time search algorithm, such as LSS-LRTA* [10]. The only difference here is related to the way heuristic values are stored: instead of a hash table, a sparse matrix is used. Considering a distributed memory system, there will be many memory transfers between cores. The use of a sparse matrix reduces the time spent with synchronization, eliminating steps of filling buffers. Due to the fact that the system can have limited memory (which is the case of the Cell/B.E.), a sparse matrix is also a better choice compared to a distributed hash, because it allows a better control of the size of data that will be transferred.

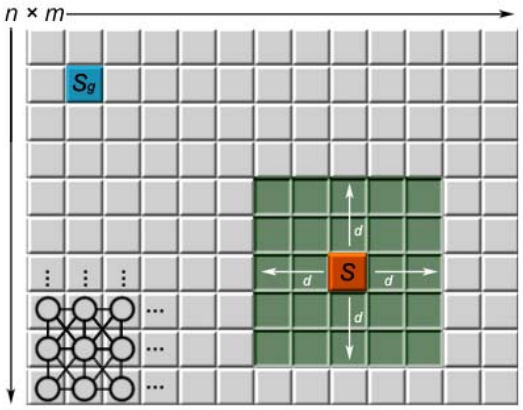

Figure 1: The task composition.

Before explaining how the master manages tasks, we first explain the concept of a task. A task is formed by 3 elements: a start node, the local search space, and a goal node. The structure of a task is shown in Figure 1. For an arbitrary map of dimensions $n \times m$, each tile in the figure matches a node in the grid representation of the map. A tile keeps information about the node navigability (if the node is passable or not), and the heuristic value from that position to the goal node S_g. The tile denoted by S represents the node where the auxiliary search will begin. The darker tiles represent the local search space, area delimited by the *lookahead* d. The larger is d, the larger will be the task size. Moreover, a larger d for the auxiliary searches imply on larger local search spaces, and therefore deeper searches, compared to the main search, which allows more data to be processed, and more heuristic values to be updated.

Tasks are created by the master core according to the following rule: each time the main search has traveled a path with an accumulated length multiple of d, nodes from this resulting path are chosen to form the S parameter of the new tasks. The first node selected to compose a task is the last one visited in the path. The second one will be the node on the path d nodes distant from the first, and so on for the next selected nodes. The distance that separates the selected nodes aims to avoid the overlapping of local search spaces between tasks, spreading the searches along the map.

After being created, a task is placed into a heap, to be distributed to the auxiliary cores. In order to reduce the time spent by the master with task creation, the tasks are created only with the S and S_g parameters. The main core lead this job to the auxiliary cores. This optimization reduces the time spent with synchronization, in order to keep real-time constraints on the master.

Tasks into the heap are sorted in a particular way, that differs from a simple FIFO queue. For the sorting, we created a rule based on swarm behavior [11]. Tasks are sorted inside the heap based on an amount, named *trace*, present in the tile that represents the S state. Basically, a node accumulates *traces* according to the following rule: each time a complete search is performed by the main core, a comparison is made between the cost of this search and a stored cost of a previous search. If the search cost is lower than the stored one, then each node in the path produced by this search receives *trace*, an allusion to the pheromone left by ants on the path they passed. The new cost then becomes the stored one, for further comparisons.

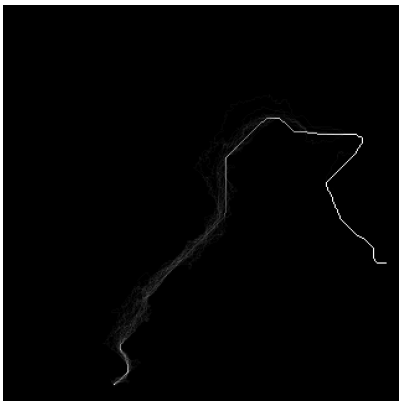

Figure 2: *Trace* trail for an arbitrary run.

The objective of storing *traces* is to form a sort of trail, according to the most traveled paths. Figure 2 shows the resulting trail for an arbitrary run, after the convergence to the minimal path. The more *traces* had been added to a node, the more highlighted it is. An important fact is that the states that form the optimal path lie within the trail region, and usually are more highlighted. Thus, the idea to give priority for states with more *traces* is to start the auxiliary searches in the region around the optimal path.

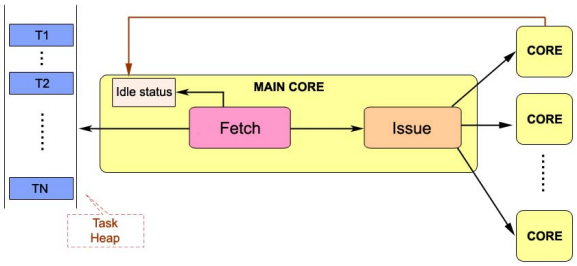

Figure 3: The synchronization between searches.

One important point of the parallelization regards the synchronization among the cores. Figure 3 shows how synchronization among searches works. This scheme was based on the task issuer created by Xia & Prasanna [12], for parallel Exact Inference on the Cell/B.E. The synchronization process begin when a core emits an idle signal and waits for synchronization. When the main core enters the synchronization step and reads the signal, it fetches a task from the heap and issues to the auxiliary core. After receiving the task, the auxiliary core request DMA transfer of data to completes its assembly, filling the tiles related to the local search space, and then begin the execution of the task. When execution ends, the updated heuristics are transferred to the sparse matrix on the main memory. The next task is then assembled by the auxiliary core, based on the agent's current position in the map, that will be the new S parameter. This last step is repeated, interleaving with execution, until the agent reaches the goal. Finally, a new idle signal is emitted, restarting the synchronization.

Another important detail upon the parallelization regards the distributed memory architecture. In order to hide the latency of memory transfers, the processing of tasks on the auxiliary cores are made using task buffers. Basically, each auxiliary core has two task buffers on its local memory. While a task is being transferred to the local memory of a core by the DMA controller, another one is being processed by this core. After processing, the core requests another task by synchronization, and starts processing the task that was transferred to the other buffer. Moreover, if we recall the fact that a core only synchronizes for another task when the search agent reaches the goal, we can realize that two independent searches are being performed by each core at the same time.

IV. Experimental Results

To analyze the performance and effectiveness of the proposed method, we did a series of experiments using different scenarios. Since real-time search algorithms are commonly used in digital games, we used different maps/graphs from

real game environments to perform the experiments. We measured the total time to convergence to the shortest path using the parallel method. We also evaluated the throughput for three distinct *lookaheads*. Finally, we show the results, measured in numbers of trials to convergence, for runs performed in a more realistic digital game setting, in which search should be interleaved with other tasks in the main core.

Due to the speculative nature of the heuristic based search algorithms, convergence is not regular for a given search. Searches performed in regions with more local minima tend to generate more task overlapping than the ones performed on regions with fewer local minima. Thus, the speedup is not uniform for all searches, and consequently, we did not evaluate it directly. Also, tasks being executed in parallel do not guarantee the order heuristic values will be updated. Furthermore, the fact tasks can update overlapping regions generates a non deterministic behavior. Thus, each result is given by the mean of five executions of the same search.

The experiments were executed on IBM BladeCenter QS20 blades. Each blade is composed by 2 Cell/B.E. 3.2GHz processors, with 1GB total memory. The Cell/B.E.is a multi-core chip composed of one Power Processor Element (PPE) and multiple Synergistic Processing Elements (SPE). The PPE and SPEs are linked together by an internal high speed bus called Element Interconnect Bus (EIB). All the SPEs access data from a limited local memory, named Local Storage. The EIB is responsible for DMA transfers between the main memory and the SPEs Local Storage. For further details, please refer to [3].

We have performed a total of 14 arbitrary searches, distributed among 13 distinct graphs. We used the LSS-LRTA* as the real-time search algorithm for both the main and the auxiliary searches. The choice of the LSS-LRTA as the search method, besides a better performance, given due to a better control of the local search space, based on the *lookahead* size. The *lookahead* is the parameter that delimits the search depth. The size of the local search space implies directly on the size of the tasks that will be created. This allows a more precise evaluation of the data transfers, a major factor on distributed memory systems. For the search performed on the main core, a *lookahead* of 5 was used, and for the auxiliary searches, the *lookahead* is 15, except for the throughput evaluation, where the *lookahead* varies between 5, 10, and 15.

Figures 4 and 5 show the total time to convergence to the optimal path, for runs related to the 14 arbitrary searches. The results, also shown in Table I, are organized as follows: the first seven searches (1-7) are performed on smaller maps, with fewer local minima, and consequently, requires less time for converging to the optimal solution. The other seven searches (8-14) are performed on larger and more complex maps, with more local minima, therefore requiring more processing and resulting in a slower convergence. For the parallel execution, the numbers shown in the table represent time segmentation for the three parts of the execution: task processing, synchronization between cores, and task management by the main core. Since the task processing step corresponds to the search itself, the last 2 sections cited imply on a overhead.

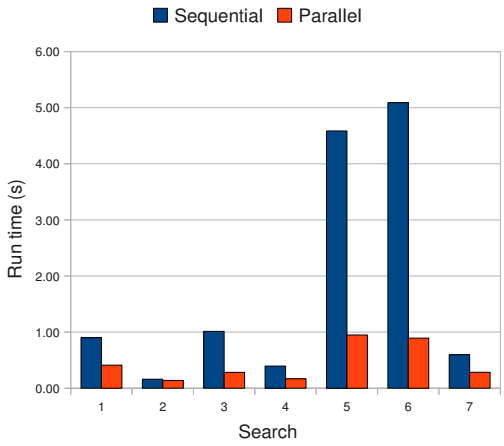

Figure 4: Total time to convergence. Searches 1 to 7.

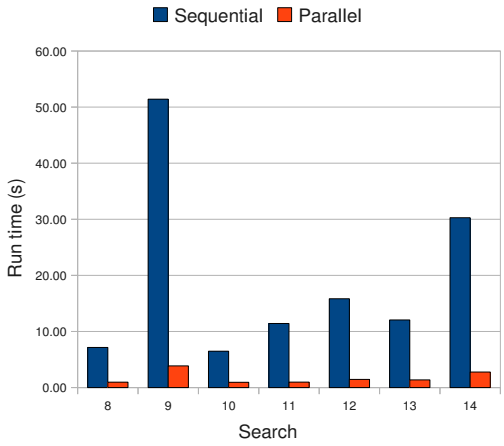

Figure 5: Total time to convergence. Searches 8 to 14.

The average overhead calculated for this searches represents 38% of the total time. This overhead lead to a performance loss, forcing the use of a smaller *lookahead* on the main search. A smaller *lookahead* implies on a smaller search space. Thus, a smaller area will be explored in the planning step, which makes the run to convergence more lengthy. However, even when using a small *lookahead*, we observe from the results that the average gain can be at about an order of magnitude. In addition, it is remarkable that the synchronization time is even proportionally smaller for the larger searches. This happens due to the fact that the

Section\Run	1	2	3	4	5	6	7
Processing	0.1678	0.0553	0.1336	0.0690	0.5614	0.5922	0.1219
Synchronization	0.2421	0.0829	0.1475	0.0984	0.3852	0.2977	0.1608
Management	0.0008	0.0003	0.0006	0.0004	0.0015	0.0024	0.0006
Total	**0.4107**	**0.1385**	**0.2817**	**0.1677**	**0.9481**	**0.8923**	**0.2833**
Sequential	**0.9010**	**0.1598**	**1.0130**	**0.3941**	**4.5859**	**5.0894**	**0.5975**

Section\Run	8	9	10	11	12	13	14
Processing	0.527	2.709	0.593	0.660	1.024	0.984	1.874
Synchronization	0.434	1.138	0.344	0.308	0.420	0.380	0.884
Management	0.003	0.015	0.003	0.004	0.005	0.006	0.007
Total	**0.964**	**3.862**	**0.940**	**0.971**	**1.449**	**1.369**	**2.765**
Sequential	**7.150**	**51.410**	**6.480**	**11.420**	**15.836**	**12.050**	**30.280**

Table I: Total time to convergence in the 14 different searches.

auxiliary cores just need to synchronize and ask for a new task when they complete their current searches, as described in Section III. Meanwhile, they have autonomy to assemble their own tasks, based on the current node the agent is. In general, there are gains in both kinds of searches, with the larger ones showing more relative gains than the others, due to a greater influence of the parallelization on the larger searches.

Figures 6 and 7 show the throughput for the main and the auxiliary cores, respectively, for the 12th search among the 14 ones. This search had been chosen due to its longer run to convergence, which results in a greater number of tasks to be processed, allowing a better evaluation of the throughput. While the *lookahead* for the main search was set to 5, three different values for the *lookahead* of the auxiliary ones was chosen: 5, 10, and 15.

As observed, the throughput on the master core decreases as the number of auxiliary cores increases. More cores being used in parallel increases the time spent by the master core with manage and synchronization, which explains the reduction in the number of tasks processed. Furthermore, the more auxiliary cores are used, the more searches are performed, and consequently more areas of the map are explored in parallel. Thus, the rate of heuristic values that are updated increases, decreasing the total time to convergence.

Differently from the main core execution, where the *lookahead* of the auxiliary searches does not have a great influence on the throughput, its straightforward to note its influence on the auxiliary cores, showed in Figure 7. In this case, the throughput decreases as the *lookahead* increases. These results reflect the two buffers way tasks are processed on the auxiliary cores, in order to hide the latency between memory transfers. Thus, while a task with *lookahead* 5 already fills this gap, increasing the *lookahead* makes the task processing more time consuming, reducing the throughput. However, this reduction in the throughput does not imply on reduction of performance by the algorithm. In fact, increasing the *lookahead* inflicts on a larger area of the heuristics table to be updated and, therefore, on a reduction of the total time to convergence.

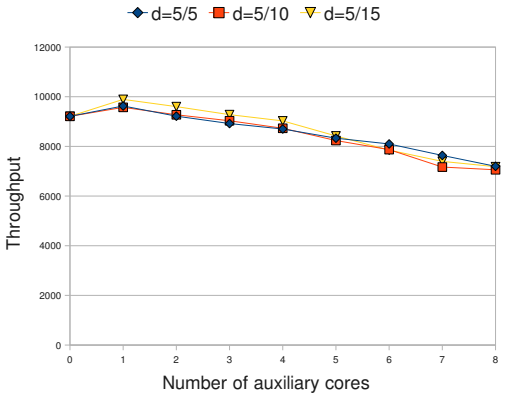

Figure 6: Throughput for the main core.

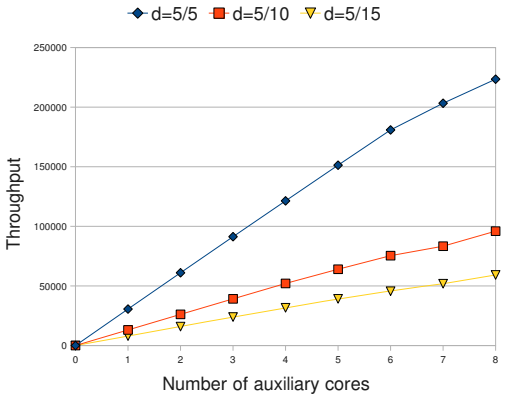

Figure 7: Throughput for the auxiliary cores.

So far, all experiments performed here consider the uninterrupted execution of the parallel algorithm. This means that the calls are made in a successive way, without any other kind of processing between them. However, real-time search is characterized by interleaving planning with other steps, such as execution. In some situations, such as digital games, there are even other types of processing, not related to the

search itself. Thus, to obtain a more realistic measure of performance, we modeled the time conditions to be similar to a digital game scenario

In a digital game scenario, all the processing is done inside a main loop called *game-loop*. Normally, this loop is executed 30 to 60 times per second (the term generally used is frames per second – fps) and is responsible for all the main tasks such as computing all the game physics, executing game AI and redrawing all the screen objects. Thus, the path planner that runs in the AI is interleaved with several other processing tasks. Sometimes it is even executed asynchronously, in a rate that is much lower than 30fps.

The frame rate on current console games uses the synchronization frequency fixed on $60Hz$. This means that the *game-loop* is executed 60 times in one second. Then, we can consider the time between each *game-loop* execution to be $1/60$ seconds. Thus, we can use this time interval to estimate the game-loop size. To simulate an execution of a real-time search under real conditions, we have to use an interval of $1/60$ seconds between calls of the planning algorithm. This interval represents the other processing jobs performed in the game-loop, including those not related to the search itself.

Method\Run	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Sequential	208	35	158	95	186	205	113
Parallel	34	13	22	12	9	7	15
Method\Run	**8**	**9**	**10**	**11**	**12**	**13**	**14**
Sequential	611	2058	727	1098	675	397	726
Parallel	13	20	26	26	8	7	5

Table II: Runs on a game execution scenario.

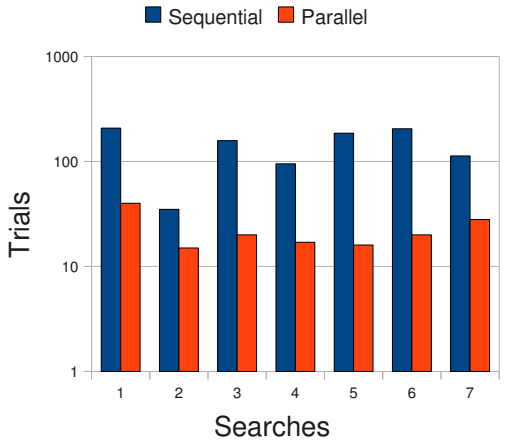

Figure 8: Runs on a game execution scenario. Searches 1 to 7.

Figures 8 and 9 show results for the execution of the 14 searches with the time interval between calls. The results, also shown on Table II, are measured in number of trials

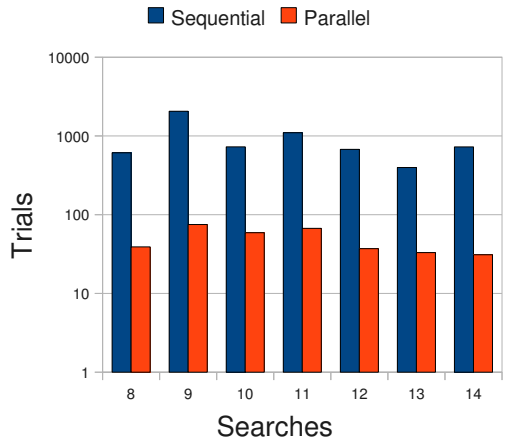

Figure 9: Runs on a game execution scenario. Searches 8 to 14.

to convergence, and presented on a logarithmic scale, for better visualization. We see from the results that the gains are much significant compared to the uninterrupted execution. In certain cases, the gains are about two orders of magnitude superior, being more than one hundred times faster than the sequential execution. This gain is possible due to the time interval inserted between calls. In this case, while the main core is busy with other processing tasks, the auxiliary cores continuously update the heuristic values in background. In addition, the decision to delegate the task assembling to the auxiliary cores, for current searches they are performing, decreases the overhead caused by synchronization on the main core and reduces the idleness of the auxiliary cores. Thus, even without an effective participation of the main core, we can observe the speedup on learning, compared to the sequential execution of the search.

V. Conclusion

This paper has introduced a parallelization method to speed up learning on real-time searches, and therefore the run to convergence to the optimal path. While other methods focuses on sequential solutions to speed up learning, we showed a method that use searches in parallel and share the acquired learning. We proposed a parallelization based on the master/slave protocol, designed for distributed memory systems. Basically, the searches executed in the auxiliary cores uses larger *lookaheads*, sharing the updated heuristic values, while keeping the constraints of a real-time search on the main core tasks. We also designed a particular sorting technique, based on swarm behavior, to assimilate priority to the tasks executed by the auxiliary cores. Our experiments have shown that the generated overhead may be compensated by a smaller *lookahead* without compromising the gains. The results showed that, using a *lookahead* three

times higher for the auxiliary searches, the average gains are one order of magnitude higher, comparing to the sequential execution of the same search. The results were even better when a game execution scenario is considered. In this scenario, while the execution of the search algorithm is interleaved with other tasks in the main core, the parallel executions in the auxiliary cores work on background, significantly reducing the convergence time.

ACKNOWLEDGMENT

This work was partially supported by CNPq and FAPEMIG. The authors acknowledge Georgia Tech, its Sony-Toshiba-IBM Center of Competence, and the NSF, for the use of Cell/B.E. resources that have contributed to this research.

REFERENCES

[1] R. E. Korf, "Real-time heuristic search," *Artif. Intell.*, vol. 42, no. 2-3, pp. 189–211, 1990.

[2] A. G. Barto, S. J. Bradtke, and S. P. Singh, "Learning to act using real-time dynamic programming," *Artif. Intell.*, vol. 72, pp. 81–138, January 1995. [Online]. Available: http://dx.doi.org/10.1016/0004-3702(94)00011-O

[3] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata, "Cell broadband engine architecture and its first implementation: a performance view," *IBM J. Res. Dev.*, vol. 51, pp. 559–572, September 2007. [Online]. Available: http://dx.doi.org/10.1147/rd.515.0559

[4] L. Rios and L. Chaimowicz, "A survey and classification of a* based best-first heuristic search algorithms," in *Advances in Artificial Intelligence - SBIA 2010*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, vol. 6404, pp. 253–262.

[5] T. Ishida and M. Shimbo, "Improving the learning efficiencies of realtime search," in *Proceedings of the thirteenth national conference on Artificial intelligence - Volume 1*, ser. AAAI'96. AAAI Press, 1996, pp. 305–310. [Online]. Available: http://portal.acm.org/citation.cfm?id=1892875.1892921

[6] T. Ishida, *Real-time search for learning autonomous agents*. Norwell, MA, USA: Kluwer Academic Publishers, 1997.

[7] D. Furcy and S. Koenig, "Speeding up the convergence of real-time search," in *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*. AAAI Press, 2000, pp. 891–897. [Online]. Available: http://portal.acm.org/citation.cfm?id=647288.721736

[8] C. Hernández and P. Meseguer, "LRTA*(k)," in *Proceedings of the 19th international joint conference on Artificial intelligence*. Morgan Kaufmann Publishers Inc., 2005, pp. 1238–1243.

[9] V. Bulitko, N. Sturtevant, and M. Kazakevich, "Speeding up learning in real-time search via automatic state abstraction," in *Proceedings of the 20th national conference on Artificial intelligence - Volume 3*. AAAI Press, 2005, pp. 1349–1354. [Online]. Available: http://portal.acm.org/citation.cfm?id=1619499.1619550

[10] S. Koenig and X. Sun, "Comparing real-time and incremental heuristic search for real-time situated agents," *Autonomous Agents and Multi-Agent Systems*, vol. 18, no. 3, pp. 313–341, 2009.

[11] J. Kennedy, R. Eberhart, and Y. Shi, *Swarm intelligence*. Morgan Kaufmann Publishers, 2001.

[12] Y. Xia and V. K. Prasanna, "Parallel exact inference on the cell broadband engine processor," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–12.