Supplemental Material for: Block-Parallel IDA* for GPUs

SoCS Submission 48 supplement

Abstract

This is the supplemental material for SoCS submission 48.

S.1 Thread-Based Parallel IDA* Details

We provide further details on the Thread-based parallel IDA* implementations which were sketched in Section 4 of the submission. The overall algorithmic scheme is shown in Alg. S.1.

PSimple (baseline) In this baseline configuration, for each f-bounded iteration of IDA*, CreateRootSet (Alg. S.1, line 20) performs A* search from the start state until unique states as many as the number of threads are in OPEN. Then, each root is assigned to a thread, No load balancing is performed, i.e., UpdateRootSet and DynamicLoadBalance do nothing. The subtree sizes under each root state can vary significantly, so some threads may finish subproblem much faster than other threads. Each f-bounded iteration must wait for all threads to complete, so PSimple has very poor load balance. Therefore, some *load balancing* mechanisms which (re)distribute the work among processors are necessary.

PStaticLB (static load balancing) This configuration adds static load balancing to PSimple. In IDA*, the i-th iteration repeats all of the work done in iteration $i-1$. Thus, the number of states visited under each root state in the iteration $i-1$ can be used to estimate the number of states which will be visited in the current iteration i.

UpdateRootSet (Alg. S.1, line 26) implements the following *static load balancing* mechanism. Let $load(n)$ be the number of nodes expanded in the previous iteration under node n. If $load(n) > averageload$, where $averageload$ is the average number of nodes expanded under all of the root nodes, we *split* n as follows. We then initialize $droots$ to ϕ, and perform an A* expansion of the search tree starting at n, adding generated nodes into $droots$, until $droots$ has $\geq load(n)/averageload$ nodes. Then, we remove n from the root set, set $load(m) \leftarrow load(n)/|droots|$ for each $m \in droots$, and add $m \in droots$ to the root set.

Then, we allocate each root in the root set to threads. For each thread t, roots are assigned to t until the sum of the $load(n)$ for the roots assigned to t exceeds $averageload$. Roots are allocated to threads in the order which they were generated – we experimented with other orders for this assignment but did not obtain significant differences.

A goal node might be found during this split-redistribute phase, but the path to such a goal node may or may not be optimal-cost, so it needs to be returned to OPEN without expanding it.

This rebalancing phase is performed on the CPU, so duplicate detection and pruning via the CLOSED list is performed during rebalancing.

PFullLB (thread-parallel with dynamic load balancing) This configuration adds dynamic load balancing to PStaticLB, i.e., the DynamicLoadBalance function (Alg. S.1, line 8) is enabled.

Dynamic load balancing moves work to idle threads from threads with remaining work *during* an iteration. On a GPU, work can be transferred between two threads *within* a single block relatively cheaply because this can be done using the shared memory within a block, while transferring work between two threads which are in different blocks is expensive because it requires access to the global memory. When dynamic load balancing is triggered, then within a block, idle threads steal work from threads with remaining work.

We experimented with various dynamic load balancing strategies including variants of policies investigated by (Powley and Korf 1989; Mahanti and Daniels 1993). The best performing policy for triggering dynamic balancing (the trigger is checked in algorithm S.1, line 8) is based on the policy by Powley and Korf: perform load balancing when the fraction of idle (completed) threads within a block exceeds $W(t)/(L+t)$, where L is the time spent for the previous load balancing operation, t is the time since the previous load balancing operation, $W(t)$ is the amount of work (number of nodes visited) since the previous load balancing operation. Note that time t is not wall-clock time since the last rebalancing, but

Algorithm S.1 Parallel IDA*

```
1: function DFS(root, goals, limit_f)
2:   openList ← root
3:   f_next = ∞
4:   repeat
5:     s ← POP(openList)
6:     if s ∈ goals then
7:       return s and its parents as a shortest path
8:     if dynamic load balance is triggered then DYNAMICLOADBALANCE(stat)
9:     for all a ∈ applicable_actions(s) do
10:      t ← successor(a, s)
11:      f_new ← g(s) + cost(a) + h(t)
12:      if f_new <= limit_f then
13:        openList ← t
14:      else
15:        f_next ← min(f_next, f_new)
16:  until openList is empty
17:  return f_next, stat                    ▷ no plan is found
18:
19: function PARALLELIDA*(start, goals)
20:  rootset ← CREATEROOTSET(start, goals)
21:  limit_f ← DECIDEFIRSTLIMIT(rootSet)
22:  repeat
23:    parallelForByThreads root ∈ rootset do
24:      limit_f, stat ← DFS(root, goals, limit_f)
25:    end parallelForByThreads
26:    UPDATEROOTSET(start, goals, rootSet, stat)
27:  until shortest path is found
```

the actual amount of GPU time consumed. In addition, load rebalancing is constrained so that it can not be triggered until at least $L/2$ seconds have passed since the previous rebalancing.

S.1.1 Evaluation of Thread-Parallel IDA*

PSimple on 1536 cores required a total of 3378 seconds to solve all 100 problems, a speedup of only 18.6 compared to G1 (1 core on the GPU).

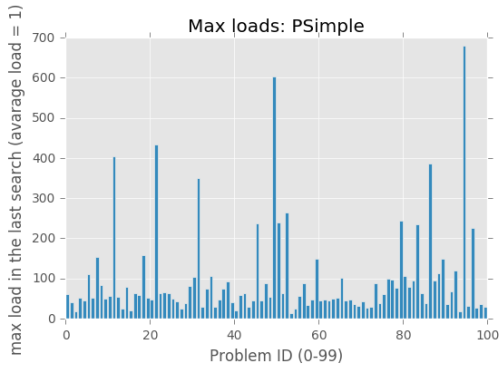


Figure S.1: Maximum thread workload in PSimple

We define load balance as maxload/averageload, where averageload is the average number of nodes visited among all threads, and maxload is the number of states visited by the thread which performed the most work (visited the largest number of nodes).

Figure S.1 plots, for each of the 100 problem instances, the maximum workload (number of nodes visited) among all threads of PSimple relative to the average workload, for the next-to-last iteration , i.e., a value of 1 means perfect load balance. We measure load balance for the next-to-last iteration because in the last iteration, only part of the tree needs to be expanded. Figure S.1 shows that some threads expand up to 600 times as many nodes as the average thread, indicating that PSimple has extremely poor load balance.

Figure S.2a compares the relative runtimes, i.e., the y-axis shows Runtime(PSimple)/Runtime(PStaticLB) for each instance. PStaticLB required a total of 1069 seconds to solve all 100 problems, a speedup of 3.16 compared to PSimple.

Figure S.2b plots, for each of the 100 problem instances, the maximum load (number of nodes visited) among all threads of PStaticLB relative to the average load for all threads for the next-to-last iteration. On one hand, Figure S.2b indicates that $maxload/averageload$ is significantly smaller than for PSimple. This explains the 3.16x speedup by PStaticLB compared to PSimple. On the other hand, the load balance is still quite poor, with some problems having a $maxload/averageload$ ratio > 50, with most problem instances having a $maxload/averageload > 5$.

Figure S.2d compares the relative runtimes, i.e., the y-axis shows Runtime(PStaticLB)/Runtime(PFullLB) for each instance. PFullLB required a total of 892 seconds to solve all 100 problems, a speedup of 1.20 compared to PStaticLB.

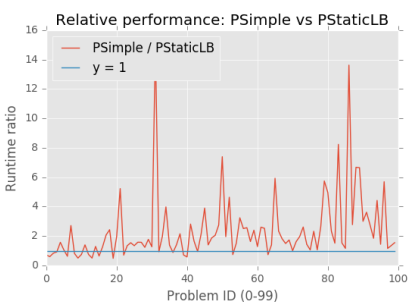
Figure S.2e plots, for each of the 100 problem instances, the maximum load (number of nodes visited) among all threads of PFullLB relative to the average load for all threads for the next-to-last iteration. Figure S.2f overlays the results in Figure S.2e with the results in Figure S.2b. Figures S.2e-S.2b indicate that $maxload/averageload$ is significantly smaller than for PStaticLB. In particular, the worst $maxload/averageload$ ratios have been reduced to < 20 (compared to > 50 for PStaticLB). Nevertheless, most of the $maxload/averageload$ ratios are > 4, indicating that even the combination of static and dynamic load balancing is insufficient for achieving good load balance.

We experimented extensively with both static and dynamic load balancing strategies, but so far, we have not found any strategy/configuration that significantly outperforms the results presented here.

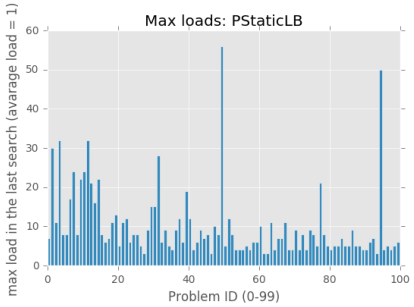
References

Mahanti, A., and Daniels, C. J. 1993. A SIMD approach to parallel heuristic search. *Artificial Intelligence* 60(2):243–282.

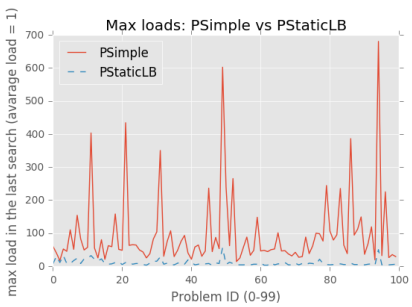
Powley, C., and Korf, R. E. 1989. Single-agent parallel window search: A summary of results. In *IJCAI*, 36–41.



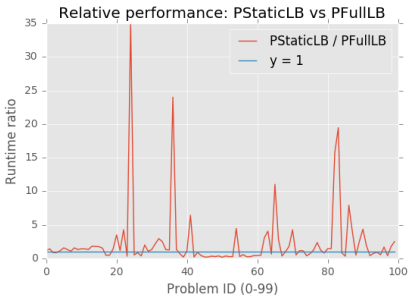
(a) Relative runtimes: PSimple vs. PStaticLB



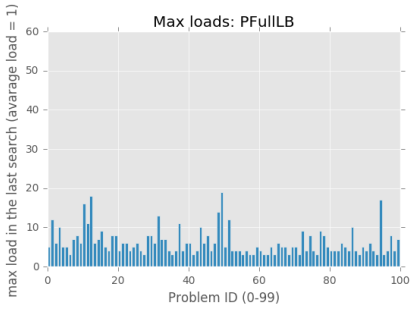
(b) Maximum thread workload: PStaticLB



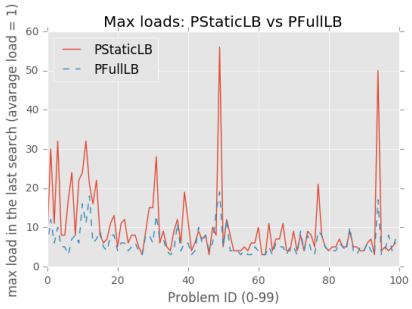
(c) Max workload comparison: PSimple vs. PStatisLB



(d) Relative runtimes: PStaticLB vs. PFullLB



(e) Maximum thread Workload: PFullLB



(f) Max workload comparison: PStaticLB vs. PFullLB

Figure S.2: PStaticLB Evaluation