Efficient Parallel Implementations of Controlled Optimization of Traffic Phases

Sameh Samra, Ahmed El-Mahdy*, Walid Gomaa*
Egypt-Japan University of Science and Technology (E-JUST), Egypt

Yasutaka Wada
Faculty of Science and Engineering, Waseda University
Japan

Amin Shoukry*
Egypt-Japan University of Science and Technology (E-JUST), Egypt

Abstract. Finding optimal phase durations for a controlled intersection is a computationally intensive task requiring $O(N^3)$ operations. In this paper we introduce cost-optimal parallelization of a dynamic programming algorithm that reduces the complexity to $O(N^2)$. Three implementations that span a wide range of parallel hardware are developed. The first is based on shared-memory architecture, using the OpenMP programming model. The second implementation is based on message passing, targeting massively parallel machines including high performance clusters, and supercomputers. The third implementation is based on the data parallel programming model mapped on Graphics Processing Units (GPUs). Key optimizations include loop reversal, communication pruning, load-balancing, and efficient thread to processors assignment. Experiments have been conducted on 8-core server, IBM BlueGene/L supercomputer 2-node boards with 128 processors, and GPU GTX470 GeForce Nvidia with 448 cores. Results indicate practical scalability on all platforms, with maximum speed up reaching 76x for the GTX470.

Keywords: parallel processing, high performance computation, traffic phases

1 Introduction

Traffic control requires real-time adjustment of the traffic signal [1,2] in order to ensure smooth and safe passage of vehicles. It is a key application in modern cities and has a strong effect on the quality of life and economy of these cities. However, optimal traffic control is computationally intractable, NP-hard problem, generally requiring an exponential order of computational steps. Moreover, traffic control requires real-time response, as it controls live traffic flows.

This paper focuses on the control of a single traffic intersection, as well as many independent intersections. This problem asks for obtaining an optimal sequence of green durations for every possible traffic flow (phase) across the intersection. It is essentially a sequential decision making problem that requires solution paradigms such as dynamic programming

With the increasing trend in utilizing multicore in today's processors, a tremendous increase in processing power have become possible at the expense of a

* Currently on-leave from Alexandria University

substantial change in the programming model. Harnessing this processing power requires devising explicitly parallel algorithms that match the characteristics of these processors.

In this paper, we study the utility of modern parallel architectures for the single traffic intersection problem. A parallel algorithm is developed based on a sound serial algorithm (developed by Sen and Head [3]). The serial algorithm is based on dynamic programming, and has time complexity of $O(N^3)$ for one intersection, where N is the net number of discrete time steps that represent the temporal window over which the global decision making problem is solved. Our parallel algorithm is cost-optimal achieving a time complexity of $O(N^3/p)$, where p is the number of processors. The paper investigates the implementation of the algorithm on a variety of platforms that include a multicore server (8-core), a GPU (448-core), and the IBM BlueGene/L supercomputer (2-node boards, 128 processors). The study identifies the scalability and the minimum problem size for a given platform, aiding in possible design and implementation of future traffic control systems on a variety of platforms.
More specifically the paper has the following contributions:

- A parallel cost-optimal control algorithm for a single intersection based on Sen and Head's algorithm [3] with:
  - A corresponding a shared-memory (OpenMP) implementation.
  - A Message-passing (MPI) implementation that incorporates loop reversal, communication messages pruning, and load-balancing optimizations.
  - A Data-parallel (CUDA) implementation that incorporates loop reversal, efficient use of caches, and efficient assignment of threads to hide latency, and utilize GPU parallelism.
- Experimental performance evaluation of the above implementations on different parallel platforms.

The paper is organized as follows: Section 2 briefly reviews related work. Section 3 provides background and definition of the intersection control problem including the serial intersection algorithm. Section 4 introduces our parallel algorithm and develops its the time complexity. Sections 5 and 6 explain our message passing and data parallel implementations, respectively; each section provides necessary background on the programming model, and the implementation algorithm. Section7 provides the experimental setup and results. Section 8 concludes the paper.

2 Related Work

In the following, a brief literature survey about parallelization of dynamic programming and the traffic control problem is given.

GPUs have been used to accelerate the execution of an instance of dynamic programming called the Smith-Waterman algorithm [4]. In [5, 6, 7] the authors give different parallel implementations of this algorithm on GPUs.

Heunget et al. [8] give a dynamic programming based algorithm for traffic control. Their approach is decentralized and based on installing local controllers at the junctions of traffic lights. These controllers are physically and functionally independent employing fuzzy logic and genetic algorithms to handle the local control

and the learning process, respectively. Coordination is introduced among the local controllers to derive optimal green time decisions using a global dynamic programming algorithm. The algorithm is a conventional serial CPU based implementation. A parallel solution of the traffic control problem is given in [9]. The authors employ a game-theoretic approach of a fictitious play to iteratively find coordinated traffic split plan.

The work done in [10] views the traffic control problem as an online optimization problem that is characterized both as nonlinear and non-convex. The authors reformulate the problem as a mixed-integer linear programming MILP (an LP mathematical program where some of the variables have to be integers). MILP solvers have already been around in the literature. Though, the authors propose an approach (non-parallel) to reduce the complexity of the MILP optimization problem in order to increase the real-time feasibility of the optimization problem.

The serial dynamic algorithm we utilize is serial polyadic dynamic programming algorithm; though computations happen in regular epochs, the communication pattern is variable and different than existing solutions in the literature.

3 Serial Intersection Algorithm

Algorithm 1 presents the serial algorithm Controlled Optimization of Traffic Phases (COP) for traffic control given in [3]. First, the basic conventions and notions are given, followed by the algorithm itself. Let r be a constant integer denoting the effective clearance is the amount of time necessary to make a safe phase transition. Let γ be a constant integer denoting the minimum green time. The control/decision variable x_j denotes the amount of green time allocated for stage (phase) j. The state variable s_j represents the total number of discrete time steps that have been allocated so far after stage j has been completed. The space of all feasible control decisions at state s_j is denoted $X_j(s_j)$. The value function and the performance measure at stage j are $v(s_j)$ and $f(s_j,x_j)$ respectively. T denotes the total number of discrete time steps (the temporal window over which the decision making problem is solved). This parameter plays an essential role in the parallel version of the algorithm as will be seen later.

1 procedure COP(T,r,γ)
2 begin
3 Initialize j:= 1, s_0:= x_0:= v(s_0) = 0;
4 **for** s_j = r **to** T do // *running* over all possible values of the state variable at stage j
5 **begin**
6 **If** $s_j - r < \gamma$ **then**// the case where the allocated green time is less than the min. threshold (γ)
7 $X_j(s_j)$:= {0}; // in such case no green time should be allocated for phase j
8 **else**
9 $X_j(s_j)$ = {0, γ , γ + 1, … , $s_j - r$};// otherwise, these are the feasible values for green time at state sj
10 **For each** $x_j \in X_j(s_j)$ **do** // choose the best value for the decision variable (green time) according to the value fn
11 **begin**
12 **If** x_j == 0 **then**

13 $s_{j-1} := s_j$; // phase j is allocated zero time
14 **else**
15 $s_{j-1} := s_j - x_j - r$;
// forward recursion, clearly sj = sj-1 + xj + r (for the current suggested value of xj)
16 $v_j'(s_j,x_j) := f_j(s_j,x_j) \bullet v_{j-1}(s_{j-1})$; // evaluating the value fn at the particular state sj with the control variable set to xj
17 **end**// end for xj
18 $x_j^* := \text{argmin}_{xj} v_j'(s_j,x_j)$;// computing the optimal value of the control variable at state sj
19 $v_j(s_j) := v_j'(s_j,x_j^*)$;// computing the optimal value fn at state sj given the optimal value of the control var xj*
20 **end** // end for sj
21 $j := j + 1$; // next stage
22 Repeat the above loop (line 4) until the stopping criterion is satisfied
23 end// end procedure COP

Algorithm 1: Serial Algorithm

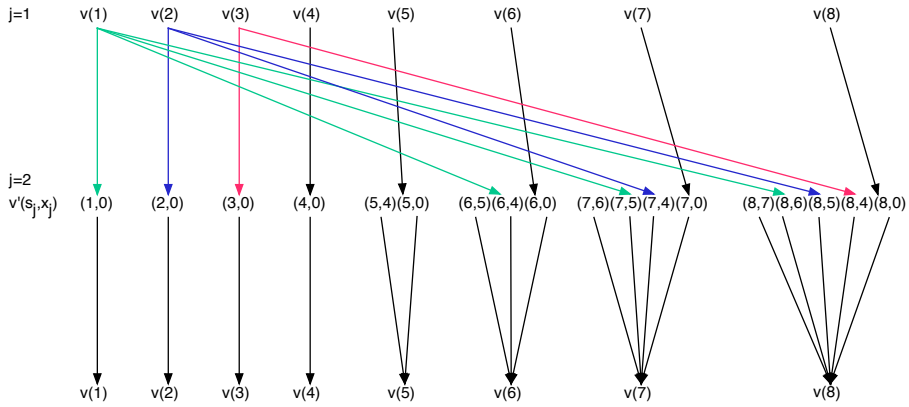

Fig. 1. Task Dependence Graph (T=8, γ=4, r=1)

The stopping criterion in line 22 is reached when no change occurs in the value function for the last ϱ stages, where ϱ is the total number of phases for the given intersection. In such a case, no further phase change would incur any difference in the value function; hence the optimization process is stopped.

The time complexity of calculating the value function in line 16 depends on the particular choice of the performance measure $f_j(s_j,x_j)$ and the particular implementation, especially considering the time/space trade-off (for example, whether or not to use a lookup table to store old values instead of re-computing them). Let N be the total number of processed time steps per stage ($N=T-r+1$) and assume line 16 takes O(m) steps, where m is a function of N that depends on the particular choice of the performance measure. Then the time complexity of computing each stage j is O(mN^2). To compute the complexity of the whole algorithm we need to compute the maximum possible number of stages. Based on the stopping criterion mentioned in the previous paragraph and the worst-case scenario that each phase with non-zero duration takes only 1 time step (γ=1), and assuming zero clearance interval

(r=0), we can have at most ϱ N stages. ϱ is a constant, hence the total number of stages is O(N) and the whole algorithm takes O(mN^3) computation steps. In our implementations, we consider the cost function to be the cars queue length as defined in the COP algorithm [3]. Computing this cost function is O(1), thus the overall complexity is O(N^3).

Fig. 1 shows the task dependence graph of a stage of the serial algorithm described in the previous section. The figure assumes r=1, T=8, and γ=4. The top row lists the obtained $v(sj)$ values from the first iteration (j=1). Obtaining these values only depend on the initial conditions. The middle line shows the required computations $v'(s_2,x_2)$. For T higher than 8, the graph can be easily extended to the right to $v(T)$ by:

1) Extending the first row from $v(8)$ to $v(T)$.
2) Extending the second row with T-8 new clusters of $v(s_j,x_j)$ such that each cluster corresponds to an s_j with entries $(s_j,0), (s_j,\gamma), (s_j,\gamma+1)$,..., (s_j,s_j-r).
3) Connect a directed edge from $v(sj)$ to each new $v(s_j,x_j)$, where $l_j = s_j-r-\gamma$ for $sj>r+\gamma$, and $l_j = 0$, otherwise.
4) Similarly, extend the last row from $v(8)$ to $v(T)$.

The $v(sj)$ values are obtained as a reduction operation on the $v'_j(sj)$ values. For higher values of j, the graph would simply require repeating the j=2 portion of the dependence graph.

4 Shared Memory Parallel Intersection Control Algorithm

Our parallel formulation of the algorithm is to assign processor p_j to state s_j for j = 1, ... , N. Each processor is, therefore, responsible for computing the $v'_j(s_j,x_j)$ value as well as the corresponding optimal parameters x_j^* and $v_j(s_j)$. The execution time complexity for each stage would be O(mN) giving a total parallel time complexity of O(mN^2), which when multiplied with the number of processors N would give the serial time complexity O(mN^3). It is worth noting that an execution time of O(mN log N) can be achieved using N^2 processors. This can be done by requiring each processor to compute exactly one $v'_j(s_j,x_j)$ value, and this can be done in O(m) time. Then computing their minimum, that is the $v_j(s_j)$ value, by arranging the processors into a balanced binary tree, where at level i of the tree a total of $N^2/2^i$ different partial minimum values are computed. The depth of the tree is O(log N), and an upper bound of computation cost O(N) is needed at each level of the tree. Hence, a total of O(mN log N) is achieved. However, such algorithm is not cost optimal, since in such a case the product of the parallel complexity O(mN log N) with the number of processors N^2, giving O(mN^3 log N), is not equal to the serial complexity of O(N^3). The degree of parallelism in the problem exceeds the number of cores in commodity[1]. It is also worth noting that the latter algorithm will be useful when computing several intersections in a pipelined fashion.

[1] In the GPU case, more thread is favoured, but the control-flow may not be suitable for data-parallel nature of the GPU.

The shared memory implementation is done by parallelizing the loop at lines 4-20 (Algorithm 1). $v()$, x^*, are designated as shared variables.

5 Message Passing Implementation

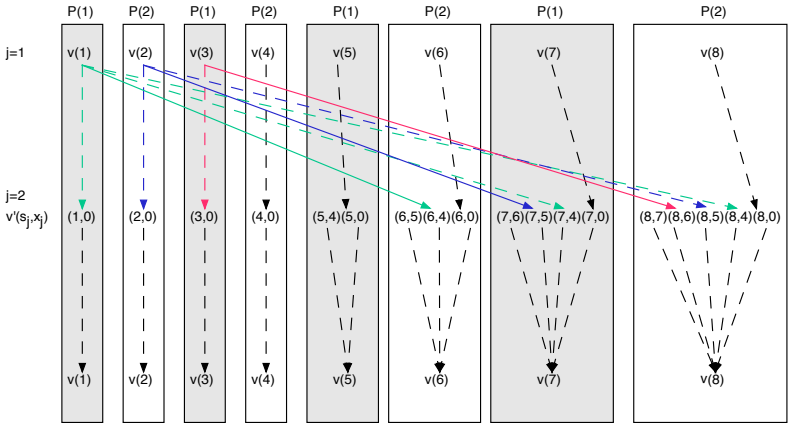

Fig. 2. Processor Assignment Effect on the Task Dependence Graph (T=8,γ=4,r=1)

The message-passing programming model has the advantage of allowing more scalability than the shared-memory programming model. This is due to the fact that shared memory architectures are economically more expensive in addition to the overhead of memory coherence. On the other hand, message passing programming is much more complex.

Since the degree of parallelism in the COP algorithm is quite large, and to allow for scalability studies, we designed the parallel version such that the actual number of processors can be smaller than the theoretical number of processors identified in our main parallel formulation. This has introduced the problem of mapping many logical processors into one physical processor. It is worth noting that simulating parallel execution of the logical processors lead to very poor results due to the fact of sending/receiving many unnecessary messages. The communication pattern can negatively affect the performance by having the processors remaining in idle state for long periods of time. This case will be discussed after presenting the algorithm.

Fig. 2 illustrates our main idea about the mapping of logical processors into physical ones, pruning redundant messages. The figure shows an interleaving degree of 2. The solid arrows show actual communication/messages between processors, whereas the dashed arrows indicate redundant communication (the messages have already been sent to the same physical processor). The interleaving of the logical processors among the available pair of physical processors (the odd-numbered logical tasks are assigned to physical processor 1 and the even-numbered logical tasks are assigned to physical processor 2) allows for uniform distribution of the messages among the processors, thereby decreasing idle time and avoiding deadlocks. If logical processors are to be grouped as contiguous blocks where each block is assigned to some physical processor, the processors on the left-hand side will only send to

processors to the right side resulting in a serial execution of tasks. Deadlocks are prevented by executing from left-hand side to right-hand side.

Messages among logical processors that are assigned to the same physical processor are avoided. For example, P(1) does not send $v(1)$ to the state-decision pair (7,5) as it is already assigned to the same physical processor P(1). Also, redundant communication is avoided. For example, P(1) does not send $v(1)$ to P(2) in the computation of the pair (8,6) since P(1) has already sent the same message to P(2) when computing the pair (6,4).

```
1 procedure Solve(rank, p, r, γ)
\\ rank: processor id
\\ p: number of processors
2 begin
3    j := 1; sj_1 := 0;  v := 0;   \\ v is the value function
3    while checkStoppingCriteria(v,j-1)
4    begin
5       for sj := r + rank to T step p
6       begin
7          TargetProcIds := {sj + r + γ, … ,Min(sj + r
            + γ + p, T)};
8          if j > 1 then
9             AllToAll v [si][j-1], xj_star[sj][j-1]
10      end
11   end
12   v [sj][j] = Min_xj_star (fj(sj,xj) . v [sj_1][j-1])for
all xj in { sj, sj-1, …, γ, 0};
13   xj_star[sj][j] = xj_star;
14end
15j:=j+1;
16end
17 ReportSolution();
```

Algorithm 2: MPI Solve

Algorithm 2 gives the pseudo-code of our MPI Solve procedure using MPI communication method of **AllToAll()**. AllToAll() send given data to all processors in the cluster and it is cost optimized to be O(p) only [14]. Each processor executes the Solve procedure with *rank* set to the processor's id. The algorithm proceeds similar to the serial one till line 4. Line 5 distributes the loop iterations among p processors such that iteration k is given to processor $k - r \bmod p$.

The first stage of computation (j=1) relies on the initial conditions and no message is communicated among processors. For j>1, line 9 sends the previous stage messages to the corresponding physical processors (the list of processors in *TargetProcIds*). It is worth noting that the number of messages sent is at most p. Also line 9 receives the previous stage data from corresponding physical processors. The function $h(x_j)$ in line 14 is defined as follows:

$$\mathrm{h}(x_j) = \begin{cases} 0 & x_j = 0 \\ x_j + r & \text{Otherwise} \end{cases} \qquad (1)$$

The implementation of parallel checkStoppingCriteria procedure differs from the serial algorithm in that processor with rank p-1 is responsible for checking the stopping condition when $j > \varrho$. The processor broadcasts its decision to other processors. Other processors receive the decision from that processor.
It is worth noting that the stopping criteria computational complexity is $O(\varrho) = O(1)$, each computational step requires O(log p) communication steps to broadcast the decision, therefore it does not interfere with the overall complexity of the algorithm<$O(N^3/p)$. Also, practically, ϱ is much smaller than the number of processed time slots *(N)*; therefore its contribution is negligible.

6 Data Parallel Implementation

General Purpose Graphics Processing Units (GPGPUs) favour data parallel programming models. OpenCL [11] and CUDA [12] are popular data parallel models targeting GPGPUs, that are similar; this paper focuses on CUDA being more mature and easier to program than OpenCL as the latter is targeting diverse GPUs from different vendors.

In the CUDA programming model, data parallelism is expressed in a form of Single-Program Multiple Data streams (SPMD) model. A group of programs (i.e. threads) is executed in a lock-step fashion. Such a group is called 'warp' that is of size 32 in current GPUs. Warps are collectively executed on a Streaming Multiprocessor (SM). In CUDA notation, collections of threads are called 'thread blocks'. One noticeable feature of thread blocks is the availability of thread synchronization operation, as well as shared memory. Thread blocks share global memory with high bandwidth but of high latency. The latter is potentially hidden using the underlying multithreading execution engine of the GPU.

Recently, the current generation of CUDA enabled GPUs (code named Fermi) supports data caches [13]. There are L1 data cache for each SM and one L2 data cache for global memory. In our algorithm, we, therefore, maximize data reuse to increase cache hit ratios.

An important performance aspect of our algorithm is load-balancing the stage iteration loop. Mapping many iterations into a thread is not plausible here as it would decrease multithreading and, therefore, expose GPU's high latency. Moreover, threads will have different control-flow patterns, which are not plausible for the underlying data-parallel execution engine, resulting in serialization in the execution of threads. Our algorithm deals with this case by mapping each iteration linearly to threads; since the warp size is small with respect to the total number of threads, we only waste negligible cycles (provided N is much bigger than warp size).
CUDA GPUs support memory coalescing, where same memory access from threads from the same warp is coalesced saving memory bandwidth. To achieve this, the loop reversal in the earlier algorithm (MPI), in addition to the linear mapping of sj's into threads result in all threads having similar data accesses (access the same element v).
Another aspect is the high latency of GPUs, which should be hidden by using multithreading. Generally, the latency results from accessing global memory (100s of cycles in current GPUs) as well as dependent read-after-writes register operations (10 cycles); since the algorithm optimizes cache performance, the latter smaller latency is

required to hide, which requires number of threads proportional to the latency value. The algorithm, therefore, assigns each thread to *sj*; assigning an *v'*() would result in higher degree of parallelism by that would be traded off with synchronization among threads and possibly blocks (for larger *T*) which would significantly reduce the regularity of control-flow and synchronization overhead among blocks. Increasing the number of threads can however increase the number of communication messages among threads; however, the GPU has high communication bandwidth to global memory (1 word per 32 instructions on average for the GTX470) that exceeds the required global memory access (approx. N^2 global memory accesses/ N^2 computations = 1 word per operation, where the operation easily exceeds 32 instructions).

The use of data caches in CUDA GPUs allows for reducing the communication from global memory. Our linear mapping achieves the same interleaving behavior we utilized in the MPI algorithm, but this time among the blocks, thereby each block behaves as a faster shared memory parallel subsystem, and the blocks operate in slower shared memory system.

Algorithm 3 is the data parallel version of the algorithm. The stopping criterion is similar to Algorithm 3 of the MPI, but thread synchronization is implemented by a call to a host function, as the ratio of synchronizations to computations is 1 : *N*, amortizing the overhead of synchronization.

```
1 procedure Solve(blockidx, blockwidth, threadidx)
\\ blockidx: block index
\\ blockwidth: number of threads per block
\\ threadidx: thread number within a block
2 begin
3    j:=1;sj_1:=0; ν:=0; \\ ν is the value function
4    sj := blockidx × blockwidth + threadidx + r
5    while checkStoppingCriteria( ν,j-1)
6    begin
7      Compute Solution as per lines 6-19 of algorithm1
8      j:=j+1;
9    end
10    if sj = T
11    begin
12       ReportSolution();
13    end
14 end
```

Algorithm 3: Data Parallel Algorithm

7 Experimental Study

We have implemented the algorithm using three parallel programming models: (1) OpenMP, which is a shared memory model, (2) MPICH, which is a message passing model, and (3) CUDA, which is a data parallel model. The platforms used for our algorithms are the following: OpenMP runs on a machine, named fuji, with two Quad-Core processor (Intel Xeon), hence an overall of eight processing units (8

cores). MPICH runs over the same fuji machine as well as the IBM supercomputer BlueGene/L, the latter with 128 processors. The machine used for CUDA is an Intel Core I7 machine with one quad-core processor, though the algorithm is run using its GPU which is a GTX470 GeForce Nvidia with 448 cores.

According to the nature of the particular problem we investigate, traffic control, three input parameters control the workload: the number of phases, and the total number of discrete time steps T which represents a finite future horizon over which the optimization algorithm considers for decision making, and the traffic load itself (for example, the rate of cars entering the intersection). As can be seen from the parallel algorithm, only T would control the degree of parallelism, hence it is the only parameter varied in our experiments as seen below. The other parameters would just add more parallel iterations and hence would scale the obtained curves. Therefore, they are kept fixed throughout all our experimentation (we have used the values from the original paper [3]) : the number of phases is 3 and the car generation is scheduled according to Table 1 in [3]. In the experiments, we vary workload and number of processors and measure the corresponding execution speed, Intersections/Sec.

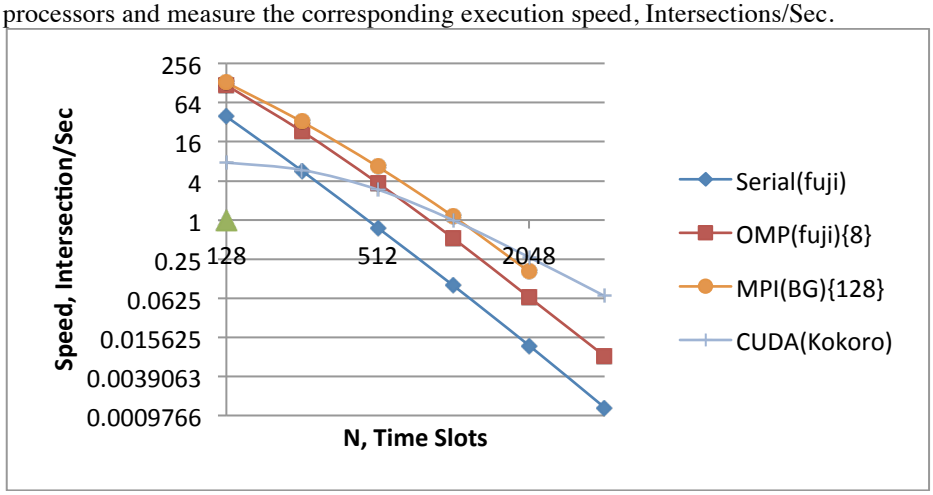

Fig. 3. Best Parallel Execution for a Single Intersection

Fig. 3 plots the execution speed (in Intersection/Sec) while changing the number of 'Time Slots', N. The 'Serial' case represents the serial algorithm run on fuji machine; with the increase of N, the execution speeds decrease quadratically, reaching 0.1 Intersection/Sec for $N = 1024$.

The OMP(fuji){8} represents the case of OpenMP algorithm over 8 cores, on the machine fuji. It shows a speedup of 5 to 6 with respect to the 'serial' case, with increasing of N. The MPI(BG){128} represents the case of MPI algorithm over 128 cores. It shows speedup of 4 to 13 times over the serial case.

The CUDA(kokoro){448} represents the case for the CUDA algorithm over 448 cores on the machine kokoro. The performance for small N is smaller than all other cases; this is mainly due to the lack in the number of threads to hide latency and make sure of GPU parallelism. With increasing N, the GPU significantly improves. The GPU becomes dominant after N=1024, reaching a speedup factor of 52 over the serial processor for N=4096.

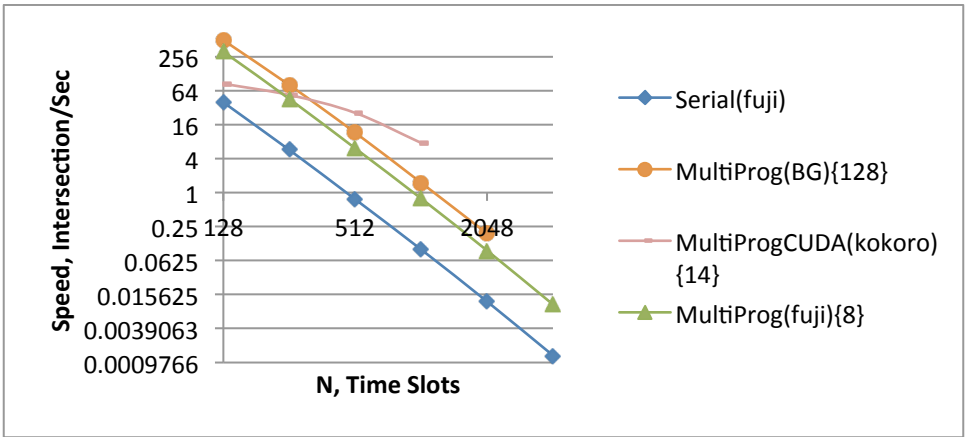

Fig. 4. Best Execution Speeds with Many Intersections

The analysis above focused on improving the execution time of a single intersection. However, if processing a single intersection time is not critical, and the throughput of processing many intersections is critical, simple multiprogramming would suffice and provide good scalability. Fig. 4 explores that aspect. We still include the 'Serial' case as a baseline reference. The MultiProg(BG){128} and MultiProg(fuji){8} are scaled up execution of the serial case for each platform using the maximum available processors; the former using 128 processors, and the latter using 8 processors. The MultiProgCUDA(kokoro){14} allow for running 14 intersections on the same GPU, each in a single SM. That case combines parallelism within an intersection, and among other intersections. The results show that CUDA is up to 76 times faster than the serial case. However, N cannot be 2048 or higher due to reaching the memory capacity. The CUDA case is up to 5 times faster than the BG.

8 Conclusion

This paper introduced a cost-optimal, parallel version of an optimal traffic control for an intersection [3]. Given the diversity of existing parallel architectures, the paper developed three different parallel implementations, using well-known parallel programming models: shared memory, message passing, and the recently introduced CUDA data parallel models. Experimentations on typical high-end servers, BlueGene/L supercomputer, and CUDA GPU, have shown that the parallel algorithm is scalable on all of the parallel programming models; comparing with the serial algorithm, speedups up to 76x has been achieved.

CUDA GPU performance is superior to BlueGene/L server. But, on the other hand, BlueGene/L is more scalable than CUDA GPU.

Future work would target other parallel programming models such as MapReduce and n-body models. Also, we would extend the algorithm to optimize traffic control on multiple intersections.

Acknowledgments. The authors would like to acknowledge their fruitful discussions with Dr Hisham El-Shishiny, manager of IBM Center for Advanced Studies in Cairo. The authors would like also to acknowledge the support of Mohamed Baddar, from IBM Center for Advanced Studies in Cairo, who helped with running the developed codes on IBM BlueGene/L super computer. This work is partially funded by IBM PhD Fellowship and Faculty Award.

References

1. G. Zhou, A. Gan, and L. Shen: Optimization of Adaptive Transit Signal Priority Using Parallel Genetic Algorithm. Tsinghua Science & Technology, vol. 12, no. 2, pp. 131–140, Apr. (2007)
2. R. Machemehl and M. Shenoda: Development of a Phase-by-Phase, Arrival-Based, Delay-Optimized Adaptive Traffic Signal Control Methodology with Metaheuristic Search. Center for Transportation Research, University of Texas at Austin (2006)
3. S. Sen and K. Head: Controlled optimization of phases at an intersection. Transportation science, vol. 31, no. 1, pp. 5–17 (1997)
4. M. S. Waterman and T. F. Smith: Identification of common molecular subsequences. J. Mol. Biol., vol. 147, pp. 195–197 (1981)
5. S. Manavski and G. Valle: CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. BMC Bioinformatics, vol. 9, no. 2, p. S10 (2008)
6. S. Xiao, A. M. Aji, and W. Feng: On the Robust Mapping of Dynamic Programming onto a Graphics Processing Unit. Parallel and Distributed Systems, International Conference on, vol. 0, pp. 26–33 (2009)
7. T. R. P. Siriwardena and D. N. Ranasinghe: Accelerating global sequence alignment using CUDA compatible multi-core GPU. in Information and Automation for Sustainability (ICIAFs), 2010 5th International Conference on, pp. 201–206 (2010)
8. T. H. Heung, T. K. Ho, and Y. F. Fung: Coordinated Road-Junction Traffic Control by Dynamic Programming. IEEE Transactions on Intelligent Transportation Systems, vol. 6, no. 3, pp. 341–350, Sep. (2005)
9. S. Cheng and M. Epelman: CoSIGN: A Parallel Algorithm for Coordinated Traffic Signal Control. IEEE Transactions on Intelligent Transportation Systems, vol. 7, no. 4, pp. 551–564 (2006)
10. S. Lin, B. De Schutter, Y. Xi, and H. Hellendoorn: Fast Model Predictive Control for Urban Road Networks via MILP. IEEE Transactions on Intelligent Transportation Systems (2011)
11. J. E. Stone, D. Gohara, and G. Shi: OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. Computing in Science & Engineering, vol. 12, no. 3, pp. 66–73 (2010)
12. D. Kirk and W. Hwu: Programming Massively Parallel Processors. Morgan Kaufmann (2010)
13. NVIDIA CUDA C Programming Guide. 3rd ed. NVIDIA Corporation (2010)
14. Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar: Introduction to Parallel Computing, Second Edition. Addison Wesley (2003)