Journal of Engineering Science and Technology
Vol. XX, No. Y (Year) PPP - QQQ

EFFICIENT SCHEDULING OF DYNAMIC PROGRAMMING ALGORITHMS ON MULTICORE ARCHITECTURES

TAUSIF DIWAN*, S. R. SATHE

Department of Computer Science and Engineering,
Visvesvaraya National Institute of Technology, Nagpur, Maharashtra, India
*Corresponding Author: tausifdiwan.tausif@gmail.com

Abstract

Dynamic programming is one of the Berkley 13 dwarfs widely used for solving various combinatorial and optimization problems, including matrix chain multiplication, longest common subsequence, binary (0/1) knapsack and so on. Due to nonuniformity in the inherent dependence in dynamic programming algorithms, it becomes necessary to schedule the subproblems of dynamic programming effectively to processing cores for optimal utilization of multicore technology. We have divided the computational matrix of dynamic programming in three parts; growing region, stable region and shrinking region depending on whether the number of subproblems increases, remain stable or decreases uniformly phase by phase respectively. We realize the parallel implementations of matrix chain multiplication, longest common subsequence and 0/1 knapsack on Intel Xeon X5650 and E5-2695 using OpenMP with different scheduling policies and adequate chunk sizes. We conclude that, for the growing or the shrinking region of dynamic programming parallelization adopted in this article, guided schedule is better as compared to other scheduling scheme. Static or dynamic schedule is better for the stable region of dynamic programming. Dynamic programming approach, where all three regions are present, we achieve more speedup by applying the mixed scheduling approach rather than applying only single scheduling technique for the entire computations. In LCS, we achieve approximately 20% more speedup using a mixed scheduling technique over the conventional single scheduling approach on Intel Xeon E5-2695.

Keywords: Dynamic programming, multicore, OpenMP.

1. Introduction

Dynamic programming is widely used for discrete and combinatorial optimization problems. Dynamic programming is based on storing all intermediate results in a

Abbreviations	
GPU	Graphics Processing Unit
LCS	Longest Common Subsequence
MCM	Matrix Chain Multiplication

tabular form, so as to utilize it for further computations. Due to its amenable computational approach, this technique has been largely adopted for solving various optimization problems, including matrix chain multiplication, longest common subsequence, binary knapsack, travelling salesman problem and so on. While solving any optimization problem using dynamic programming technique, we get the characteristic equation along with some terminating conditions. Based on the characteristic of this recursive equation, classification of dynamic programming is summarized. If the characteristic equation contains only one recursive term on the right hand side, then dynamic programming is called monadic otherwise it is called polyadic. Formulation of the solution of any optimization problem using dynamic programming can be divided into different phases. The computations belong to a particular phase depend on the previous phases. If it depends only on the computations of immediate previous phase, then this type of dynamic programming is called serial otherwise it is called non-serial. Based on this categorization dynamic programming is divided into 1) Serial monadic 2) Serial polyadic 3) Non-serial monadic 4) Non-serial polyadic.

The main contributions of this research work are listed as follows;

- We present parallel implementations and comparative study of three categories of dynamic programming on multicore using OpenMP for different scheduling techniques with appropriate chunk sizes.
- We have divided the computational matrix of regular dynamic programming in three parts; growing region, stable region and shrinking region and finally depending on the region we select the scheduling policy.
- We show that the mixed scheduling approach is better as compared to single scheduling approach for that kind of dynamic programming where more than one region is present and at least one of them should be stable region.

The remaining paper is organized as follows. Related work and literature about the parallelization of dynamic programming algorithms on multicore and manycore are described in Section 2. Required background and related definitions about dynamic programming algorithms and OpenMP is discussed in section 3. In section 4, parallel implementations and dependence analysis of several categories of dynamic programming is discussed. Division of computational matrix into different regions, different scheduling policies for different regions is proposed. A new scheduling policy i.e. mixed scheduling approach is also proposed in this section. Results of parallelization with different scheduling techniques for three categories of dynamic programming algorithms are presented in section 5. Conclusions and future work are discussed in section 6.

2. Related Work

Dynamic programming is one of the Berkley 13 dwarfs and as there is inherent parallelism in every dynamic programming approach, parallelization and load

balancing of dynamic programming algorithms are widely studied in literature. In [1], the authors present the parallelization of serial monadic dynamic programming algorithms for clusters and network of workstations using message passing interface. Latency tolerant model and percolation techniques for programming on multi-core architectures for parallelizing the most complex category of dynamic programming, i.e., non-serial polyadic dynamic programming is presented in [2]. In [3], the authors study the asynchronous analysis of dynamic programming algorithms and their effects of various delays due to the communication and cache misses.

The computational demand of non-serial polyadic dynamic programming has migrated from the conventional multicore to throughput efficient manycore i.e. on GPUs. In [4 - 6] authors parallelized the MCM using dynamic programming on the GPUs and studied the behaviour of various optimization techniques on the GPUs such as tiling, memory coalescing and matrix realignment. Inherent load imbalance in the non-serial polyadic dynamic programming using Compute Unified Device Architecture and its efficient mapping to processing elements is presented in [7]. The two stage adaptive thread model for efficient mapping of subproblems belonging to a particular phase to the processing cores is illustrated by employing a different number of threads for different phases while solving parallel MCM.

OpenMP uses different work sharing constructs along with different scheduling policies [8, 9]. OpenMP provides efficient synchronization primitives for achieving synchronization among consecutive phases in the parallel implementation of any dynamic programming algorithm. An effort has been presented for efficient mapping of irregular application over multicore using OpenMP in [10]. The authors focused on the parallelization of irregular applications by enforcing the threads distribution in the close proximity of the parallel region. A heuristics for distributing the works of homogeneous parallel dynamic programming on heterogeneous systems is proposed in [11]. A very good classification of dynamic programming algorithms on the basis of table size and dependence of one entry on the other entries is presented in [12]. This gives us a clear understanding of the parallelization of dynamic programming algorithms in terms of phases by considering the total number of entries required for the computation of one phase. Multithreaded implementations of various applications using OpenMP and related issues such as load balancing, threads assignments are presented in [13]. Authors, mainly focused on nested parallelism of various applications using OpenMP and show that load balancing is the key for achieving scalability by demonstrating parallelization of wavelet compression application. A bridging model for achieving portability of parallel applications implemented for multicore system has been presented in [14]. In addition, this model is capable of achieving load balanced parallelization by taking care of availability of number of processors, cache size, synchronization and communication cost. Applications having an inherent load imbalance are currently being targeted for parallelization on multicore and manycore in the recent literature [7], [14]. Parallel dynamic programming on clusters and GPUs and its usage in bioinformatics algorithms are being widely discussed in recent literature [15, 16].

3. Background and Definitions

3.1. Matrix chain multiplication

MCM is a classic example of non-serial polyadic dynamic programming. This problem is also known and widely discussed in literature as an optimal matrix parenthesization problem. Given a sequence of matrices which are compatible to multiplication and as we know, matrix multiplication is associative in nature; sequence of multiplications of matrices greatly affects the final number of scalar multiplications for the actual multiplication.

More specifically, MCM can be defined as: given a chain of n matrices, where the dimensions of the matrix m_i is ($p_{i-1} * p_i$), $1 \le i \le n$. $p[0..n]$ is the dimension vector of size (n+1) that specifies the dimensions of all n matrices. The algorithm finds out the optimal sequence of matrix multiplications with the help of a dynamic programming technique so that the total number of scalar multiplications for the actual multiplication would comes out to be minimum. $m[i, j]$ indicates optimal number of multiplications required for actual multiplication of the sequence from matrix m_i to matrix m_j. In phase t, the difference between i and j is t, i.e. $i + t = j$. The final entry that is to be computed is $m[1, n]$. The recurrence equation of MCM can be expressed as follows:

$$m[i,j] = \begin{cases} 0 & if\,(i = j) \\ \min_{i \le k < j} \{ m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \} & if\,(i < j) \end{cases} \quad (1)$$

There are (n-1) phases in the computation of MCM after initializing $m[i, i] = 0$, $\forall i : 1 \le i \le n$. As we proceed from the computation of phase i to phase (i+1), number of subproblems is decreased by 1 and an amount of computations for one subproblems is increased. The time complexity for calculation of one $m[i,j]$ is $\Theta(j - i)$.

3.2. Longest common subsequence

LCS is an example of non-serial monadic dynamic programming. Formally, we define the LCS as: given a sequence $A = \{a_1, a_2, ..., a_m\}$, another sequence $C = \{c_1, c_2, ..., c_k\}$ is said to be a subsequence of A if there exists a strictly increasing sequence $\{t_1, t_2, ..., t_k\}$ of indices of A such that for all $j = 1, 2, ..., k$, we have $a_{t_j} = c_j$. For example, consider the sequence $A = \{b, c, b, a, d, a, b\}$, $C = \{c, a, a, b\}$ is a subsequence of A with the strictly increasing sequence of indices of A is $\{2, 4, 6, 7\}$. We have given two sequences A and B, another sequence C is said to be a common subsequence if C is a subsequence of A as well as of B. The number of characters in the sequence is called the length of the sequence. Common subsequence C with length k is said to be LCS of the sequence A and sequence B if there is no other common subsequence exists for sequence A and sequence B with length k' such that $k' > k$.

This problem can also be solved with the help of dynamic programming with $O(n^2)$ time complexity when both the considered sequences are of length n. Given two sequences $A = \{a_1, a_2, ..., a_m\}$ and $B = \{b_1, b_2, ..., b_n\}$, $c[i, j]$ indicates the

length of the LCS for the two sequences $\{a_1, a_2, \ldots, a_i\}$ and $\{b_1, b_2, \ldots, b_j\}$. Finally, we have to compute $c[m, n]$. All $c[i, j]$ are stored in the matrix $c[0..m, 0..n]$. Entries of the c matrix are computed row-wise i.e. entries of the first row are computed from left to right, then the second row, and so on. The recurrence relation for solving the LCS problem using dynamic programming is defined as follows:

$$c[i,j] = \begin{Bmatrix} 0 & if\,(i=0) or (j=0) \\ c[i-1,j-1]+1 & if\,(i,j>0) and (a_i = b_j) \\ \max(c[i-1,j], c[i,j-1]) & if\,(i,j>0) and (a_i \neq b_j) \end{Bmatrix} \quad (2)$$

3.3. Binary knapsack

Binary knapsack or 0/1 knapsack problem is also a classic example that can be solved using dynamic programming. Given n objects, each associated with unique weight and profit, one knapsack with capacity C is with us, we have to select some of the objects within the capacity limit of knapsack so that the profit should be maximized. More formally, p_j and w_j are the profit and weight respectively of j^{th} object, $1<=j<=n$. x_j is set to 0 or 1, 0 if j^{th} object is not included in the knapsack, 1 if j^{th} object is included in the knapsack. For a non trivial solution of 0/1 knapsack $\sum_{j=1}^{n} w_j > C$. The algorithm selects objects from the list of given objects so that $\sum_{j=1}^{n} p_j x_j$ should be maximum and $\sum_{j=1}^{n} w_j x_j <= C$. Let $t[i, j]$ be the maximum profit while considering first i objects with the considered knapsack capacity j. $t[0..i, 0..j]$ is computed in row major order and each row is computed from left to right. The recurrence relation for solving the binary knapsack problem using dynamic programming is defined as follows:

$$t[i,j] = \begin{Bmatrix} 0 & if\,(i=0) or (j=0) \\ t[i-1,j] & if\,(j<w_i) \\ \max(p_i + t[i-1][j-w_i], t[i-1,j]) & if\,(i>0) and (j \geq w_i) \end{Bmatrix} \quad (3)$$

We can easily understand from the above recurrence that either if no object is taken into consideration or if considered knapsack capacity is zero, no profit can be gained. We can add the current object in the knapsack only if considered knapsack capacity is not less than the weight of the current object.

3.4. OpenMP

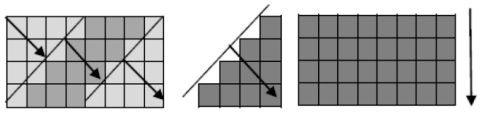
OpenMP is one of the favorite Application Programming Interface used for parallelization on the shared memory architecture, adopted by a majority of high performance community due to its higher programming efficiency. OpenMP is shared memory programming fork join model that provides various directives and library functions for creating and managing a team of threads. Various synchronization and work sharing constructs are provided by OpenMP, using which we automatically or manually divide the task among threads.

OpenMP provides four different types of scheduling for assigning the loop iterations to different threads: static, dynamic, guided and runtime. Schedule clause is provided for specifying schedule and numbers of iterations i.e. chunk size. In static scheduling, chunks are assigned to processing cores in round robin fashion. It is the simplest kind of scheduling with minimum overhead. In dynamic scheduling, thread requests for new chunk as it finishes the assigned chunk. In the guided scheduling thread request for newer chunks, but chunk size is calculated as the number of unassigned iterations divided by the total number of threads in the team. Guided scheduling seems to be more efficient scheduling, but involves a little bit of overheads in the calculation of chunk size. In runtime scheduling, schedule and optional chunk size are set with the help of environment variables. The details of scheduling techniques are discussed in [8, 9].

4. Parallel Implementation

Computations of any dynamic programming formulation can be divided into different phases. Sub-problems belong to a particular phase can be computed in parallel. We broadly classify the DP formulation in two fashions. If the entire computations of a dynamic programming formulation can be accommodated easily and uniformly in a matrix, then we call this as a regular dynamic programming otherwise it is treated as irregular dynamic programming. Examples of regular dynamic programming are LCS, 0/1 knapsack and MCM. Examples of irregular dynamic programming are the single source shortest path, multistage graph and travelling salesman problem. For example, solving single source shortest path problem using dynamic programming, subproblems of a particular phase depends on the number of incoming edges towards that particular node [17]. Generally graph problems fall under the category of irregular dynamic programming. Here, in our study, we focus on the study of parallelization efforts of regular dynamic programming on multicore.

We have divided the entire computation of regular dynamic programming in three parts based on the number of subproblems in each phase: 1) growing region: number of subproblems increases uniformly phase by phase, 2) stable region: number of subproblems are fixed in each phase, and 3) shrinking region: number of subproblems decreases uniformly phase by phase. In MCM computations, only shrinking region is present. In LCS, phases are considered in an anti-diagonal fashion. In LCS, first growing region, then stable region and finally shrinking region, all three regions are present. In 0/1 knapsack, the parallel computations proceed row wise. In 0/1 knapsack, neither growing region nor shrinking region is present, only the stable region is present. Fig. **1** represents the region wise partition and arrows indicate direction of parallelization strategies for the LCS, MCM and 0/1 knapsack.



(a) LCS (b) MCM (c) 0/1 knapsack

Fig. 1. Dependence and parallelization strategies for three categories of dynamic programming algorithms.

For each phase, the numbers of subproblems are assigned to the threads which are handled by the chunk size parameter in OpenMP and finally threads execute those assigned subproblems over physical cores which are handled by a scheduling policy in OpenMP. While executing a subproblem over physical core by a thread, data required for calculation play an important role in the efficiency of dynamic programming algorithms. Finally, it leads to an optimization sort of problem. **Table 1** represents the different characteristics of three dynamic programming algorithms in the context of parallel processing of subproblems of a specific phase and amount of computations belonging to that phase. We summarize the factors affecting the efficiency of regular dynamic programming algorithms as follows:

1. Number of subproblems in each phase
2. The amount of the computations of a subproblem belonging to a particular phase
3. Feasibility of dynamic adjustment of number of threads for different phases and its consequences in terms of parallelization overheads
4. Scheduling policy
5. Consequences of disjointness of data required for calculations of different subproblems belonging to a particular phase

Table 1. Characteristics of three categories of dynamic programming.

	Number of sub problems in subsequent phases	**Amount of computations for a subproblem as the computation proceeds to the next phase**	**Serial/ Non-serial**	**Monadic/ Polyadic**	**Nature of data requirements of subproblems of a particular phase**
MCM	Decreasing	Monotonically increasing	Non-serial	Polyadic	Completely disjoint
LCS	Increasing, fixed followed by decreasing	fixed	Non-serial	Monadic	Partially overlapping for consecutive subproblems
0/1 Knapsack	Fixed	Fixed	Serial	Monadic	May be disjoint

5. Results

We have evaluated the performance of MCM (shrinking region), LCS (growing, stable region followed by shrinking region) and 0/1 knapsack (stable region) on Intel Xeon X5650 Quad Core processor with CPU clock 2.67 GHz, 12 CPU cores, 4 GB of RAM and Intel Xeon E5-2695 with CPU clock 2.3GHz, 28 CPU cores, 32 GB of memory. The operating system used for performance evaluation is openSUSE 13.1 64-bit Linux with GNU GCC compiler 4.8.3 with OpenMP

3.1. Speedup is computed for all three categories of dynamic programming algorithms as the ratio of time taken by sequential algorithm to time taken by parallel algorithm. As all the speedups are greater than one, OpenMP performs better as compared to sequential algorithms.

Fig. **2** and 3 represent graphical representations of the effect of parallelization efforts with different number of matrices for various scheduling techniques for MCM. We have applied different scheduling techniques for different numbers of matrices. Since the parallel computational matrix is triangular, the number of subproblems decreases by one in the subsequent phase; we have observed that guided scheduling performs much better as compared to other scheduling schemes for a large number of matrices.

Fig. **4** and Fig**. 5** show pictorial representations of speedup with different lengths of strings for different scheduling policies for LCS algorithm. The *y*-axis represents speedup and the *x*-axis represents the length of the first string and the length of the second string is twice the length of the first string. As Fig. 1 indicates, the computational matrix have growing and shrinking region of the same size. Central computation, i.e. stable region of the LCS having the same number of phases as the length of the first string. In parallel computation of LCS, guided scheduling is applied for growing and shrinking regions and static scheduling is applied for the central computation i.e. stable region. This method of computation of applying different scheduling schemes named as a mixed scheduling. We have observed that the mixed scheduling performs better as compared to other scheduling approaches in the parallel computation of LCS. We achieve speedup of 18x by applying mixed scheduling approach in comparison with other scheduling approaches on Intel Xeon E5-2695.

Fig. **6** and Fig**. 7** represent the relation between speedup and the number of items for 0/1 knapsack problem for different scheduling schemes. In parallel computation of 0/1 knapsack, phases are considered row-wise. The number of subproblems is fixed in each phase. Therefore, the entire computations of parallel 0/1 knapsack comes only under the stable region. After experimentation and analysis, we found that static scheduling performs better as compared to other scheduling for stable region.

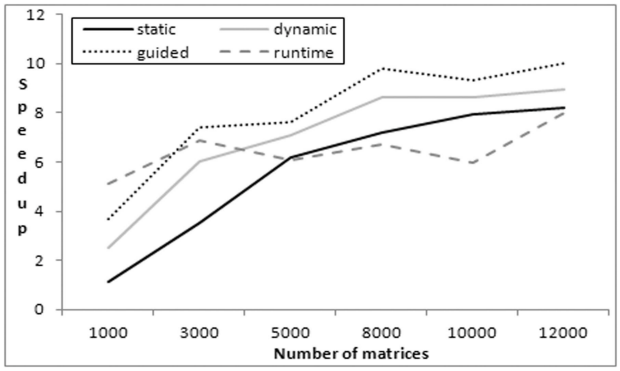


Fig. 2. Comparison of speedup with different numbers of matrices for different scheduling techniques for MCM on Intel Xeon X5650.

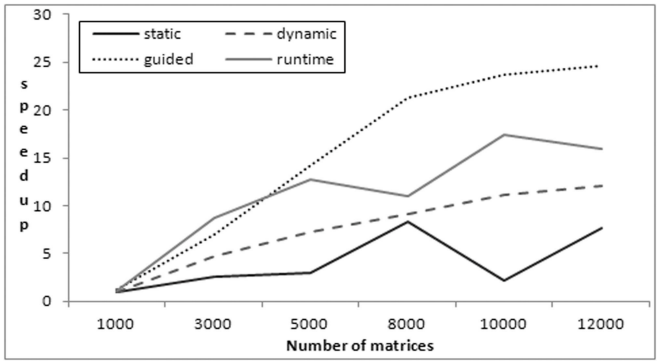


Fig. 3. Comparison of speedup with different numbers of matrices for different scheduling techniques for MCM on Intel Xeon E5-2695.

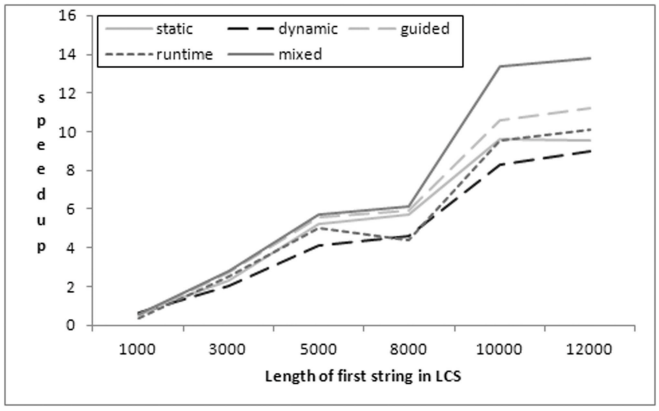


Fig. 4. Comparison of speedup with different lengths of strings for different scheduling techniques for LCS on Intel Xeon X5650.

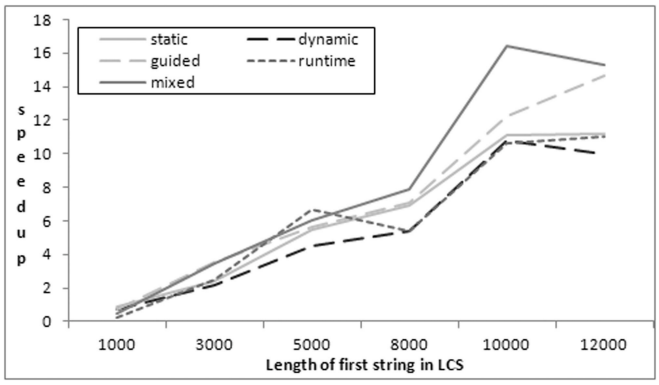


Fig. 5. Comparison of speedup with different lengths of strings for different scheduling techniques for LCS on Intel Xeon E5-2695.

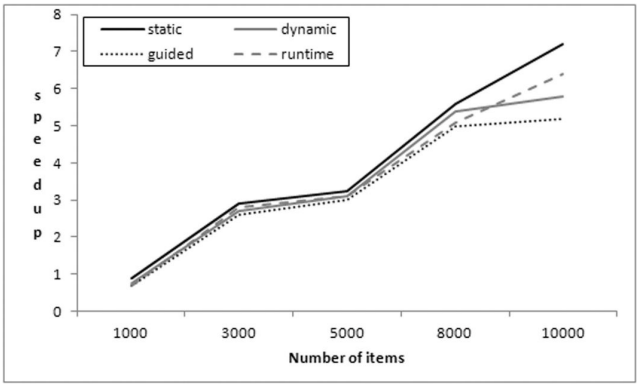


Fig. 6. Comparison of speedup with different numbers of items for different scheduling techniques for 0/1 knapsack on Intel Xeon X5650.

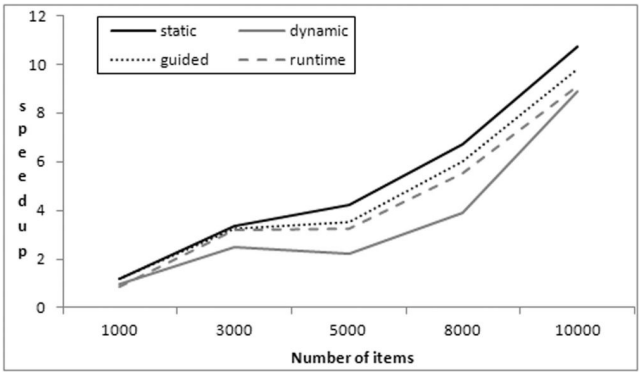


Fig. 7. Comparison of speedup with different numbers of items for different scheduling techniques for 0/1 knapsack on Intel Xeon E5-2695.

We also notice that, in MCM parallelization, we achieved more speedup on Intel Xeon E5-2695 as compared to Intel Xeon X5650 for the same number of threads i.e. 12 threads for large number of matrices because the calculation of subproblems belonging to a particular phase in MCM requires all the previously computed data and data is completely disjoint for the computations of two subproblems belonging to a particular phase. On Intel Xeon E5-2695, we have sufficient memory to accommodate all the previously computed data for the computation of a particular phase, which is a limitation on the Intel Xeon X5650.

We can also note that deadlock and race will not arise in the parallelization of these three categories of dynamic programming algorithms such as MCM, LCS and 0/1 knapsack because we apply parallelization phase by phase. Entries of a particular phase are computed in parallel. The subproblems of a particular phase cannot be computed unless all the subproblems of the immediate previous phase are computed. Neither more than one thread access the same memory location for writing nor two threads wait for each other indefinitely in the proposed approach of parallel computations.

6. Conclusion and Future Work

We conclude from Fig. 2 and 3 that for shrinking region, guided scheduling compensates the calculation overhead of chunk sizes and dynamic allocation of chunks to processing cores. Though static scheduling takes decisions on the allocation of subproblems to the processing cores at compile time itself, it fails to remain consistent in the performance due to the inherent non-uniformity in the shrinking region present in MCM. In MCM only shrinking region is present, due to that it performs better using guided scheduling. Guided scheduling is better for that category of dynamic programming where only growing or shrinking region is present. In other words, we can say that guided scheduling is not suitable for stable region.

In 0/1 knapsack only stable region is present. We observe from Fig. 6 andFig. **7**, due to the uniform load distribution in each phase of 0/1 knapsack, static scheduling performs well as compared to other scheduling schemes because of compile time decision of allocation of chunks of iterations to processing cores in round robin fashion. Static scheduling is better for the stable region of any dynamic programming algorithms. Dynamic scheduling should also be selected for a stable region of dynamic programming when processing cores are not uniformly loaded.

Total computational time for the calculation of score matrix of the LCS algorithm can be divided into three parts; the time taken for the first part, i.e. growing region, time taken for the second part, i.e. stable region and time taken for third part, i.e. shrinking region. First and third parts are amenable for guided scheduling, whereas static/dynamic scheduling is suitable for second part. In mixed approach, we apply guided in the first part, static for the second part and once again guided for the third part. With reference to Fig. 4 and Fig**. 5**, the reason why guided is performing better as compared to static scheduling is that, $2/3^{rd}$ portion of computational matrix is suitable for guided scheduling.

The generalized conclusion can be drawn as follows: Parallel dynamic programming which satisfies two conditions; 1) Number of subproblems in different phases are not same and 2) More than one region should be there in a computational matrix with atleast one region be the stable region, mixed scheduling approach with appropriate chunk size performs better as compared to single conventional scheduling approach.

We can extend these comparisons of parallel dynamic programming on GPUs. Various CPU and GPU optimizations can also be studied in this series of work in the context of regular dynamic programming.

References

1. Canto, S.D.; Madrid, A.P.; and Bencomo, S.D. (2005). Parallel dynamic programming on clusters of workstations. *IEEE Transaction on Parallel and Distributed Systems*, 16(9), 785-798.
2. Tan, G.; Sun, N.; and Rao, G.R. (2009). Improving Performance of Dynamic Programming via Parallelism and Locality on Multicore Architectures. *IEEE Transaction on Parallel and Distributed Systems*, 20(2), 261-274.

3. Lewandowski, G.; Condon, A.; and Bach, E. (1996). Asynchronous Analysis of Paralle1 Dynamic Programming Algorithms. *IEEE Transaction on Parallel and Distributed Systems*, 7(4), 425-438.
4. Dash, T.; and Nayak, T. (2012). Chain Multiplication of Dense Matrices: Proposing a Shared Memory based Parallel Algorithm. *International Journal of Computer Applications*, 8(1), 11-16.
5. Xiao, S.; Aji, A.M.; and Feng, W.C. (2009). On the Robust Mapping of Dynamic Programming onto a Graphics Processing Unit. *Proceedings of 15th International Conference on Parallel and Distributed Systems (ICPADS)*, 26-33.
6. Nishida, K.; Ito, Y.; and Nakano, K. (2011). Accelerating the Dynamic programming for Matrix Chain Product on the GPU. *Proceedings of 2nd International Conference on Networking and Computing (ICNC)*, 320-326.
7. Wu, C.C.; Ke, J.Y.; Lin, H.; and Feng, W.C. (2011). Optimizing Dynamic Programming on Graphics Processing Units via Adaptive Thread-Level Parallelism. *Proceedings of 17th International Conference on Parallel and Distributed Systems (ICPADS)*, 96-103.
8. OpenMP specifications. Retrieved May 7, 2015, from http://www.openmp.org/specs/.
9. Chapman, B.; Jost, G.; and Van Der Pas, R. (2007). *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press.
10. Broquedis, F.; Diakhaté, F.; Thibault, S.; Aumage, O.; Namyst, R.; and Wacrenier, P.A. (2008). Scheduling Dynamic OpenMP Applications over Multicore Architectures. *Proceedings of OpenMP in a New Era of Parallelism, Lecture Notes in Computer Science*, 5004, 170-180.
11. Cuenca, J.; Gimenez, D.; and Martinez, J.P. (2005). Heuristics for Work Distribution of a Homogeneous Parallel Dynamic Programming Scheme on Heterogeneous Systems. *Parallel Computing*, 31(7), 711-735.
12. Galil, Z.; and Park, K. (1992). Dynamic programming with convexity, concavity and sparsity. *Theoretical Computer Science*, 92(1), 49-76.
13. Blikberg, R.; and Sorevik, T. (2005). Load Balancing and OpenMP implementation of nested Parallelism. *Parallel Computing*, 31(10-12), 984-998.
14. Valiant, L.G. (2011). A bridging model for multi-core computing. *Journal of Computer and System Sciences*, 77(1), 154-166.
15. Tang, S.; Yu, C.; Sun, J.; Lee, B.S.; Zhang, T.; Xu, Z.; and Wu, H. (2011). EasyPDP: An Efficient Parallel Dynamic Programming Runtime System for Computational Biology. *IEEE Transaction on Parallel and Distributed Systems*, 23(5), 862-872.
16. Chowdhury, R.A.; Le, H.S.; and Ramachandran, V. (2008). Cache-oblivious Dynamic Programming for Bioinformatics. *IEEE Transactions On Computational Biology and Bioinformatic*, 7(3), 495-510.
17. Cormen, T.H.; Leiserson, C.E.; Rivest, R.L.; and Stein, C. (2008). *Introduction to Algorithms* (2nd ed.). PHI Learning Private Limited.