EasyHPS: A Multilevel Hybrid Parallel System for Dynamic Programming

Jun Du, Ce Yu*, Jizhou Sun, Chao Sun
School of Computer Science and Technology
Tianjin University
China
Email: {tjudujun, yuce, jzsun}@tju.edu.cn

Shanjiang Tang
School of Computer Engineering
Nanyang Technological University
Singapore
Email: stang5@ntu.edu.sg

Yanlong Yin
Department of Computer Science
Illinois Institute of Technology
Chicago, IL 60616
Email: yyin2@iit.edu

***Abstract*—Dynamic programming approach solves complex problems efficiently by breaking them down into simpler subproblems, and is widely utilized in scientific computing. With the increasing data volume of scientific applications and development of multi-core/multi-processor hardware technologies, it is necessary to develop efficient techniques for parallelizing dynamic programming algorithms, particularly in multilevel computing environment. The intrinsically strong data dependency of dynamic programming also makes it difficult and error-prone for the programmer to write a correct and efficient parallel program. In order to make the parallel programming easier and more efficient, we have developed a multilevel hybrid parallel runtime system for dynamic programming named EasyHPS based on the Directed Acyclic Graph(DAG) Data Driven Model in this paper. The EasyHPS system encapsulates details of parallelization implementation, such as task scheduling and message passing, and provides easy API for users to reduce the complexity of parallel programming parallelization. In the DAG Data Driven Model, the entire application is initially partitioned by data partitioning into sub-tasks that each sub-task processing a data block. Then all sub-tasks are modeled as a DAG, in which each vertex represents a sub-task and each edge indicates the communication dependency between the two sub-tasks. In task scheduling, a dynamic approach is proposed based on DAG Data Driven Model to achieve load balancing. Data partitioning and task scheduling are both done on processor-level and thread-level in the multilevel computing environment. In addition, experimental results demonstrate that the proposed dynamic scheduling approach in EasyHPS is more efficient in comparison with those static ones such as block-cyclic based wavefront.**

***Keywords*-dynamic programming; EasyHPS; multilevel computing environment; Directed Acyclic Graph Data Driven Model; task scheduling; load balancing**

I. Introduction

Dynamic programming (DP) is a popular algorithm-designing technique for resolving many decision and optimization problems. It is a method for solving complex problems by breaking them down into simpler subproblems and then solving them one by one. Dynamic programming has been widely applied in many scientific applications such as computational biology, mathematics, computer science, and economics. Typical applications include RNA and protein structure prediction[1], genome sequence alignment[2], context-free grammar recognition[3], string editing, optimal static search tree construction[4] and so on. With the data volume increasing, the computation time cost becomes too high and impractical. Therefore, DP problem parallelization has become very crucial and necessary.

In parallel programming, task partition, scheduling and data synchronization among the different sub-tasks are typically some of the greatest obstacles to getting good parallel program performance. With the rapid development of multi-core and multi-processor hardware technologies, each node in a cluster also forms a shared-memory parallel computing environment, so the cluster can be regarded as a multilevel computing environment. On such multilevel parallel architectures, parallel computing can be more efficient than before, but parallel programming is even harder than traditional single level parallel programming.

This paper presents a runtime system EasyHPS based on Directed Acyclic Graph(DAG) Data Driven Model for dynamic programming parallelization in multilevel computing environment. EasyHPS is deployed in master-slave mode. To make parallelization more efficient in multilevel computing environment, EasyHPS divides the computing resources(processors, cores) into several groups. All the computing resources in the same group are on the same multi-processor environment and share their memories. The parallelization on processor-level works among different groups and the parallelization on thread-level works in a single group. For each level, the EasyHPS runtime system automatically handles process/thread creation, communication and management. It also distributes sub-tasks and data dynamically to worker processes and worker threads hierarchically to ensure the load balance by DAG Data Driven Model. EasyHPS runtime system provides APIs for users to implement their specific application in order to make the dynamic programming parallelization become easier. In this paper, we mainly discuss the architecture of EasyHPS and compare it with other parallel design patterns by their experimental results.

The remainder of this paper is organized as follows. Section II lists the related work on parallelization techniques of dynamic programming algorithms. Section III gives an introduction to framework of EasyHPS. Section IV describes DAG Data Driven Model of EasyHPS. Section V presents task scheduling and fault tolerance based on DAG Data Driven

*Ce Yu(yuce@tju.edu.cn): the corresponding author.

Model in EasyHPS. Section VI presents experimental results of EasyHPS and compares them with other DP parallelization methods to verify the advantages of EasyHPS. Section VII concludes the paper and discusses the future work.

II. Related Work

As an efficient algorithm-designing technique, DP has been widely applied in many scientific applications such as computational biology and production scheduling. Though the amount of scientific data is burgeoning, DP computation cost is still too high. So it is necessary and meaningful to parallelize it. Lots of work has been done to exploit the parallelization of DP algorithms.

For shared memory systems, Edmonds et al.[5] and Galil[6] described several parallel algorithms on general shared memory multiprocessor systems. Bradford[7] presented several algorithms that solve optimal matrix chain multiplication parenthesization using the CREW PRAM model. Tan[8][9] focused on a specific type of non-serial polybasic DP with triangular matrix, for which he introduced some optimization algorithms and theoretical models on multi-core architectures. But all of these works cannot support large-scale parallelization because of the poor scalability of share memory systems.

For the distributed memory multi-processor system, Almeda et al.[10] presented a parallel implementation tiling on a ring processor, whereas this parallel tiling algorithm cannot achieve load balance. Zhou[11] proposed a parallel out-of-core algorithm based on the conventional out-of-core model. In[12][13], the authors presented a static parallel scheduling strategy named block-cyclic based wavefront method and did some work including giving some algorithms, making a pattern-based prototype system afterward. All of these works did not consider the architecture of cluster and parallelize the application on a single level which is not efficient enough in multilevel computing environment.

We have published open source EasyPDP[14] system based on DAG Data Driven Model for DP applications running on shared memory environment. EasyPDP is a runtime system which aims at parallelizing dynamic programming algorithms. Under the concept of software reusability and complexity reduction of parallel programming, a DAG Data Driven Model is proposed in EasyPDP, which supports those applications with a strong data interdependence relationship. In order to support larger data scale computing, we develop EasyHPS system which is deployed in multilevel computing environment. Compared with previous work, EasyHPS has two obvious advantages. The first one is realizing parallelization for DP algorithms in multilevel computing environment, which is more theoretically efficient and can support larger scale computing. The second one is reducing parallel programming implementation complexities in multilevel computing environment.

III. Framework of EasyHPS

EasyHPS is designed in master-slave mode. The framework of EasyHPS is shown in Figure 1. Master part is responsible for parallelization on processor-level; slave part is responsible for parallelization on thread-level. While doing a DP problem parallelization based on EasyHPS, the whole task of the application should be divided into some sub-tasks in master part firstly. Then each sub-task will be sent to the slave part. We call it the parallelization on processor-level. The platform of slave part is an individual computing node, which is usually a multi-core/multi-processor computing environment. Each sub-task will be computed in parallel in a slave computing node, which is the parallelization on thread-level. In EasyHPS runtime environment, there will be one master node for sub-task scheduling on processor-level and several slave nodes for computing. Master node is deployed as the master part and slave nodes are deployed as the slave part. All the master part and slave parts in EasyHPS runtime system have two main components, DAG Data Driven Model and scheduler. We define the DAG Data Driven Model in master part as master DAG Data Driven Model and the DAG Data Model in slave part as slave DAG Data Driven Model. Similarly, we also have the master scheduler and the slave scheduler.

DAG Data Driven Model mainly describes the data dependency of DP problem, which is used in task partition and scheduling. The master/slave scheduler is responsible for task scheduling and load balancing in EasyHPS. Master part and slave parts start at the same time when a DP problem parallelization is scheduled in the EasyHPS. The general work flow can be described as follows:

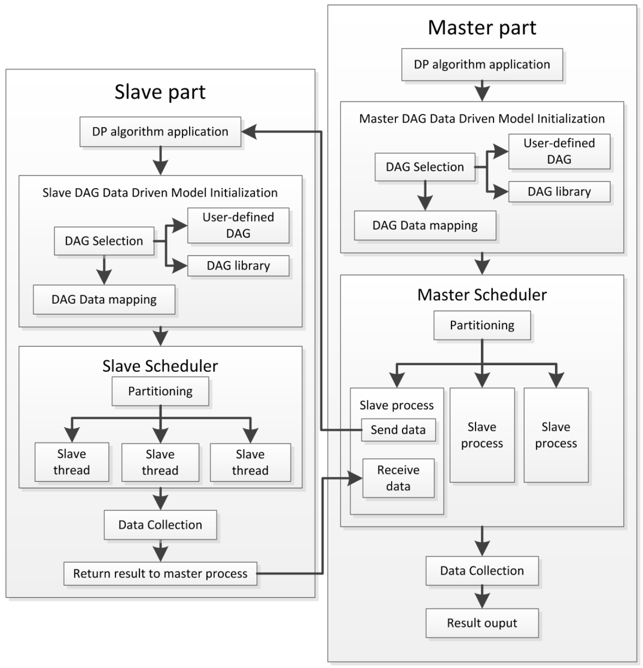


Fig. 1: Framework of EasyHPS

a. Master part initializes the master DAG Data Driven Model for the whole computing task based on the DP algorithm application.
b. After master DAG Data Driven Model initialization finishes, the master scheduler does the task partition based

on the master DAG Data Driven Model.

c. The master scheduler distributes all the computable sub-tasks among all slave parts.
d. For each slave part, it gets computable sub-task from master part and initializes the slave DAG Data Driven Model for the sub-task based on the DP algorithm application.
e. The slave scheduler does the sub-task partition based on the slave DAG Data Driven Model.
f. The slave scheduler distributes sub-sub-task (Task which is the sub-task of the sub-task sent by master part.) among slave threads in slave part.
g. Slave threads execute the computation in parallel.
h. Slave part collects computed data from threads and returns them to master part.
i. Master part collects computed intermediate results from all the slave parts and output the final result.

IV. DAG Data Driven Model

DAG Data Driven Model is an important component in EasyHPS. In the DAG Data Driven Model, a DAG Pattern Model is defined to describe the sub-tasks and their communication dependencies for DP algorithms. User can select the DAG Pattern Model provided by DAG Pattern Model library or define specific DAG Pattern Model by user APIs. User APIs can also help user initialize the sub-task partition size and implement data mapping function for data blocks mapping to the vertices of DAG Pattern Model.

A. DAG Pattern Model Definition

A DAG Pattern Model is denoted as $D = \{V, E\}$, where $V = \{V_1, V_2 \cdots V_n\}$ is a set of n vertices; E is a set of unidirectional edges, as shown in Figure 2. DAG has no circles.

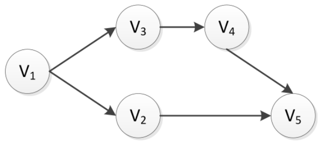


Fig. 2: A example DAG Pattern Model

DAG Pattern Model is the major component of DAG Data Driven Model. In the DAG, each vertex describes a sub-task and each unidirectional edge which starts from one vertex to the other represents the communication relationship and the precedence constraints among sub-tasks. If there is a unidirectional edge E_{12} from vertex V_1 to vertex V_2, we call V_1 is the precursor of V_2 and V_2 is the successor of V_1. The unidirectional edge E_{12} shows that V_2 can start computing only after V_1 is completed. Each vertex in the DAG has some precursors and successors. After task partition, each DP problem can be mapped with a corresponding DAG Pattern Model.

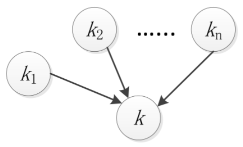


Fig. 3: Optimal solution to problem k represented by DAG

B. Relationships between DP problem and DAG Pattern Model

There are two key attributes that a problem must have in order to benefit from dynamic programming: optimal substructure and overlapping subproblems.

Optimal substructure means that the solution to a given optimization problem can be obtained by the combination of optimal solutions to its subproblems. DP problems are solved by decomposing the problem into a set of interdependent subproblems, and using their results to solve larger subproblems until the entire problem is solved. If $S(k)$ represents the solution to DP problem k, $S(k)$ can be written as a formulation $S(k) = g(S(k_1), S(k_2) \cdots S(k_n))$, where g is called the composition function and k_n is the subproblem of k. This formulation is said to be a DP formulation[15]. Then we can use a DAG Pattern Model to describe a DP formulation, which is shown in Figure 3. Overlapping subproblems means that the space of subproblems must be small, that is, any recursive algorithm solving the problem should solve the same subproblems over and over, rather than generating new subproblems. Just like the DP problem k described above, all the subproblems of k can be solved in the same way, which means that all the non-atomic subproblems k_n can be further represented by a DAG as shown in Figure 4.

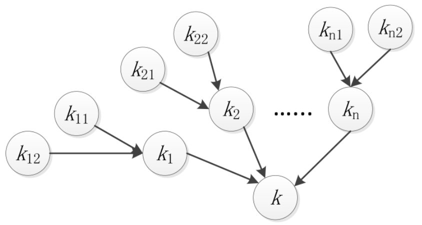


Fig. 4: The entire solution to problem k represented by DAG Pattern Model.

As described above, we can get a corresponding DP formulation for each DP problem to describe its optimal substructure and build a DAG Pattern Model based on the DP formulation. In other words, we can build a corresponding DAG Pattern Model for each DP problem. For example, Nussinov[16] algorithm is a classical DP algorithm. The formulation of Nussinov Algorithm can be listed as follows:

$$F[i,j] = max \begin{cases} F[i,j-1] \\ F[i,k-1] + F[k+1,j-1] + 1 & i \le k \le j-2 \end{cases}$$

where $F[i,0]$ and $F[0,j]$ have been initialized.

Correspondingly, the DAG Pattern Model of Nussinov algorithm can be built based on the above formulation, as shown in Figure 5.

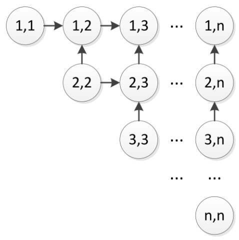


Fig. 5: DAG Pattern Model for Nussinov algorithm

C. DP problem classification and DAG Pattern Model Library

DP algorithms can be classified according to the matrix size and the cell's data dependencies[6]: a DP algorithm for a problem of size n is called a tD/eD algorithm if its matrix size is $O(n^t)$ and each matrix cell depends on $O(n^e)$ other cells. The DP formulation of a problem always yields an obvious algorithm whose time complexity is determined by the matrix size and the dependency relationship. It takes $O(n^{t+e})$ of time to solve a tD/eD problem, provided that the computation of each term takes constant time. Three examples are given in Algorithm 4.1 to 4.3.

Algorithm 4.1: $(2D/0D)$: Given $D[i, 0]$ and $D[0, j]$ for $1 \leq i, j \leq n$,

$$D[i, j] = min\{D[i-1, j]+x_i, D[i, j-1]+y_j, D[i-1, j-1]+z_{i,j}\}$$

where x_i,y_j and $z_{i,j}$ are computed in constant time.

Algorithm 4.2: $(2D/1D)$: Given $w(i, j)$ for $1 \leq i, j \leq n$;$D[i, i] = 0$ for $1 \leq i \leq n$

$$D[i, j] = w(i, j) + min_{i \leq k \leq j}\{D[i, k-1] + D[k, j]\}$$

for $1 \leq i, j \leq n$

Algorithm 4.3: $(2D/2D)$: Given $w(i, j)$ for $1 \leq i, j \leq 2n$; $D[i, 0]$ and $D[0, j]$ for $0 \leq i, j \leq n$,

$$D[i, j] = min_{0 \leq j' < j, 0 \leq i' < i}\{D[i', j'] + w(i' + j', i + j)\}$$

for $1 \leq i, j \leq n$

To reduce the complexity of parallel programming, we have implemented several frequently used DAG Pattern Models which are organized by a DAG Pattern Model library in EasyHPS. Besides the system provided patterns in the library, there is also another kind of patterns called user-defined patterns, which are defined and added to the library by the programmer. For some special DP problems, DAG Pattern Model in the library cannot describe its data dependency; programmers should define and implement the DAG Pattern Model by themselves.

TABLE I: User APIs of DAG Data Driven Model

Data structure description in DAG Data Driven Model		
DAGElements/description for DAG vertex		
Data Member Name	Data Type	Corresponding Description
pre_cnt	int	prefix degree
pos_cnt	int	postfix degree
data_pre_cnt	int	prefix degree of data dependency
posfix_id	pointer to int	linked list of postfix vertices
data_prefix_id	pointer to int	linked list of data dependency vertices
process	pointer to function	a pointer which points to the task function for DAG vertex
dag_pattern/decription of DAG Pattern Model		
dag_pattern_element	pointer to DAGElement	linked list of DAG vertices
dag_size	SizeT(row,col)	the size of DAG
partition_size	SizeT(row,col)	sub-task size after task partition
rect_size	SizeT(row,col)	the size of abstract DAG after task partition
dag_pos	PosT(x,y)	position of upper left DAG
dag_pattern_type	enum DAG_pattern_type	enum DAG type
data_mapping_function	pointer to function	mapping computed data to DAG Pattern Model

D. DAG Data Driven Model initialization with User APIs

The DAG Data Driven Model gets initialized at the beginning of DP problem parallelization. Firstly, according to the specific application, programmers should get corresponding DAG Pattern Model firstly. The DAG Pattern Model can be selected from the DAG Pattern Model library or defined by programmers. Programmers define the DAG Pattern Model by setting correct information for *dag_pattern* data structure which presents DAG Pattern Model. After identifying DAG Pattern Model, programmer sets the parameters of DAG Pattern Model, e.g. *dag_size*, *partition_size* and *data_mapping_function*, as shown in TABLE I. *dag_size* is determined by data scale. *partition_size* presents the size of sub-tasks. Because EasyHPS is a runtime system that parallelizes DP problems in multilevel parallel environment, there are two kinds of *partition_size*: *process_partition_size* which describes partition size of sub-task divided in process level and *thread_partition_size* which describes partition size of sub-task divided in thread level. *data_mapping_function* helps programmer create a rule which describes the relationship between DAG nodes and sub-tasks with its data. The EasyHPS system can set the mapping function automatically if a built-in DAG Pattern Model fits the problem. For the user-defined DAG Pattern Model, the programmer should also design the data mapping function by implementing the user APIs. Other data member in DAG Data Driven Model structure will be set automatically during DAG Data Driven Model initialization.

Task partition will be done during DAG Data Driven Model initialization. Smith-Waterman[17] General Gap algorithm is a good example to describe the task partition processing, as shown in Figure 6 and Figure 7. After setting DAG Data Driven Model parameters, we can get enough information for task partition. Figure 6(a), the DAG Pattern Model shows the original DAG Pattern Model. In Figure 6(b), EasyHPS divides the entire task into some separated groups. For each group, DAG nodes construct a smaller DAG which represents data dependency in sub-task. After partition, original DAG Pattern Model can be also abstracted as a higher-level DAG Pattern Model, as show in Figure 6(c). Task scheduling on processor-level/thread-level is based on the abstract DAG Pattern Model.

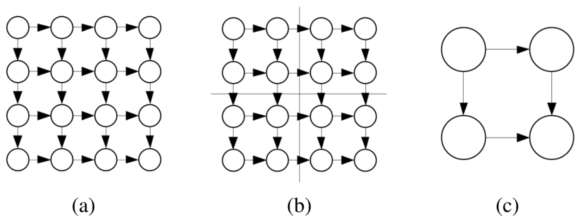


Fig. 6: (a) original DAG before partition (b) DAG pattern partitioned by settings (c) abstract DAG pattern after partition that each vertex represents a sub-task

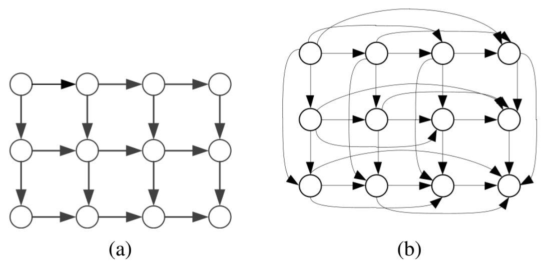


Fig. 7: (a) DAG data driven model in topological level (b) DAG data driven model in data communication level

DAG Data Driven Model has two levels which can be described as topological level and data communication level. Topological level is used to describe the sub-task topological order, which is responsible for parsing and task scheduling, as is shown in Figure 7(a). Data communication level is used to describe data dependencies among sub-tasks, which is responsible for data synchronization, as is shown in Figure 7(b). Data synchronization is done in two stages, before and after sub-tasks executing. Before the sub-task begins executing, master part should send the necessary data to the corresponding slave part which executes the sub-task. After sub-task finishes executing, slave part returns master part the calculation data. Both task scheduling and data synchronization utilizes DAG Data Driven Model which thus has an impact on the efficiency of EasyHPS runtime system.

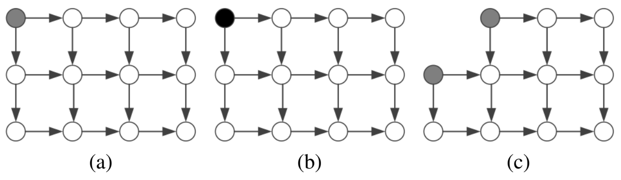


Fig. 8: (a) a sub-task is computable (b) sub-task is computing (c) sub-task which is finished is removed with its connecting edges

E. DAG Parsing in runtime system

The DAG parsing operation aims at discovering current new computable sub-tasks for task scheduling. Each computable sub-task in the DAG Pattern Model is represented by the vertex which has no precursors. For example, the grey vertex in DAG shown in Figure 8(a) represents a computable task and the black vertex in DAG shown in Figure 8(b) represents a finished task. Parsing DAG Pattern Model is a process of topological sorting in DAG. While parsing, master/slave scheduler gets new computable sub-tasks and allocates them among slave nodes/threads for parallelization. The vertices and the connecting unidirectional edges will be removed one by one after corresponding computable sub-task finished, as is shown in Figure 8(c). Parsing process will not be ended until all the vertices and edges in the DAG Pattern Model have been removed.

V. Sub-task scheduling and fault tolerance in EasyHPS

After DAG Data Driven Model initialization finishes, computable sub-tasks can be produced by parsing the DAG Pattern Model. Computable sub-tasks are distributed to different nodes/threads by scheduler of EasyHPS for parallelization. To ensure the load balance among different processors, EasyHPS runtime system adopts dynamic worker pools which use dynamic task allocation and scheduling strategy. EasyHPS runtime system adopts master-slave mode in multilevel computing environment and dynamic worker pools can be described as master worker pool and slave worker pool. Master worker pool is used to do the task scheduling among different processor groups or multi-processor computing nodes and slave worker pool is used to do the task scheduling among different processors in a same group or multi-core computing node.

Fault tolerance and recovery is a very important issue in distributing computing. EasyHPS detects faults by timeout checking. If a task is not completed within a reasonable duration, we assume that there is a failure in the process of the task computing. Faults caused by incorrect or incomplete data cannot be recognized. In order to detect faults thoroughly and as quickly as possible in hybrid platform, EasyHPS provides a hierarchical fault tolerance and recovery mechanism, which is the integration of process-level fault tolerance and thread-level fault tolerance.

A. Worker pool components in EasyHPS

There are three main components in the master/slave worker pool: computable sub-task stack, finished sub-task stack and overtime queue. There is also a sub-task register table used to record the executing sub-tasks in the master worker pool.

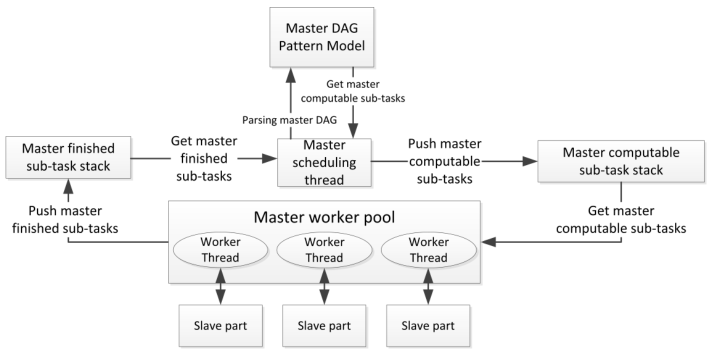
1) Computable sub-task stack: Computable sub-task stack is implemented as a linked list in EasyHPS. Elements in the linked list are the vertices' id in DAG Pattern Model which represents the corresponding computable sub-task. On processor-level, master worker pool gets computable sub-tasks from master computable sub-task stack and allocates them to the idle slave parts; on thread-level, slave worker pool gets computable sub-tasks from the slave computable sub-task stack and allocates them to the idle slave computing threads.

2) Finished sub-task stack: Finished sub-task stack is implemented as a linked list in EasyHPS. Elements in the linked list are the vertices' id in DAG Pattern Model which represents the corresponding sub-tasks that have been finished. Scheduler in master/slave part get finished sub-tasks from master/slave finished sub-task stack and updates the DAG pattern by removing the corresponding vertex and its connecting unidirectional edges.

3) Overtime queue: Overtime queue is a queue which stores the executing sub-tasks/sub-sub-tasks in EasyHPS. While a computable sub-task/sub-sub-task starts executing, both the current time information and the corresponding vertex's id are put in the overtime queue. The vertex's id and time information are removed after the corresponding sub-task is finished. Fault tolerance and recovery are implemented by detecting runtime failure through checking on the master/slave overtime queue in the master/slave worker pool.

4) Sub-task registered table: Sub-task registered table is used to record the executing sub-task in master worker pool. Before a computable sub-task is put in the master sub-task queue, it should register in the sub-task registered table firstly.

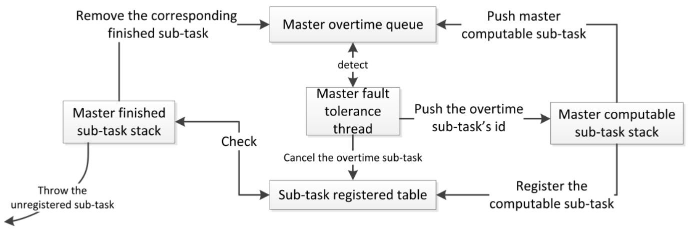
B. Task scheduling and fault tolerance in master worker pool



Master DAG Pattern Model
Get master computable sub-tasks
Parsing master DAG
Master finished sub-task stack
Get master finished sub-tasks
Master scheduling thread
Push master computable sub-tasks
Master computable sub-task stack
Push master finished sub-tasks
Master worker pool
Worker Thread
Worker Thread
Worker Thread
Get master computable sub-tasks
Slave part
Slave part
Slave part

Fig. 9: Task scheduling in master worker pool

Based on the master DAG Data Driven Model, master scheduler gets computable sub-tasks by parsing the master DAG Pattern Model constantly and make them to be executed in master worker pool in parallel. The whole process of task



Remove the corresponding finished sub-task
Master overtime queue
Push master computable sub-task
detect
Master finished sub-task stack
Master fault tolerance thread
Push the overtime sub-task's id
Master computable sub-task stack
Check
Cancel the overtime sub-task
Throw the unregistered sub-task
Sub-task registered table
Register the computable sub-task

Fig. 10: Fault tolerance in master worker pool

scheduling and fault tolerance in master worker pool is shown in Figure 9 and Figure 10. The details are described as follows:

a. Master DAG Data Driven Model is initialized.
b. Master worker pool is initialized. Master worker pool creates n worker threads where n is the number of slave computing nodes. In the master worker pool, each thread detects a slave computing node. Master fault tolerance thread detects the master overtime queue.
c. Master scheduling thread parses the master DAG Pattern Model and pushes the computable sub-tasks' id into the master computable sub-task stack. If master scheduling thread detects that all the sub-tasks has been finished, it will go to the step *i*.
d. Once one slave computing node is idle, the corresponding worker thread in master worker pool will get a computable sub-task's id from the master computable sub-task stack and send the sub-task's id with its necessary data to the idle slave computing node. At the same time, the computable sub-task is registered in the sub-task registered table and its information is pushed into the master overtime queue.
e. Once one slave computing node finishes its sub-task, it will reply master worker pool with a signal and the sub-task result.
f. Each worker thread detects the information of the corresponding slave computing node and puts the finished sub-task's id into the master finished sub-task stack.
g. Once a sub-task failure is detected, fault tolerance thread will cancel the registration of the corresponding sub-task in the sub-task register table and redistribute the same sub-task in the master overtime queue to continue computing.
h. Master scheduling thread gets finished sub-tasks' id from the master finished sub-task stack. If the sub-task is registered in the sub-task register table, master scheduling thread will update the master DAG Pattern Model by removing the corresponding vertex and its edges. Then it goes to step *c* again.
i. Task scheduling comes to the end. EasyHPS destroys the master worker pool, master DAG Pattern Model, master computable sub-task stack, master finished sub-task stack and master overtime queue, then sends end signals to all the slave computing nodes.

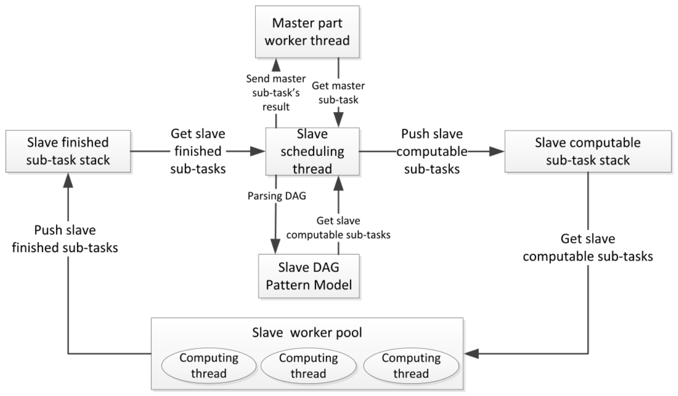

Master part worker thread
Send master sub-task's result
Get master sub-task
Slave finished sub-task stack
Get slave finished sub-tasks
Slave scheduling thread
Push slave computable sub-tasks
Slave computable sub-task stack
Parsing DAG
Get slave computable sub-tasks
Push slave finished sub-tasks
Slave DAG Pattern Model
Get slave computable sub-tasks
Slave worker pool
Computing thread
Computing thread
Computing thread

Fig. 11: Task scheduling in slave worker pool

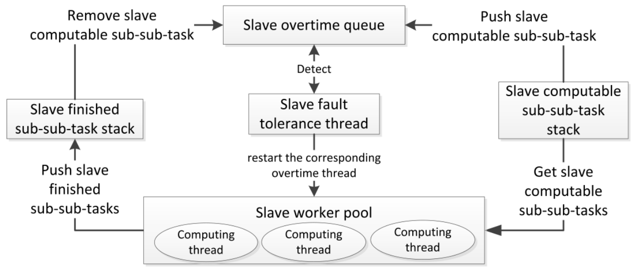

Remove slave computable sub-sub-task
Slave overtime queue
Push slave computable sub-sub-task
Detect
Slave finished sub-sub-task stack
Slave fault tolerance thread
Slave computable sub-sub-task stack
restart the corresponding overtime thread
Push slave finished sub-sub-tasks
Get slave computable sub-sub-tasks
Slave worker pool
Computing thread
Computing thread
Computing thread

Fig. 12: Fault tolerance in slave worker pool

C. Task scheduling in slave worker pool

Similarly, based on the slave DAG Data Driven Model, slave scheduler gets computable sub-tasks by parsing the slave DAG Pattern Model constantly and makes them to be executed in slave worker pool in parallel. The whole process of task scheduling and fault tolerance in slave worker pool is shown in Figure 11 and 12. The details are described as follows:

a. Slave scheduling thread sends an idle signal which indicates that it is idle to the master part.
b. Slave scheduling thread receives the signal from the master part. If it is the end signal, go to step *k*. Otherwise, it receives a computable sub-task's id with its necessary data from the master part.
c. Slave DAG Data Driven Model is initialized.
d. Slave worker pool is initialized. Slave worker pool creates n slave worker threads where n is the sum of slave computing threads. Slave fault tolerance thread detects the slave overtime queue.
e. Slave scheduling thread parses the slave DAG Pattern Model and puts the computable sub-sub-tasks' id into the slave computable sub-task stack. If slave scheduling thread detects that all the sub-sub-tasks has been finished, it will go to the step *j*.
f. Once one slave computing thread becomes idle, it will get a computable sub-sub-task's id from the slave computable sub-task stack and execute it. At the same time, the computable sub-sub-task's information is pushed into the slave overtime queue.
g. Each computing thread puts the sub-sub-task's id into the slave finished sub-task stack after it finishes computing the corresponding sub-sub-task.
h. Once a sub-sub-task failure is detected, slave fault tolerance thread will restart the corresponding computing thread.
i. Slave scheduling thread gets the finished sub-sub-tasks' id and updates the slave DAG Pattern Model by removing the corresponding vertex and its edges. Then it goes to step *e* again.
j. Destroy the slave worker pool, slave DAG Pattern Model, slave computable sub-task stack, slave finished sub-task stack and slave overtime queue. Slave task scheduling comes to the end. Then goes to step *b*.
k. Task scheduling in slave worker pool comes to the end.

VI. Evaluation

In order to verify that EasyHPS is efficient on DP problem parallelization, we have implemented two popular DP algorithms based on the EasyHPS, Smith-Waterman General Gap(SWGG) algorithm and Nussinov algorithm. Block-cyclic based wavefront[13] method is a parallel model which has been put forward to parallelize DP algorithms in computational biology and scientific computing. In this section, we have carried out a set of experiments based on EasyHPS and block-cyclic based wavefront method on Tianhe-1A system and compared their performances on DP problem parallelization. Each computing node of Tianhe-1A system is a multi-core SMP server which has dual 2.93GHz Intel Xeon 5670 six-core processors with 24GB memory. The network is Infiniband QDR. EasyHPS is implemented in C and complied with MPICH ver 1.4.1 and POSIX thread.

EasyHPS is deployed in master-slave mode on processor-level and thread-level. On thread-level, we use up to 11 threads to execute sub-tasks in parallel due to the limitation of hardware platform. We defined *Experiment_X_Y* as an experiment using Y cores on X multi-core nodes. For each experiment on EasyHPS, one multi-core node is used to do the task scheduling on processor-level and each of the other $(X-1)$ multi-core nodes for computing use one thread to do the task scheduling on thread-level. Therefore EasyHPS use $Y-2*X+1$ cores which are distributed on $(X-1)$ computing nodes to do the computing for *Experiment_X_Y* in fact. Due to the architecture of EasyHPS, when EasyHPS is deployed on N nodes, the sum of cores used is $N+(N-1)+ct*(N-1)$ where the 1st N describes the sum of scheduling threads on process-level, the 2nd $(N-1)$ describes the sum of scheduling threads on thread-level and ct is the sum of computing threads in one computing node. As mentioned above, the value of ct is from 1 to 11. That is the reason why we used 4-14 cores on two nodes and 10-40 cores on four nodes.

First, we have implemented Smith-Waterman General Gap algorithm based on EasyHPS with setting seq_len=10000 and process_partition_size=200/thread_partition_size=10. We have deployed EasyHPS on different number of nodes on Tianhe-1A system and done a series of experiments which are *Experiment_2_*$K_2(4 \leq K_2 \leq 14)$, *Experiment_3_*$K_3(7 \leq$

$K_3 \leq 27$), *Experiment_4_*$K_4(10 \leq K_4 \leq 40)$ and *Experiment_5_*$K_5(13 \leq K_5 \leq 53)$.

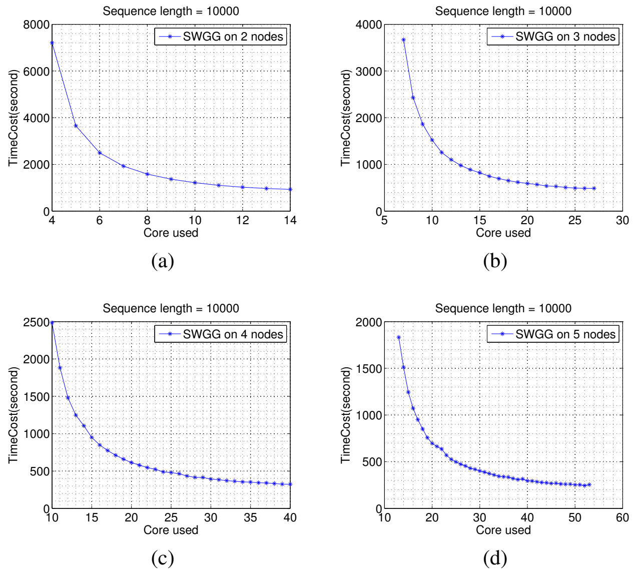


Fig. 13: SWGG algorithm implemented by EasyHPS on different number of multi-core computing nodes (a) Deployed on 2 nodes (b) Deployed on 3 nodes (c) Deployed on 4 nodes (d) Deployed on 5 nodes

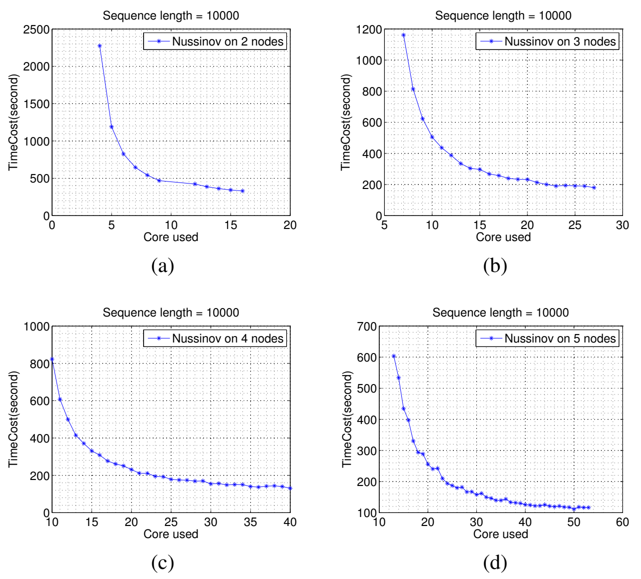


Fig. 14: Nussinov algorithm implemented by EasyHPS on different number of multi-core computing nodes (a) Deployed on 2 nodes (b) Deployed on 3 nodes (c) Deployed on 4 nodes (d) Deployed on 5 nodes

As shown in Figure 13, our EasyHPS system performs well when deployed on different number of computing nodes. While increasing the sum of computing cores used, our EasyHPS system achieves a great time reduction due to the parallelization on multilevel. In order to verify our system could be widely used to parallelize DP algorithms, we have also implemented Nussinov algorithm which is also a DP algorithm for RNA secondary structure prediction and done experiments with same settings as Smith-Waterman General Gap algorithm in EasyHPS. As shown in Figure 14, Nussinov algorithm implemented by EasyHPS system also has a good time reduction with increasing cores. Because each DP problem can be mapping into a DP formulation which guides DAG Data Driven Model initialization, EasyHPS can parallelize DP algorithms well.

We have also compared the performances of EasyHPS deployed on different numbers of computing nodes for Smith-Waterman General Gap algorithm and Nussinov algorithm. As shown in Figure 15(a), for Smith-Waterman General Gap algorithm, we can see that the performance using 20 cores on 4 nodes is better than that using the same number cores on 5 nodes, while the performance using 40 cores on 4 nodes is worse than that using the same number of cores on 5 nodes. The same results are also observed when Nussinov algorithm is deployed on different number of computing nodes, as is shown in Figure 15(b).

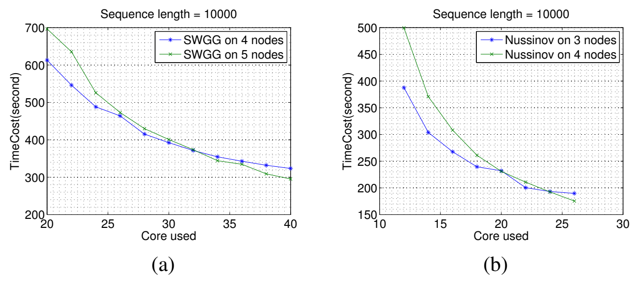


Fig. 15: (a) Comparsion of SWGG algorithm on different number of nodes (b) Comparsion of Nussinov algorithm on different number of nodes

More nodes can execute more sub-tasks in parallel and more nodes also cost more in data communication at the same time, so a good parallelization strategy should divide cores in an appropriate way in order to balance the positive impact and negative impact of multi-node on performance. Therefore we have chosen the optimal core group strategy for SWGG and Nussinov and shown the elapsed time line and speedup line while the total number of used cores increase in Figure 16. The ideal speedup is supposed to be linear. As the EasyHPS architecture, we should use at least 4 cores while parallelization. That is the reason why the speedup line for EasyHPS doesn't start at one node. From the experimental results, we can see that EasyHPS achieves 30 times speedup by using 50 cores for Smith-Waterman General Gap Algorithm and 20 times speedup by using 50 cores for Nussinov Algorithm.

In contrast with EasyHPS, the column based wavefront(CW)[15] and block-cyclic based wavefront(BCW)[13] are static data partitioning methods to parallelize DP algorithm. The CW algorithm can be viewed as a special BCW

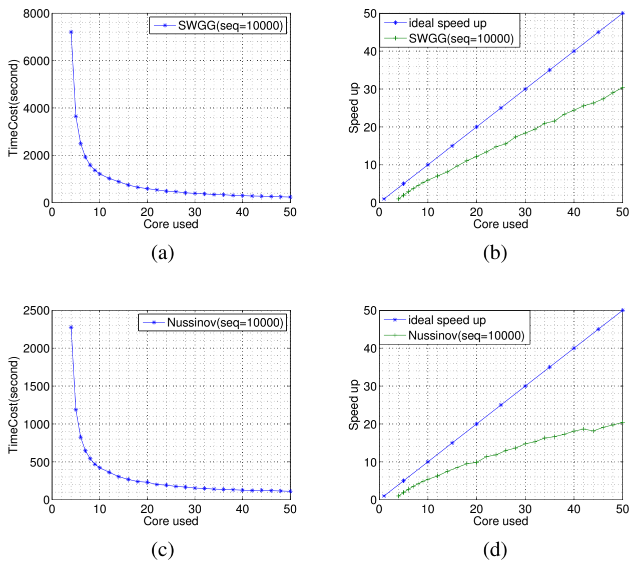


Fig. 16: (a) SWGG elapsed time (b) SWGG speedup (c) Nussinov elapsed time (d) Nussinov speedup

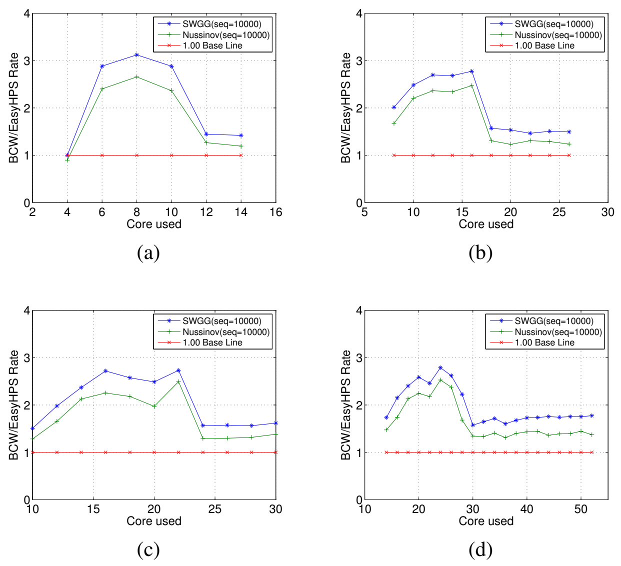


Fig. 17: BCW/EasyHPS rate of SWGG algorithm and Nussinov algorithm (a) Deployed on 2 nodes (b) Deployed on 3 nodes (c) Deployed on 4 nodes (d) Deployed on 5 nodes

algorithm when setting the argument block col to the result of data col divided by thread number of BCW. Thereby it only needs to compare the performance with BCW for EasyHPS. To the BCW algorithm implementation, we also adopt the DAG Data Driven Model, but the static worker pool is considered here. In order to compare the EasyHPS with BCW thoroughly and completely, we define the metric BCW/EasyHPS rate as BCW divided by EasyHPS with their run times in the same condition. Figure 17 presents the comparison experiments between EasyHPS and BCW method. The baseline 1.00 LINE is given. EasyHPS is better if the rate points are above the baseline, otherwise BCW is better. A conclusion drawn from the diagram is that EasyHPS is more efficient than BCW for DP algorithms, as it can be observed that almost all the experimental rate curves are above the 1.00 LINE. For static data allocation in BCW method, it has a fatal situation case during the runtime that there are some computable DAG nodes as well as some idle threads simultaneously, which never happens in EasyHPS.

VII. Conclusion and Future Work

In this paper, we introduce a runtime system named EasyHPS based on the DAG Data Driven Model for dynamic programming parallelization in multilevel computing environment. EasyHPS can do task allocation and scheduling automatically; and the only requirement is that the programmers implementation uses APIs supplied by EasyHPS, which reduces the complexity of parallel programming. To match the hardware architecture of cluster of multi-core/multi-processor computing nodes, the basic framework of EasyHPS is implemented based on MPI and POSIX Thread for the two levels of parallelization. A series of experimental tests have been conducted on Tianhe-1A system. Two dynamic programming algorithms have been implemented based on EasyHPS in order to verify the high performance on dynamic programming parallelization of EasyHPS. The results of experiments also show that proper deploying on multilevel environment can improve the performance. The comparison experiments with block-cyclic Wavefront method show advantages of EasyHPS for parallelization on multilevel computing environment.

Despite the good performance on DP algorithm parallelization, there are still some shortcomings in EasyHPS. Space complexity is a major issue in the usability of many DP algorithms. When parallelizing DP algorithms, EasyHPS consumes a lot of memories, which could be a limitation in the future work. DAG Data Driven Model can be also improved to adopt more kinds of algorithms for parallelization.

Acknowledgment

The work is sponsored by the National Natural Science Foundation of China (71131007), Key Technologies R&D Program of Tianjin, China (11ZKFGX01000), and FP7 of European Union (No. 287746).

References

[1] J. Bowie, R. Luthy, and D. Eisenberg, “A method to identify protein sequences that fold into a known three-dimensional structure,” *Science*, vol. 253, no. 5016, pp. 164–170, 1991.
[2] X. Huang, “A space-effcient parallel sequence comparsion algorithm for a message-passing multiprocessor,” *International Journal of Parallel Programming*, vol. 18, no. 3, pp. 223–239, 1989.
[3] C. Ciressan, E. Sanchez, M. Rajman, and J. Chappelier, “An fpga-based coprocessor for the parsing of context-free grammars,” in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000.
[4] M. Farach and M. Thorup, “Optimal evolutionary tree comparsion by sparse dynamic programming,” in *Proceedings. IEEE International Conference on Cluster Computing*, 2003, pp. 456–459.

[5] P. Edmonds, E. Chu, and A. George, “Dynamic programming on a shared memory multiprocessor,” *Parallel Computing*, vol. 19, no. 1, pp. 9–22, 1993.

[6] Z. Galil and K. Park, “Parallel algorithm for dynamic programming recurrences with more than o(1) dependency,” *Journal of Parallel and Distributed Computing*, vol. 21, no. 2, pp. 213–222, 1994.

[7] P. Bradford, “Efficient parallel dynamic programming,” in *Proceedings. 30th Ann.Allerton Conf. Comm. Control and Computing*, 1992, pp. 185–194.

[8] G. Tan, H. Sun, and R. Gao, “A parallel dynamc programming algorithm on a multi-core architecture,” in *Annual ACM Symposium on Parallelism in Algorithms and Architecture*, 2007, pp. 135–144.

[9] G. Tan, S. Feng, and N. Sun, “Locality and parallelism optimization for dynamic programming algorithm in bioinformatics,” in *Proceeding of the 2006 ACM/IEEE Conference on Supercomputing, SC’06*, 2006, pp. 11–17.

[10] F. Almeida, R. Andonov, and D. Gonzalez, “Optimal tiling for rna base pairing problem,” in *Proceedings of the 14th Ann. ACM Symp. Parallel Architecture and Algorithm (SPAA 02)*, 2002, pp. 173–182.

[11] W. Zhou and D. K. Lowenthal, “A parallel, out-of-core algorithm for rna secondary structure prediction,” in *Proceedings of the 35th Intl Conf. Parallel Processing (ICPP 06)*, 2002, pp. 74–81.

[12] W. Liu and B. Schmidt, “Parallel pattern-based systems for computational biology: A case study,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 8, pp. 750–763, 2006.

[13] W. Liu and B. Schmidt, “A generic parallel pattern-based system for bioinformatics,” in *Proc. EURO-PAR*, 2004, pp. 989–996.

[14] S. Tang, C. Yu, J. Sun, B.-S. Lee, T. Zhang, Z. Xu, and H. Wu, “Easypdp:an efficient parallel dynamic programming runtime system for computational biology,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 5, pp. 862–872, 2012.

[15] V. Kumar, A. Grama, A. Gupa, and G. Karypis, *Introduction to Parallel Computing*. The Benjamin-Cummings Publishing Company,Inc., 1994.

[16] R. Nussinov, G. Pieczenik, J. R. Griggs, and D. J. Kleitman, “Algorithms for loop matchings,” *SIAM Journal on Applied Mathematics*, vol. 35, no. 1, pp. 68–82, 1978.

[17] T. Smith and M. Waterman, “Identification of common molecular subsequences,” *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, 1981.