Abstract

Dynamic Programming has a long history in the development of algorithms whenever a complex problem can be broken down into a sequence of optimal subproblem. At the same time, parallel computer systems are rapidly gaining popularity as processor costs decrease and as system interconnect speeds increase. The combination of multiple processor systems and dynamic programming seems a natural fit, and indeed, much research has been done on the topic since the first multiple processor systems became available to researchers. This paper examines a representative cross-section of research papers from various eras of parallel algorithm research and their impact. Future research directions based on each paper are put forward.

1 Introduction

1.1 Dynamic Programming

There exists a large set of programming problems that generically fall under the category of optimization problems. These are problems that have multiple possible solutions, but generally only one optimal solution. This optimal solution is the solution that is maximal or minimal for a given instance of the problem. These problems can usually be decomposed into subproblems, which are simply smaller versions of the same type of problem. Finding the optimal solution to a given instance of this kind of optimization problem is simply the sum of the optimal solutions to the subproblems.

The traditional way to solve this category of problems is called dynamic programming. By creating a table of solutions to the optimal subproblems, we can create a schedule for the complete solution of the optimization problem. The general process for implementing a dynamic programming algorithm begins by finding the structure of the optimal subproblems, then recursing on the original data defining the problem set until the base case of the recursion is reached. Combining the values of the solutions to the subproblem yields the overall solution for the problem instance, but doesn’t yield the solution itself. For example, a traditional dynamic programming task is finding the longest common subsequence of characters in two sequences. The process defined to this point will only tell us how long
the common subsequence is, and not what the sequence actually is. Proper bookkeeping
during the process allows the algorithm to retrace its steps and generate the solution.

Two of the most common dynamic programming problems introduced by textbooks and
researchers are *Longest Common Subsequence* and *Matrix Parenthesization*. The former
problem has been alluded to already. The inputs to the problem are two sequences of
characters or numbers and the task is to find the longest subsequence of characters that
the two input sequences have in common. There are various flavors of this problem where
the characters must be sequential and others where this constraint is lifted. The most
common practical application for this problem is genetic sequencing. Especially given the
length of genetic sequences, this problem demands to be run on larger parallel machines.
The Matrix Parenthesization problem attempts to find the minimum cost order in which
to perform a sequence of matrix multiplications. The goal is to reduce the number of scalar
multiplications needed to generate the final product matrix. More details on any of these
problems can be found in any of the papers in the References.

### 1.2 Parallel Programming

The key idea of parallel programming is to partition a given problem such that multiple
processors are kept as equally busy as possible in order to dramatically reduce the runtime
of a given problem instance. Generally we cannot reduce the cost of an algorithm from
say, $O(n)$ to $O(\log n)$ by throwing multiple processors at a problem. Rather, we reduce the
complexity by a constant factor. Of course, the complexity reduction is tightly coupled to
the amount of parallelization that can be done to the problem and the data set.

The key question that is usually broached when talking about parallel algorithms is how
to do that partitioning. The target machine architecture usually factors heavily into the
partitioning paradigm. The number of processors, the memory architecture and overhead
costs — especially interprocessor communication costs — factor into the algorithm design.
Thus the focus of most research papers on parallel algorithms focus on the partitioning
step.

Most dynamic programming problems are classified as $\mathcal{NC}$ — belonging to the set of
decision problems decidable in polylogarithmic time on a parallel computer with a polyno-
mial number of processors. In other words, a problem is in $\mathcal{NC}$ if there are constants $c$ and
$k$ such that it can be solved in time $O((\log n)^c)$ using $O(n^k)$ parallel processors. Intuitively,
this simply means that the problem can be efficiently solved on a parallel machine in much
the same way that a $\mathcal{P}$ problem can be efficiently solved in polynomial time.

Frequently $\mathcal{NC}$ parallel algorithms are defined to run on a *PRAM* or Parallel Random
Access Machine. The PRAM is a totally academic and hypothetical construct, as there is
no way to effectively implement a PRAM in real hardware. There are various flavors of the
PRAM which are usually divided by memory access paradigms. The PRAM would appear
to be rapidly falling out of favor with researchers due to this hypothetical nature combined
with the resulting processor requirements. For example, a simple matrix parenthesization
may require as many as \( n^3 \) processors. Considering just a simple six matrix chain multiplication, a PRAM algorithm may demand 216 processors. While this may be “efficiently solvable”, it is hardly practical. As such, newer research tends to focus on more practical and realizable hardware.

1.3 Organization

The rest of this paper will be organized as follows. Each of five papers will be introduced and summarized, attempting to gauge their impact on the overall field of parallel dynamic programming. Shortcomings will be noted, and future research directions as well. Findings will be summarized at the end.

2 Published Paper Summaries

2.1 Gilmore

It would seem that research into parallel dynamic programming is far older than the reader may have supposed. In 1968, P. A. Gilmore, a Goodyear Aerospace engineer, authored a paper titled “Structuring Parallel Algorithms” [1] which might well be considered one of the seminal works on the topic. Gilmore’s paper largely revolves around the application of dynamic programming to two different problems. One is purely mathematical in nature and the other focuses on the parsing of an old mathematical language called Mad. He very clearly demonstrates the fundamentals of dynamic programming, but the paper is not without fault.

Gilmore uses an example arithmetic function to show how dynamic programming works. Given a multi-variable function \( R_n \) in \( n \) variables, we wish to find values for each \( x_n \) to maximize \( R_n \). An example of this would be

\[
R_6(x_1, x_2, x_3, x_4, x_5, x_6) = x_1 + x_2^2 + x_3^{\frac{1}{2}} + 2 \sin x_4 + g_5(x_5) + g_6(x_6),
\]

where \( g_5 \) and \( g_6 \) are any scaling function. The goal is to find the maximized value for \( R_6 \) over the range \( x_i = 0(0.1)2 \). This strange syntax indicates the values of \( x_i \) will vary from 0 to 2 (inclusive) in increments of 0.1. Obviously, if we find maximal values for each \( x_i \), the sum will similarly be maximal. This seems to be a rather trivial problem without an apparent practical application. However, due to the lack of interdependence of the calculations, it can be fully parallelized. It is worth noting that this point is not explicitly mentioned in the text.

The second problem discussed is parsing out “triples” in an old mathematical programming language called MAD. A triple is a group of three elements of a statement that are, in order: operand, operator, operand. The example code used in the paper is

\[
F = A + B * .ABS.(C + D).
\]

An example triple would be \( C + D \). The goal of this exercise is to parse this statement to find triples and eventually discover what the overall value of this statement is, given values for \( A, B, C \) and \( D \). The author argues that the most efficient way to parallelize this is to throw a processor at each operator. While this may be the case, this doesn’t
seem like a true dynamic programming task. We are not actually optimizing anything, though there are some subproblems that must be solved (building a triple whose result will eventually feed another triple), they are not optimal subproblems.

It would certainly be interesting to see how the parsing and compilation process can be parallelized, but it would seem there are more appropriate ways than dynamic programming to do it.

2.2 Karypis and Kumar

Perhaps a more seminal work than Gilmore’s paper would be “Efficient Parallel Formulations for Some Dynamic Programming Algorithms” [2]. Karypis and Kumar are quite well known in the field, as they have truly “wrote the book” on parallel programming. As a primer in dynamic parallel programming, this paper is a truly comprehensive look at the matrix parenthesization problem. Perhaps the only considerable downside to this paper is its reliance on the PRAM architecture for its algorithm analysis.

After a thorough discussion of the problem itself, the authors launch into three different partitioning paradigms, discuss the merits of each from a theoretical point of view and run experiments on an nCUBE/2 workstation. The first partitioning method is perhaps the most intuitive, and the authors refer to it as “Checkerboarding”. The matrix parenthesization problem creates a $n \times n$ matrix of values which represent the number of scalar multiplications required by that combination of matrix multiplications. However, the matrix is only filled out above the identity diagonal. The Checkerboard partitioning method will inevitably assign a big empty chunk of the matrix to at least one processor. Obviously this is a less than ideal situation, as a good partitioning will keep all of the processors busy equally. One benefit of this mapping though is the low communication cost between processors. Our next step should then be to get all the processors busy, but retain the low communication overhead. Thus enters the Modified Checkerboard Mapping. Essentially, we repeat the checkerboarding to give some work to the otherwise inactive processor. This is better, but some processors are still more heavily burdened than others. The Shuffling Mapping addresses the unequally burdened processor issue while only mildly worsening the issue of processor communication. The basic idea is to break up the populated part of the matrix into chunks that are $\sqrt{p} \times \sqrt{p}$, where $p$ is the number of processors. Each processor then gets one of those matrix calculations — repeat until all $n$ elements have been assigned to a processor.

The authors then analyze the theoretic efficiency of these mappings, and find that Checkerboarding has a best-case efficiency of only $16.7\%$. Modified Checkerboarding tops out at $66.7\%$ efficiency, but Shuffling approaches $100\%$ as $n$ increases. Experimental results match with the theory.

Most future work would likely involve submitting the other mappings and partitioning strategies in the other papers, especially the “striping” discussed in [4] and the diagonal strategy in [3] to the same kind of rigorous analysis as the other mappings in the paper,
and to examine these mappings on a more modern architecture than the PRAM.

2.3 Lewandowski

Lewandowski, Condon and Bach collaborated on [3], titled “Asynchronous Analysis of Parallel Dynamic Programming Algorithms”. This paper is very different in nature from the other papers in this survey. Rather than focus on algorithms to solve specific problems, the authors here attempted to create an asynchronous model for analyzing parallel dynamic programming. They work with the assumption that the time to complete a task is random, but follows some sort of probability distribution. Intuition and experimental work indicates that this distribution is the exponential distribution. The model then attempts to predict which of two algorithms will run faster. Largely, the authors concern themselves with two categories of algorithms: the pipeline variety and the diagonal variety. Since dynamic programming is about building a table of intermediate results, regardless of the application domain, we can build solutions by walking through that table one of two ways: row by row (pipelined) or diagonal by diagonal. For problems like the matrix multiplication, the diagonal method seems most intuitive, but the model would suggest that processors will be idling much more with the pipeline method. Indeed, the greater the size of the input $n$, the more dramatic the difference between the pipeline and diagonal methods becomes.

After presenting the results, the authors begin extensive formal proofs of the lower bounds of dynamic programming running time and the underlying operation of their model. These proofs are quite complex, but very thorough. Lower bounds for these algorithms range from $2n - 1$ for $n = p$ up to $O(n^2/p)$ when $p$ is just greater than 1, but considerably less than $n$. It is quite interesting to note that simply throwing processors at the problem can indeed reduce the complexity. The question is then one of cost — $n$ processors isn’t terribly realistic if $n = 1000000$!

Finally, the algorithms are implemented in C on a machine called the “Sequent Symmetry”, a shared memory parallel machine. All of the theoretical measurements match up nicely to the theoretical results. The paper is quite comprehensive. The key areas for future research would be in using the model to analyze some other mappings, especially those of [2]. Also, analyzing the performance of these algorithms on a more modern machine would be enlightening.

2.4 Martins

Martins and four other researchers contributed to [4], “A Multithreaded Parallel Implementation of a Dynamic Programming Algorithm for Sequence Comparison”. This paper does not seem to be up to the same level of quality as the others in this selection. The authors focus on biological sequence comparison, and spend an inordinate amount of the paper in background explanation. Several pages are dedicated to an explanation of dynamic programming as a paradigm, the Longest Common Subsequence problem and parallel com-
putation. Finally, a scant \( \frac{3}{4} \) page is dedicated to discussing the partitioning method used. It is similar to the pipelining algorithm from [3], but with a twist — something the authors call “fibers”. These fibers are to threads as threads are to processes. By “striping” the dataset across the processors, we can keep the processors busy on calculations on one row (done in one fiber) while another fiber is waiting on communication in a different row. The authors claim that simply by doing this, “processor utilization rises to 75%”, without providing any supporting details as to how this is true.

The authors spent a number of pages discussing the EARTH multiprocessor machine that was their target platform for their algorithm. EARTH and its accompanying programming language Threaded-C are the pet project of K.B. Theobald and doesn’t seem to have wide usage outside of Theobald’s own activities. While this is not inherently bad, it seems as though what few results are presented cannot have wide applicability.

The conclusions presented by the paper are “fuzzy” and weak. There are a lot of future research possibilities simply in verifying the results of this paper and getting a handle on the implementation of fibers.

2.5 Alves

In [1], Alves, Cáceres and Dehne wrote about the String Editing problem, also within the field of computational biology. The String Editing problem takes two strings \( A \) and \( C \) and using well-defined operations of insertion, deletion and substitution, transforms \( A \) to \( C \). The goal is to find the most cost-effective method of doing this — meaning the lowest cost. The structure of the optimal subproblem is finding a shorter prefix string’s minimal cost transformation and building the solution on progressively longer strings. Rather than doing the analysis on a table, a transformation is made into a grid graph, a weighted, directed, acyclic graph that has the same basic form as the table. By heavily weighting the edges of the graph representing the optimal solution, we trivialize finding the solution. Using any normal graph algorithm, we find the path through the graph that starts at \( A \) and ends at \( C \) with the highest weight. The authors define the algorithm based on an architecture called the CGM, a “coarse grained multicomputer”. This is a very different architecture than the PRAM or the asynchronous machine in [3]. The CGM works in cycles of alternating local computation and global communication, and the goal of any CGM algorithm would be to minimize the communication rounds. The authors claim that the CGM paradigm maps very well to cluster systems, which is a refreshing departure from shared memory machines. The final runtime for the algorithm is \( O\left(\frac{n^2 \log m}{p}\right) \) algorithm, where \( n \) is the length of the target string \( C \), \( m \) is the length of the source string \( A \) and \( p \) is the number of processors. This is constant factor improvement over the best known sequential approximation algorithm. There are only \( O(\log p) \) communication rounds and the memory requirements are only \( O\left(\frac{mn}{p}\right) \).

The partitioning of the grid graph is done in a method that is similar to Modified Checkerboarding. Small chunks of the graph are calculated and then joined. The JOIN
algorithm is extremely convoluted, but seems to work and is highly parallelizable.

The most considerable downside to this paper is the lack of experimental results. The authors present an excellent proof of their algorithm’s correctness and complexity, but have apparently not attempted to implement the algorithm in real hardware. This is an obvious hole in the research that demands to be patched.

3 Concluding Remarks

One question posed during the timeframe of this survey was whether or not any mechanical proofs of correctness were used by any of the authors of the papers, and the answer is clearly, “No”. The authors have relied on old-fashioned and (in some cases) quite complex proofs of correctness and complexity. It has been interesting to see the rise and fall of the PRAM as researchers focus on more realizable hardware.

There is clearly no “right” or “best” way to partition a dynamic programming table (or graph), and to a large extent the partitioning is heavily dependent on some amount of a priori knowledge of the nature of the problem. For example, Checkerboarding works well for LCS, but extremely poorly for Matrix Parenthesization.

It was the author’s intent in this survey to try to gain a better understanding of some current parallel programming paradigms, and in this respect, the project was a success. Whether any real windows for future research have opened remains to be seen, but the field is still ripe with opportunities.

References


