Automatic Hybrid OpenMP + MPI Program Generation for Dynamic Programming Problems

Denny R. VandenBerg
Computer Science and Engineering
University of Michigan
Ann Arbor, MI 48105
Email: devanden@umich.edu

Quentin F. Stout
Computer Science and Engineering
University of Michigan
Ann Arbor, MI 48105
Email: qstout@umich.edu

Abstract—We describe a program that automatically generates a hybrid OpenMP + MPI program for a class of recursive calculations with template dependencies. Many useful generalized dynamic programming problems fit this category, such as Multiple String Alignment and multi-arm Bernoulli Bandit problems. Solving problems like these, especially those involving several dimensions, can use a significant amount of memory and time. Our generator addresses these issues by dividing the problem into many tiles that can be solved in parallel. Programs generated using this program generator are capable of solving large problems and achieve good scalability when run on many cores. The input supplied to the generator is a high level description of the problem; the output is a fully functioning parallel program for a cluster of shared memory nodes. This high level approach to parallel computation allows the generator to have a large amount of control over memory allocation, load balancing and calculation ordering.

Keywords—dynamic programming; automatic program generation; hybrid; openmp; mpi; bandit problem

I. INTRODUCTION

The rapid introduction of parallelism into commodity computers has greatly increased the complexity of programming them. To address this we have developed an automatic hybrid OpenMP + MPI program generator for multidimensional dynamic programming problems. The goal of this work is to develop a high level tool that can be used to obtain functioning high performance code quickly and easily.

The problems that our code generator supports can be described as a recursive function \( f(\vec{x}) \) with the goal of computing some value, e.g. \( f(\vec{0}) \). The function \( f \) (not including base cases) has a template recurrence of the form:

\[
f(\vec{x}) = F(f(\vec{x} + \vec{r}_1), f(\vec{x} + \vec{r}_2), \ldots, f(\vec{x} + \vec{r}_k))
\]

where each \( \vec{r}_i \) is a constant vector.

Many scientists, especially those in bioinformatics, can use dynamic programming algorithms to help solve their problems. For example:

- Bandit problems are a fundamental model of stochastic optimization based on sequentially trying alternatives ("arms") and observing the rewards obtained. An important application of this model is to adaptive clinical trials where one wants to allocate the best treatment options to patients as outcome results for previous patients become available. Adaptive clinical trials save both time and money [1] compared to standard fixed allocation clinical trials, as well as being more ethical while remaining statistically sound. Each treatment option can be represented as an arm of a bandit problem. The \( k \)-arm Bernoulli bandit problem can be solved with \( 2k \)-dimensional dynamic programming [2]. A dynamic programming solution to the 2-arm problem is given in Figure 1. In [3] they hand optimized and parallelized a dynamic programming algorithms for the 3-arm Bernoulli bandit problem.

- Multiple Sequence Alignment (MSA) is the problem of finding the minimal cost alignment between a set of given sequences. Scientists can use this to find similarities in DNA or proteins. These similarities can be used to determine evolutionary history or the purpose of genes or proteins. MSA is related to edit distance, but is more general. The general idea is to insert gaps into the two strings with the aim of matching up related areas of strings. There is also a scoring matrix that is used to determine the distance between two characters. In real-world problems there can be a variable cost for gaps [4]. One example is having an initial gap cost more (Gap Creation Penalty) than extending an already existing gap (Gap Extension Penalty). Multiple Sequence Alignment can be represented as a \( d \)-dimensional problem, where \( d \) is the number of strings to be aligned. However, because of the dynamic programming algorithm’s slow run time, approximations are generally used for matching more than 2 strings. Unfortunately, the quality of such approximations can be poor, so in [5] they use FPGAs (Field Programmable Gate Arrays) to achieve some excellent speedups on 4 and 5 sequence alignment problems. However, they note the increasing difficulty of programming the FPGA for higher dimensions. MSA could benefit from a way to automatically generate efficient parallel code that solves the problem exactly using commodity parallel computers. A related problem, Longest Common Subsequence (LCS), can also be represented as a dynamic programming problem. LCS is also useful for matching multiple strands of DNA [6].

Processor manufacturers have been increasing the number of cores on a chip instead of increasing clock frequencies.
This is making computers more difficult for the average user to effectively utilize. These cores can be programmed using a shared memory programming standard such as OpenMP. Although it is possible to use a distributed memory model on a shared memory architecture, one of the most important reasons for using a shared memory model is that there is less overhead for sharing data between cores. However, to solve even bigger problems it is generally necessary to use to a distributed memory model of computation. Hybrid OpenMP + MPI is a natural way of programming clusters of shared memory nodes and it reflects the underlying architecture of most clusters.

There are many common issues in implementing dynamic programming for large problems, especially multidimensional ones. The number of calculations required, and the space needed to store intermediate results, can grow exponentially in the number of dimensions. Parallel computing helps address both of these issues by providing more total processing power and more RAM. The memory issue can also be addressed by discarding intermediate results as they are no longer needed, but this requires determining efficient orderings of the calculations. Another problem is that typically there are few calculations per memory access, which complicates the problem of determining efficient calculation orderings. Finally, writing scalable parallel programs which take all of the above into account is quite difficult.

We take a high level approach to parallel computation. The user input is mathematical description of the problem they wish to solve; the output is a fully functioning program that they can use to solve their problems on cluster computers.

Section II gives an example of a dynamic programming solution to the 2-arm Bandit problem. Section III describes related work. Section IV describes how the automatic program generator functions, including a description of the user input. Section V describes the programs produced by the program generator and how they work. Section VI evaluates the performance of the generated code by looking at scaling and efficiency. In Section VII we discuss future work.

## II. Example Problem: 2-arm Bandit

The 2-arm bandit as well as other bandit problems is one of the motivating problems of this work. This problem is 4 dimensional with the goal of computing \( V(0) \) which signifies the maximum expected number of successes over \( N \) trials before any arms are pulled. The coordinates of each location \((s_1, f_1, s_2, f_2)\) represent the results of all the pulls so far, where \( s_i, f_i \) represent the number of successes and failures, respectively, observed on arm \( i \). \( V(s_1, f_1, s_2, f_2) \) represents the expected number of successes over all \( N \) trials given the observations so far, assuming optimal decisions are made at each step.

In the equation for \( V \) below, \( p_i \) is the calculated probability that the next pull of arm \( i \) will be a success given the number of successes \( s_i \) and failures \( f_i \) so far.

\[
V(s_1, f_1, s_2, f_2) = \max(V_1, V_2)
\]

for \( s_1 \) from \( N \) to 0 do

for \( f_1 \) from \( N - s_1 \) to 0 do

for \( s_2 \) from \( N - s_1 - f_1 \) to 0 do

for \( f_2 \) from \( N - s_1 - f_1 - s_2 \) to 0 do

\[
V_1 = p_1 \cdot V(s_1 + 1, f_1, s_2, f_2) + (1 - p_1) \cdot V(s_1, f_1 + 1, s_2, f_2)
\]

\[
V_2 = p_2 \cdot V(s_1, f_1, s_2 + 1, f_2) + (1 - p_2) \cdot V(s_1, f_1, s_2, f_2 + 1)
\]

\[V(s_1, f_1, s_2, f_2) = \max(V_1, V_2)\]

end for

end for

end for

end for

Fig. 1. 2-arm Bandit Pseudo Code

where

\[
V_1 = p_1 \cdot V(s_1 + 1, f_1, s_2, f_2) + (1 - p_1) \cdot V(s_1, f_1 + 1, s_2, f_2)
\]

\[
V_2 = p_2 \cdot V(s_1, f_1, s_2 + 1, f_2) + (1 - p_2) \cdot V(s_1, f_1, s_2, f_2 + 1)
\]

Figure 1 gives a program for evaluating this recurrence.

An *iteration space* is the set of all possible assignments of the iteration variables of a problem. The iteration space for the dynamic programming algorithm in Figure 1 is all integer solutions to the linear constraints:

\[
s_1 + f_1 + s_2 + f_2 \leq N
\]

\[
s_1, f_1, s_2, f_2 \geq 0
\]

## III. Related Work

A SARE (System of Affine Recurrence Equations), introduced by Karp et al. [7], is a way of describing a “massively parallel algorithm” [8]. This representation is general enough to describe everything from matrix multiplication to PDEs and dynamic programming. Parameterized SAREs (PAREs), a more generalized form of SAREs, were introduced by Quinton and Van Dongen [9] because most algorithms have input parameters that vary the size of the problem and changing these parameters does not always justify recreating the entire code. Much work on SAREs focuses on systolic arrays which are processors connected in a lattice pattern.

The polytope model is a way of representing calculations over an iteration space defined by a polytope. The iteration space for the 2-arm bandit problem shown in Section II is a 4-dimensional polytope with a parameter \( N \). Many problems can be described using the polytope model, including matrix multiplication and LU decomposition. Work in the polytope model is more focused on optimizing and parallelizing compilers. These compilers generally read in programming source code, attempt to extract parallelism or optimize memory accesses, and then output optimized source code. The Pluto compiler [10] automatically generates parallel OpenMP code from the original source code. LooPo [11] is another compiler for the polytope model. In [12] they develop an approach for generating code for distributed memory systems. The Chunky
Loop Generator (CLooG) [13] is not a compiler, but generates loops for problems in the polyhedral model. It is used as a library in various programs, and has been included as a part of GRAPHITE in GCC 4.4 [14].

DPSkel [15] solves dynamic programming problems in an object oriented manner. Problems are solved using pre-defined and efficient algorithms for a variety of platforms including distributed memory systems. The idea behind skeletons is that the user won’t have to rewrite the program in order to change it to serial, parallel distributed memory or parallel shared memory. This is a flexible system, and can solve some classes of dynamic programming problems that are not solvable by the current work, but it doesn’t appear that the current implementation of DPSkel has skeletons of more than 2 dimensions.

IV. DESCRIPTION OF AUTOMATIC PROGRAM GENERATOR

In this section we give a description of the input to the program generator and the tasks that are handled for the user by the generator. These tasks include generating the code for the loops that do the calculations, handling the communication between nodes, managing memory during the calculation process, and load balancing.

This generator divides the work of the problem into individual tiles. Each tile depends only on a few of its neighbors. One benefit of using tiling for parallel programs is that the sizes of the tiles can be adjusted to reduce the frequency of communication or synchronization. A complete description of tiling can be found in [16].

One of the advantages of using a higher level strategy is that we have the flexibility to drastically change the generator and the resulting code without impacting how the user interacts with it. This allows us to add features, or make significant changes in the memory allocation or calculation ordering, transformations not easily achieved with a typical compiler.

A. Input Description

The user supplies several inputs to the program generator. First, the user must specify the names of the loop variables which we will refer to as \( \vec{x} = \langle x_1, x_2, \ldots, x_d \rangle \), where \( d \) is the number of dimensions. The user must also specify the names of the input parameters, such as \( N \) for the 2-arm bandit problem in Section II. Another input to the generator is a set of template vectors describing the dependencies of the problem. For example, the template vectors for the 2-arm bandit are:

\[
\vec{r}_1 = \langle 1, 0, 0, 0 \rangle \\
\vec{r}_2 = \langle 0, 1, 0, 0 \rangle \\
\vec{r}_3 = \langle 0, 0, 1, 0 \rangle \\
\vec{r}_4 = \langle 0, 0, 0, 1 \rangle 
\]

The input to the code generator is a text file containing the following information:

- The names of input parameters, e.g. \( N \),
- The name of the state array,
- A system of linear inequalities describing the iteration space of the problem,
- A set of template vectors describing the dependencies of each location,
- The loop ordering of the variables,
- The names and ordering of the dimensions for load balancing \( lb_1, lb_2, \ldots, lb_j \) where \( j \) is the number of dimensions selected for load balancing and \( j \leq d \), and
- The tile widths for each dimension \( w_k \) where \( k = 1 \ldots d \).

B. Programming interface

In order for the user to be able to write code that takes advantage of the generator’s features, the center loop code has certain restrictions that need to be met. The main reason for this is that the user is not in control of memory allocation and deallocation of the tile data. Access to the state array is provided by integer variables we refer to as mapping functions that supply the offsets to memory locations relative to the current location. For example, the center loop code for 2-arm bandit code in Figure 1 would look like:

\[
V_1 = p_1 \cdot V[\text{loc}_1] + (1-p_1) \cdot V[\text{loc}_2] \\
V_2 = p_2 \cdot V[\text{loc}_3] + (1-p_2) \cdot V[\text{loc}_4] \\
V[\text{loc}] = \max(V_1, V_2)
\]

There are not many restrictions on the commands that can be executed in the center loop. However, the center loop code should be restricted to calculating only the current location. Also, the user should not rely on the calculations being executed in any particular order, other than the fact that it will be valid, i.e., no state is accessed before it has been calculated.

The following is a list of all the symbols provided to the user by the program generator:

- State array \( V \) is managed by the generated program, but accessible by a user defined name.
- The current location is stored at \( V[\text{loc}] \).
- A dependence named \( r_1 \) would be located at \( V[\text{loc}_r1] \).
- The validity of a dependence named \( r_1 \) can be determined by checking a variable named \( \text{is_valid}_r1 \). This variable is useful for determining whether or not a base case is needed to calculate the associated location.

C. Generation Process

In this section we describe some of the various components that are involved in generating the resulting program.

The generation process follows this plan:

1) Create iteration spaces
2) Determine tile dependencies
3) Create template recurrence validity functions
4) Create mapping functions
5) Generate code
   a) Print out data structure code
   b) Generate packing and unpacking functions
   c) Generate load balancing code
D. Loop Bound Generation

There are several places in the generated program that execute loops. There is a lot of prior work on generating loops for scanning polyhedra. An overview of prior work can be found in [17]. Two well-known techniques used to generate loop bounds from polyhedra are Parametric Integer Programming (PIP) [18] using PipLib [19] and the Fourier-Motzkin elimination method. A comparison of these appears in [20].

Fourier-Motzkin elimination is a simple method for generating loop bounds. The method works by pairing up two inequalities that have the target variable on opposite sides and eliminating that variable. For example, to eliminate \( x_2 \) from the inequalities \( x_1 \leq x_2 \) and \( x_2 \leq x_3 \), clearly \( x_1 \leq x_2 \leq x_3 \), so elimination means to simply remove the two constraints containing \( x_2 \), and replace them with the new constraint \( x_1 \leq x_3 \). This process is repeated for all pairs of constraints containing \( x_2 \). The loop bounds are found by eliminating the variables one by one from the system starting from the innermost loop. The method generates perfectly nested loops using min and max functions. The disadvantage is that the number of constraints can multiply dramatically with each iteration. This method can generate up to \( \frac{n^2}{2} \) constraints after the elimination of a single variable and many duplicate and redundant constraints are created. Since the number of constraints can blow up if kept unchecked, these duplicate and redundant constraints must be removed after each iteration to prevent further slow down (and memory use). We employ Fourier-Motzkin elimination [21] to generate these loop bounds and whenever we need to eliminate variables from a system of inequalities. We have found that for our purposes FM-elimination works well.

E. Creating the Iteration Spaces

An iteration space is all the possible valid assignments for a set of loop variables. Generating the code requires several iteration spaces. There are 3 main iteration spaces that are created by the generator. One iteration space identifies each tile uniquely. Another iteration space for load balancing. The third and most important iteration space is the one used to calculate a tile, which is where the recurrences are evaluated. All of the internal mapping functions and validity calculations are calculated using the local iteration variables \( i_k \) instead of the original variables, though the values of the original variables are provided to the user.

F. Determining Tile Dependencies

In order to determine the dependencies between tiles, each template recurrence is examined to determine which boundaries of the tile it could cross over with. For example, the template recurrence \( (1, 0) \) will cause tile \( \tilde{t} \) to depend on \( \tilde{t} + (1, 0) \). One template recurrence can trigger dependencies on more than one tile, such as: \( (1, 1) \), causes dependencies on \( \tilde{t} + (1, 0) \), \( \tilde{t} + (1, 1) \) and \( \tilde{t} + (0, 1) \). A list is created of all these tile dependencies. This list determines which edges need packing and unpacking functions.

G. Template Recurrence Validity Functions

A template recurrence named \( r1 \) will have a corresponding boolean variable named \( is\_valid\_r1 \). This variable is provided so that the user can determine whether the calculation for the current location needs to be modified or use a base case. This calculation is done relative to the current location, with the assumption that the current location is valid. This variable is created by checking each template vector against each original constraint. If a constraint can be violated by adding a particular template vector, then that constraint must be checked before that particular template recurrence can be used. For example, if there exists the constraint \( x_1 + x_2 \leq N \), and the template vector for \( r1 \) is \( (1, 0) \), then this constraint could be violated by trying to access \( r1 \). If there is another dependence named \( r2 \) with the template vector \( (0, 1) \) could also violate this same constraint. In both cases, the inequality to check would be \( x_1 + x_2 + 1 \leq N \). Calculations are reused in cases where two dependencies share some or all of the validity calculations.

H. Mapping Functions

Mapping functions are required for accessing the current location in a loop and the locations its dependencies. The mapping functions of the template recurrences simply have a constant offset from the current location, so the calculations are almost entirely reused. In Figure 3 the values of each \( w_k \) are adjusted when calculating \( loc \) to account for the extra space used by the ghost cell data which stores data from neighboring tiles. In the figure the values are only adjusted by 1, but in general this could vary based on the actual template recurrences of the problem.
I. Packing and Unpacking Functions

Once a tile has been calculated only some of the data near the edge of the tile will be used for the calculation of the following tiles. The packing function takes this data and stores it in a more condensed form until needed for calculation. Packing the data can reduce memory usage. For example, a single edge of the 2-arm bandit problem will use $w^4$ memory where an entire tile would use $w^4$. Each tile outputs 4 edges, so if $w > 4$ then the output of a tile will be less than memory used for calculating the tile. Packing also puts the data in a form that can easily be transferred by MPI. The unpacking function is to reverse the packing function, taking the condensed edge data from one tile and attaching it as ghost cells on another tile so that it can be accessed by the template recurrence mapping functions.

Both unpacking and packing functions use slightly modified versions of the local iteration space of the source tile. Both functions must use the same local iteration space, so that data is unpacking in the correct order and in exactly the same way it was previously packed. The main difference between unpacking and packing, other than the direction of data movement, is the mapping function. Although both functions share the same iteration space, the unpacking function uses the mapping function of the destination tile for unpacking the data into the local $V$ array.

J. Load Balancing Code

The load balancing method used attempts to divide the total amount of work evenly between the nodes along the dimensions that were specified by the user. The user specifies which dimensions $\langle lb_1, lb_2, \ldots, lb_j \rangle$ are eligible for load balancing where $j$ is the number of load balancing dimensions and $j \leq d$. The load balancer attempts to evenly cut up the iteration space by using the highest priority dimension $lb_1$, lesser priority dimensions are used to refine the cut. Figure 2 shows an example of load balancing over two of the three dimensions where each of the tiles are weighted equally. Selecting which dimensions to load balance and their relative priorities is left to the user. We found that good balance can be achieved by selecting less than all the dimensions. However, if not enough dimensions are selected then work might not be balanced well. For example, if Figure 2 had been balanced with only one dimension the balance would have been much worse.

Load balancing has its own iteration space, because not all of the tile indices may have been selected for load balancing. FM-elimination is used to eliminate all non-load-balanced tile indices from the tile space, and is used again when generating the loops for the load balancing code. In general, the load balancing used by the generated code takes $O(n^d)$ time. However, the actual constant is small because the number of tiles is a small fraction of the total number of locations.

In order to assist with the task of load balancing two Ehrhart polynomials are calculated using the Barvinok library [22]. Ehrhart polynomials are polynomials that count the number of integer points inside parameterized polytopes (see [23]). The first Ehrhart polynomial used for load balancing is the one representing the total amount of work the problem has as a function of the input parameters. The second Ehrhart polynomial gives the total amount of work for all tiles with fixed tile indices $t(lb_1, lb_2, \ldots, lb_j)$. These polynomials are generated and written as code so that they can be calculated at runtime when input parameters are known.

Even though this load balancing method does a good job of evenly balancing work, it has a downside. The downside for this method is that it has a tendency to create long critical paths. For example, there is a critical path between the first processor to finish executing, and the calculation of the final tile along the highest priority load balancing dimension. As the problem gets bigger and the number of processors increases, these critical paths get longer. They can cause processors to sit idle, and diminish the speedup of the problem. Improving the load balancing of this program generator is left as future work.

K. Initial Tile Generation Code

In order to generate the initial set of tiles to execute, all tiles with only unsatisfiable dependencies must be found. One way to find these tiles is by examining the tile space and looking for locations such as corners, faces or edges where none of the tile dependencies are satisfied. When it is determined that some set of linear inequalities is violated by the tile dependencies, a new system is created where the offending inequalities are changed to equalities and all the other tile inequalities remain the same. Code is generated for this new system, which is scanned at runtime to look for tiles with no dependencies. This process is repeated for all such possible faces, edges or corners. For most problems, this will probably be a single face, edge or corner of the problem. This process could take $O(n^{d-1})$ in the worst case. Currently, this initial tile generation is executed in serial because it is a small fraction of total run time, typically $< 0.5\%$, for even the largest runs. If the need arises, this could be executed using an OpenMP “parallel for” loop on each node, or overlapped with computation.
for $i_1$ from $ub_1(P, \vec{t})$ to $lb_1(P, \vec{t})$ do
  $x_1 = i_1 + w_1 t_1$
for $i_2$ from $ub_2(P, \vec{t}, i_1)$ to $lb_2(P, \vec{t}, i_1)$ do
  $x_2 = i_2 + w_2 t_2$
  ...
for $i_d$ from $ub_d(P, \vec{t}, i_1, \ldots, i_{d-1})$ to $lb_{d-2}(P, \vec{t}, i_1, \ldots, i_{d-1})$ do
  $x_d = i_d + w_d t_d$
  $loc = (w_1 + 1) i_1 + \ldots + (w_d + 1) i_d + offset$
  $loc_{c+1} = loc + offset$
  {User Defined Statements}
end for
  ...
end for
end for

Fig. 3. Structure of loops inside tiles

L. Tile Calculation Code

Figure 3 shows the layout of the loops for executing each tile. The loops use local iteration variables $i_k$, where $k = 1 \ldots d$, represents the local iteration variables. Each of these local variables correspond to a global iteration variable $x_k$. The value of each $x_k$ is calculated based on the tile width $w_k$, the corresponding tile index $t_k$ and $i_k$. This is provided for use by the user. Each $ub_k, lb_k$ represents the upper and lower bounds of the local iteration variable $i_k$. The bounds are a combination of the global constraints as well as constraints based on the size of the tile. Each $ub_k, lb_k$ is a function of the input parameters, the tile indices $\vec{t}$ and the preceding variables in the loop $i_1, \ldots, i_{k-1}$. In Figure 3 the template vectors of the dependencies are all assumed to be positive. If this was not the case then some or all of the loops would instead iterate from $lb_k$ to $ub_k$.

V. DESCRIPTION OF GENERATED CODE

The code is a combination of generated code for the problem specific code and pre-written libraries for common functions such as communication or memory management. Options such as the number of send and receive buffers and the data type of the state array are adjustable in the generated program.

A. Overview of Generated Program

During the initialization stage there are several actions that are performed. MPI is initialized and the user supplied initialization code is executed. Next, the load balancing routine is executed. After load balancing, the set of initial tiles is generated. The main body of the program consists of a OpenMP parallel section with a while loop that each core executes independently. Tiles are not calculated in a fixed order but rather as all their dependencies are satisfied, with a priority function used to choose among those eligible for execution. Each thread allocates enough memory for a single tile plus additional memory surrounding the tile for the ghost cell data. Only tiles in execution have all their data in memory. Tiles awaiting execution only have their edge data stored in memory. The while loop contains the following steps:

1) Get next available tile
2) Unpack data for tile
3) Execute tile
4) For each valid outgoing edge
   a) Pack the tile edge
   b) Update neighboring tiles or send edge to neighboring processor
5) Add any ready tiles to priority queue
6) Poll for incoming edges if lock available

B. Data structures

There are two main data structures in the generated code. The first one stores all of the pending tiles currently known to the particular node. A tile is only stored if at least one of its dependencies is satisfied. While the total size of the iteration space can be $\Theta(n^d)$, there can only be $O(n^{d-1})$ pending tiles without violating the calculation ordering. Storing only pending tiles reduces the memory usage by an order of magnitude which allows much larger problems to be solved.

Once all of a tile’s dependencies are satisfied, it is moved to a priority queue which holds tiles waiting to be executed. When a thread is ready to execute a new tile, the first tile in the priority queue is selected. The purpose of the priority queue is to control the memory usage pattern. Different execution plans can cause a difference of almost $d$ times the peak memory usage. The reason for this is that when a tile has completed execution its edge data is stored until the corresponding neighboring tiles are ready to be executed. The longer it takes for these neighboring tiles to be executed, the longer this data will need to be buffered. In Figure 4(a), the dark edge represents the amount of memory that is being buffered for each of the tiles which is $5 + 1 = 6$ edges worth. In Figure 4(b) the amount of edges in memory is $2(5 - 1) = 8$. If there had been $n$ tiles along each dimension in Figure 4 instead of 5, the peak edges stored would be $n + 1$ and $2(n - 1)$ respectively. In general, the memory usage for the level-set ordering can be nearly $d$ times the peak memory usage of the column or row major ordering. Each execution method has its advantages. The level set priority method in Figure 4(b) maximizes parallelism at the expense of memory, however this has limited use as the number of tiles that can be executed in parallel is usually a limited quantity. Figure 5 shows the actual priority used in the generated code. We have chosen a priority that prefers column major order. The dimensions that are chosen as highest priority are the ones that are selected for load balancing, leading to tiles that cause communication to execute more quickly. The priority scheme reduces memory usage, and favors communication with neighboring nodes since the load balancing dimensions most likely to result in communication have a higher priority.

From a theoretical point of view, these data structures multiply the run time by $O(\log n)$. However, from a practical point of view, their effect on performance is small because
they will be updated a constant number of times per tile. The total number of tiles that can be executed during a run is $O(n^d)$. However, the constants for the tiles are extremely low, because the $n$ will be divided by the tile width. For example, the polynomial counting the number of tiles in the 3-arm bandit problem has a high order coefficient that is $\frac{1}{2d}$ times the coefficient of the polynomial counting the total number of locations. A tile width of 8 with the 3-arm bandit problem means that there are about $\frac{1}{16}$ as many tiles as locations. Therefore, we do not expect these data structures for storing tiles to become an issue, unless the execution time of a tile becomes significantly shorter, or the problem size grows incredibly large.

VI. Evaluation

We analyzed the performance of 4 dynamic programming problems: 3-string alignment, 2-arm bandit, 3-arm bandit and 2-arm bandit with delay problems. Three string alignment is a 3 dimensional dynamic programming problem with 7 dependencies per location. The 2-arm bandit problem is a 4 dimensional problem with 4 dependencies. The 3-arm bandit problem is a 6 dimensional problem with 6 dependencies. The 2-arm bandit with delay is six dimensional like the 3-arm bandit problem. The reason this problem has 2 more dimensions compared to the non-delay 2-arm bandit is that there is an extra dimension for each arm representing that it has been pulled, but the result is not yet known. Although both the 2-arm bandit with delay and the 3-arm bandit are

6 dimensional, the main difference between the two is the iteration space. The iteration space for the 2-arm bandit with delay defines a relationship between different dimensions. Incrementing the result dimensions requires that the arm-pulled dimension already have been incremented. This problem has an iteration space defined by the polytope:

$$t_1 + t_2 \leq N$$
$$s_1 + f_1 \leq p_1$$
$$s_2 + f_2 \leq p_2$$
$$s_1, f_1, s_2, f_2 \geq 0$$

where $s_k, f_k, t_k$ represents the number of successes, failures and tries on arm $k$.

To measure the performance of our program generator we decided to test the strong scaling of the shared memory part of the code on a single node of 24 cores. For testing performance across nodes, we test weak scaling by approximately fixing the amount of work per node, and scale the problem size according to the number of nodes. The system used for the timings was made up of nodes with 32 GB of ram each and dual 12-core AMD Magny Cours CPUs.

A. Shared Memory Scaling

We found that for all the problems we tested, the generated programs achieved a 21.8 speedup or better on 24 cores. Figure 6 shows the scaling curves for each of the 4 problems we tested. In order to determine what is causing the most inefficiency we timed the individual parts of the 2-arm bandit problem. The results of the timing are divided into 4 categories: the total calculation time, total packing time, total unpacking time, and any other overhead. The percent of time spent on each task in terms of the number of cores used is shown in Table I.

B. Weak Scaling across MPI

Figure 7 shows the weak scaling efficiency from 1 to 8 nodes of 24 cores each. For 1 node the problem sizes are the
same as were used for the shared memory scaling results. For more than one node, problem sizes are scaled up, so that the number of locations per node stays about the same. For most of these problems it is not possible to exactly scale up the total amount of work so the time is normalized by the actual number of locations before calculating the efficiency.

Overall, the scaling out to 8 nodes is fairly good for most of the problems. On 8 nodes (192 cores) the efficiency of the 2-arm bandit problem is about 90% compared to 1 node. The speedup on 1 node for the 2-arm bandit is 22.35 with 24 cores. Overall, this gives a combined efficiency of about 84% for 192 cores.

C. User configurable options affecting performance

User configurable options can have a significant impact on performance. The options that most affected performance were the tile size, the number of send and receive buffers, and the dimensions chosen for load balancing. The optimal settings for these options vary, so that finding the correct values for these options is not trivial, and would require a parameter sweep in order to find the best values.

For some problems the tile size can have a huge effect on the performance, especially if the tile size is very large. This is due to the pipelined nature of the load balancing algorithm used. A large tile can cause starvation while neighboring nodes wait for data. This starvation will cause a compounding of delays for each processor in the chain. For the 3-arm bandit a large tile width of 15 allowed better throughput for 4 nodes or less, however for larger numbers of nodes, the lengthy calculation time caused massive delays. We found that the tile width 7 allowed the nodes to communicate more frequently, thereby reducing the delays substantially. We believe this change can be partially explained by the critical path between when the first processor to complete all its tiles and the last processor to complete its tiles. With a tile size of 15, the maximum time to calculate a tile was approximately $\frac{1}{2}$ of a second. If $n = 256$, then the final edge will contain 9 tiles. If 7 of these are entirely full then the critical path is over 5.25 seconds, not including communication latency. For a tile width of 7 this critical path cannot take more than $\approx 0.58$ seconds. However, the smaller tile width increases the single node execution time significantly because it increases the frequency and amount of time spent packing and unpacking. Further, this part of the code was observed to not scale as well as the calculation code.

The number of send and receive buffers has an effect on performance. Not having enough receive buffers will decrease the speed in which data is being received which in turn can cause starvation. Our program enforces a limit on how many pending sends can be issued at a time. Once this limit is reached, then no more sends can be issued until a previous send completes. The number of send and receive buffers can also have a considerable effect on memory usage, especially if the edges of the tiles are large, or there is a large number of buffers.

The number of dimensions used for load balancing can have a large effect on performance. For problems such as the 3-arm bandit, selecting too few dimensions limits the effectiveness of load balancing which reduces the scaling of the program. The choice and priorities of the load balancing dimensions can also have an impact on performance.

VII. Future Work

A. Recovering Solutions

One issue not yet addressed is saving the entire iteration space for further calculation. For example, this can be used to do a traceback to arrive at a solution of a dynamic programming problem, as opposed to merely knowing the optimal value of the objective function. Future support for this could be added fairly easily. If the user needs the actual values in each location, then the edges of the tiles could be saved, and needed tiles recalculated on the fly during the traceback. If only the decisions are required then a run length encoded representation of the decision matrix might be acceptable.

B. Improve Load Balancing

An area of future work for this project is to develop a more general load balancing method. The load balancing method used has a tendency to create long critical paths. It is impossible to completely eliminate critical paths. However, it should be possible to reduce their effect. One method that seems to improve the load balancing of wedge shaped iteration spaces divides the work using hyper-planes. Figure 8 shows a load balancing over 2 dimensions of a 3-dimensional iteration space. The 3rd dimension is represented as weights of each
2d tile. Load balancing is done by finding planes that evenly divides the work between processors. When using this load balancing on the 2-arm bandit problem idle times were reduced when scaling across nodes. This load balancing method has not yet been incorporated into program generator.

C. Improve Shared Memory Code

There some things that can be addressed to improve the performance of the shared memory code. Many multiprocessor systems have Non-Uniform Memory Access (NUMA) which means the access time for system memory varies depending on its distance from a core. Hence one improvement would be to redesign the shared memory program to be more NUMA-aware. One might obtain performance improvement by prioritizing execution of tiles with edges nearby to a specific core. For systems with large numbers of cores, contention for the shared data structures may become a bottleneck, though it has not yet been so. This could be addressed by using separate shared data structures for groups of closely connected cores. As long as its own queue has work, a core would not need to compete for locks outside its group.

VIII. Conclusion

We have presented an automatic program generator for a useful class of recursive functions. This generator has the potential to benefit some scientists by greatly simplifying the use of parallel computers to solve large problems. The generated code has obtained good efficiency running on a generated code on up to 8 24-core nodes using OpenMP achieving speedup of 84% on 8 nodes (192 cores).

We believe there is more work that can be done in this area, including extending this basic idea to other architectures such automatic program generation for GPGPUs. Which would give the user the flexibility to use whatever system or resources they have available. Another area of future work is to make program generators for problems with recurrences that do not have a fixed template. For example, the standard recurrences for finding optimal binary search trees do not have a fixed template.

REFERENCES