Cooperative Parallelization

Praveen Yedlapalli, Emre Kultursay and Mahmut T. Kandemir
The Pennsylvania State University
{praveen, euk139, kandemir}@cse.psu.edu

Abstract—We propose a cooperation between the programmer, the compiler and the runtime system to identify, exploit and efficiently exercise the parallelism available in many pointer based applications. Our parallelization strategy, called Cooperative Parallelization, is driven by programmer directives as well as runtime information. We show that minimal information from the programmer can be combined with runtime information to extract latent parallelism in many pointer intensive applications that involve trees and linked lists. We implemented a compilation framework which automatically parallelizes programs annotated with parallelism directives. We evaluated our approach on a collection of linked list and tree based applications. Our results show that we can achieve speedups of up to 15x on a sixteen-core platform. We also compared our approach to OpenMP both qualitatively and quantitatively.

I. INTRODUCTION

An industry wide transition from single-core to multicore processors [15], [16], [1] has brought the problem of program parallelization to forefront. Parallelizing compilers can offer some help through automatic parallelization of programs. Automatic parallelization has been a well studied problem [17], [30], [4], [5], which can be used as a solution to reduce the cost and time involved in developing parallel programs. Moreover, it also helps in reducing the time spent in parallelizing existing programs.

Program parallelization is a two-fold problem. The first problem is to find where parallelism is available in the application. The second problem is to decide on how to efficiently exploit the available parallelism. The first problem is typically addressed by static analysis of the code to discover data dependencies [18], [19], [12], [20], [30]. Dependence analysis is typically used to decide whether iterations of a loop can be executed in parallel. Most of the existing analysis techniques target regular structures such as arrays and for loops that are used to iterate over them. Although these structures frequently arise in many scientific and embedded multimedia programs, most general purpose programs involve pointer based data structures, recursion and while loops. The fundamental difference of a while loop is that, its iteration count is not known at compile time and typically changes with input. Further, such loops typically include pointers which obstruct most existing compilers in analyzing and parallelizing these loops. The difficulty introduced with the use of pointers is that, they refer to dynamic entities which carry meaning at runtime. At runtime, a pointer can point to any variable (within its scope) which cannot be determined exactly by static analysis at compile time without sophisticated pointer analysis techniques. Even when such pointer analysis techniques [8], [13] are employed, successful discovery of latent parallelism is not guaranteed. Therefore, most compilers do not analyze dependencies between statements involving pointers. The second problem, i.e., efficient implementation of parallelization, is also not trivial in case of pointer-based applications. In a regular program, for loops working on arrays can be parallelized by distributing the iterations of the loop to different threads. In case of pointer based applications, such a straightforward approach cannot be taken as the work to be distributed across processors is not identified yet. The work distribution should be done in an intelligent way considering runtime information along with knowledge about the data structure. Therefore, most compilers conservatively treat pointer based loops as sequential loops.

One key observation is that, the developer of an application has a higher level view of the program than the compiler and is more likely to know which sections of the code are independent and can run in parallel. In particular, it is relatively easier for the programmer to identify simple information that is vital in parallelization of applications. In the past, there have been various efforts to parallelize programs using compiler directives. OpenMP [3] is one example where the user provides compiler directives to parallelize programs. However, in many cases, user level annotations may not be sufficient for exploiting available parallelism and we need runtime information as well. In this work, we delegate the task of identifying parallelism in the application to the developer and leverage this information to exploit the parallelism in the program. More specifically, we show that simple information provided by the programmer as hints coupled with runtime information can lead to significant parallelism opportunities in applications which are otherwise not parallelizable. In addition, we use the programmer provided information about the application to efficiently distribute the work among the threads of the application.

Conceptually our scheme can be depicted as shown in Fig. 1(a). The figure shows three sources of inputs to our parallelization strategy. The first one is from the programmer who provides application-level parallelism information (in the form of hints) which cannot be extracted by the compiler. These hints indicate which sections of the code are independent and can be parallelized. The second input is from the compiler which transforms the sequential code to parallel code using the information provided by the programmer. The compiler builds the parallel version of the program along with the appropriate runtime monitoring system based on the programmer hints. The third input comes from the runtime system which is used to determine how to run the code in parallel. This system is responsible for determining whether it is profitable to use the parallel version of the code for the current input and distributing the work among different threads of the application.

We can summarize the major contributions of this work as follows:

- We describe an interface between the programmer and the compiler such that the compiler can make use of the high level information (hints) provided by the programmer.
- We present an automated code parallelization framework comprising a source-to-source translator which takes a user-annotated program and creates an equivalent parallel program using pthreads.
- We discuss a helper-thread based method to identify the runtime information needed for parallel execution.
- We present experimental evaluation of the proposed approach and show that it brings significant speedups on pointer based applications which cannot be easily parallelized. The speedups we achieve for some applications is around 15x on a sixteen-core platform.

II. COMPARISON AGAINST CURRENT APPROACHES

A. OpenMP

The latest release of OpenMP [3] has support for task parallelism. This can be used to parallelize while loops with linked list traversals.
B. Speculative Parallelization

There have been two approaches in the literature that employ speculation in the context of code parallelization. The first approach, namely, memory alias speculation [28], [7], [10], assumes (optimistically) that the addresses accessed by different threads do not overlap. The second approach that employs speculation for parallelization is value speculation [22], [21], which predicts the values needed in future iterations and execute those iterations in parallel speculatively. It should be noted that speculative parallelization may suffer from a high misspeculation rate in some cases, which can result in the parallel code performing worse than the sequential code. Our work is a non-speculative approach to code parallelization. In our approach, we never roll back computations, and there is no need to maintain a separate program state and update it. Moreover, as will be discussed later in detail, we employ techniques to hide the performance overheads incurred in our approach.

C. Other Relevant Efforts

In [6], the authors show that simple hints from the programmer can be used to extend the already existing parallelization techniques like DSWP (Decoupled Software Pipelining). They take hints from the programmer which indicate the properties of the code. They leverage obtained information along with other techniques like DSWP and speculation and generate parallel code. This paradigm does not involve runtime information about the program variables or state to schedule the threads efficiently.

In [23], the authors propose inspector/executor model for identifying parallelism at runtime. They run an inspector loop ahead of the original loop that analyzes the cross-iteration dependencies in the loop and schedules the loop iterations using that information. The executor executes the code according to the schedule determined. Those techniques mostly target array-based applications with indirect references. We use runtime information to partition the computation into threads by identifying independent sub-problems. Further, we move the overheads of inspector to the helper-thread.

Cilk [11] is another language extension to express parallelism in programs. Work stealing is used to divide the program execution among different threads. Multilisp [14] is a variant of the Lisp programming language with constructs for parallel execution. Programmer can specify which expressions in the code are independent with a future construct which are evaluated in parallel. Neither of them consider programmer directives to schedule tasks at runtime. The works by Rus et al. [27], [26] introduce a technique for automatic parallelization by combining static and dynamic analysis of programs. In their work, the authors extract conditions for parallel execution by static analysis and use them to guard the dynamic parallelization of loops at runtime. The LRPD Test in [24] and the R-LRPD Test in [9] are techniques for automatic parallelization using only runtime analysis. In these works, target loops are executed speculatively in parallel and tested for memory dependencies at runtime. These approaches focus on using runtime information to parallelize array based applications. Rogers et al. [25] present an execution model for supporting programs that use pointer based dynamic data structures on distributed memory systems. They address the issues of data placement in such applications on a distributed memory platform. Our work, in comparison, targets shared memory machines where data layout and data migration issues can be safely ignored.

III. DETAILS OF COOPERATIVE PARALLELIZATION

A. Generic Data Structure

Two typical characteristics of applications working on dynamic data structures are, there is (i) a data structure that is constructed from the input and (ii) a function which traverses the data structure and performs a computation on its nodes. Frequently, the computations done by the function on different parts of the data structure are independent. In most cases, the fact that they are independent can be obtained from the programmer. In the absence of a parallelization
mechanism, the computations on different parts of the data structure are done sequentially using recursive function calls or while loops. After the computation on a node, called a subproblem, is performed, the function/loop finds new subproblems and continues its execution by moving on to these subproblems. However, if the function has access to multiple subproblems before starting the initial computation, it can potentially initiate processing of different parts of the problem independently in parallel. To achieve this, before starting the computation on the data structure, our approach invokes a helper thread which goes over the data structure and finds multiple independent subproblems in the data structure. It is important to note that the subproblems are runtime entities and cannot be identified at compile time. Then, the function/loop gets access to multiple subproblems and can start working on them in parallel.

The important observation behind our approach is that the task of finding subproblems in a data structure does not need (in general) the traversal of the whole data structure. Moreover, it does not involve any computations on the nodes. As a result, it takes significantly less time than the main computation.1

The motivation for using a separate thread to identify the subproblems is to reduce the overhead introduced by searching the data structure to identify subproblems. The main thread can invoke the helper thread before arriving at the parallel section and proceeds with its work. While the helper thread identifies the subproblems, the main thread can execute the code before the parallel section. Note that, the main thread has to make sure that the target data structure is not modified once the helper thread is invoked until the computation in the parallel section is started. Currently we are finding this information manually but it can also be obtained from the programmer in the form of a directive. A high level graphical illustration of cooperative parallelization is given in Fig. 1(c).

1 The details of the times taken by subproblem identification in different applications will be given in Section V.

Fig. 2. Codes for different functions in the applications.

Fig. 3. Subproblems in a tree-based application.

The important observations behind our approach is that the task of finding subproblems in a data structure does not need (in general) the traversal of the whole data structure. Moreover, it does not involve any computations on the nodes. As a result, it takes significantly less time than the main computation.1

The motivation for using a separate thread to identify the subproblems is to reduce the overhead introduced by searching the data structure to identify subproblems. The main thread can invoke the helper thread before arriving at the parallel section and proceeds with its work. While the helper thread identifies the subproblems, the main thread can execute the code before the parallel section. Note that, the main thread has to make sure that the target data structure is not modified once the helper thread is invoked until the computation in the parallel section is started. Currently we are finding this information manually but it can also be obtained from the programmer in the form of a directive. A high level graphical illustration of cooperative parallelization is given in Fig. 1(c). The solid arrows represent the control flow of the program and the dashed arrows represent the communication between threads. The left hand side of the figure illustrates the sequential execution of a program with a loop to be parallelized. On the right hand side, parallel execution of the same program using our approach is given. With the help of runtime information from the helper thread, the main thread is able to parallelize the loop. Note that, in the parallel execution, the main thread continues its work while the helper thread identifies the subproblems in the program.

Fig 2(a) shows the high level code for the proposed execution model. The main thread signals the helper thread when the data structure becomes ready. The helper thread, waiting for the signal, receives the signal, and starts its job of finding independent subproblems in the data structure. After the helper thread’s job is complete, it signals the main thread to notify the completion of finding of the subproblems. At this point, the main thread has access to multiple subproblems, and distributes these tasks to different application (worker) threads and signals them. The application threads that are waiting for a signal from the main thread start working on their part of the data structure. Once their jobs are complete, they signal back to the main thread individually. The main thread, waiting for these signals, proceeds to merge the results from the individual threads.

Another use of collecting runtime information is to determine the profitability of parallelization. The helper thread can determine the size of the data structure while traversing it for subproblems. This information can be used by main thread to decide whether to parallelize the section of code or not. This decision can be driven by the programmer in the directive. Only if the data structure is big enough, the computations on that data structure are parallelized.

B. Trees and Recursion

In this section, we describe how our approach handles tree based applications with recursion. Typically, in a tree based application, a tree data structure is constructed using the input and a function goes over the nodes of the tree and performs a computation at each node. This function frequently is a recursive one which is initially called on the root of the tree. The function performs some computation at the root node and then calls itself on each of its child nodes. Once the processing of the children is done, the function merges the results obtained from the function calls on the child nodes. Note that, the function calls on child nodes are calls to the same function, i.e., they are recursive function calls. Fig. 3 illustrates the process where the function called on the root node recursively calls itself to process the left subtree and then the right subtree.
The key observation in the described scenario is that, the function calls on different subtrees can be independent. If this is the case, then the function need not wait for the call on the first child to complete in order to proceed with the next child. In this scenario, the only independence is in merging the results obtained from the recursive calls which will be performed after all calls are completed. Therefore, the potential parallelism can be exploited by executing the function calls on child nodes in parallel. However, in order to launch these functions as threads executing in parallel, the contexts in which these functions will be called are needed. In this work, a context is defined as the set of parameters that fully describe the environment in which the parallel threads will be launched. In the tree example, if two threads will be launched for the two subtrees shown in Fig. 3, then the pointers to the roots of the two subtrees are required to start the threads, and these pointers constitute the context. Note that, in this binary tree, at a specific instance of the function, the target node has direct access only to its immediate children. As a result, the function can start a number of parallel tasks up to the number of children of its node. In order to be able to start more threads, deeper information over the tree is necessary.

As an example, consider the application perimeter, from the Olden benchmark suite [25], which works on a quad tree where each node has four children. The perimeter function takes two parameters: a pointer to the root node of a tree and the size of the tree, as shown in Fig. 2(b). The function first checks whether the target node is a leaf node. If this is the case, it performs a computation on that node; otherwise, it calls itself recursively on the child nodes (represented as nw, ne, sw, se fields) of the target node and half the size. The function adds the return values of all of recursive calls to compute the return value of the function.

Note that, in this application, the calls to the perimeter function on sibling nodes are independent and can be run in parallel. To parallelize such an application the main thread needs to know the contexts in which the functions on child nodes are called. In this case, a function context comprises a node pointer and an integer. The main thread invokes a helper thread to identify the contexts. The helper thread starts at the root of the tree and finds the pointers to the four children along with the correct size values for each child node. Next, it finds the height of the tree. Then, it returns these contexts and the height to the main thread which compares it to the programmer provided height requirement. If the current tree satisfies the minimum height requirement (which is a part of our hints), it initiates the application threads using the contexts. Otherwise, the main thread does the computations in a serial manner. The application threads have identical code as the original sequential program. Note that, if the application is to be parallelized with eight threads, then the helper thread goes one level deeper in the tree and identifies 16 second level pointers and size values. Then, the main thread must distribute the 16 tasks identified by the helper thread across the 8 application threads in a balanced fashion.

C. Linked Lists and Loops

In this section, we describe how our approach can be used to parallelize programs that use linked lists. In a typical linked list based application, once a linked list is constructed from the input, functions that operate on the list traverse it sequentially using a while loop and perform computations on each node. If the computations performed on each node are independent, then the iterations of the loop are independent. In this case, as shown in Fig. 4, the problem can be divided into subproblems, which are defined on the sublists of the original list. If the function has access to the sublists of the list before starting the loop, then it can start working on individual sublists in parallel. After the computations on the sublists are completed, the function can merge the results from the individual sublists to generate the final result.

Identification of the sublists is done by the helper thread which is invoked by the function before starting the loop. Once the helper thread completes its task, the function gains access to multiple sublists, and as a result, instead of executing the loop sequentially, it distributes the work on individual sublists to different application threads. The application threads work on the sublists sequentially assigned to them and compute the local result from those sublists. Once all the application threads complete, the main thread then merges the results obtained from the individual sublists.

To further concretize the idea, consider the em3d application which operates on a singly linked list. The function compute nodes, shown in Fig. 2(c), contains a while loop which goes over the entire list and updates each node. Note that, the updates at a node are independent of the updates at all other nodes. To parallelize such a function, one needs to know the sublists in the linked list before starting the loop. Each sublist is defined by pointers to the start and end nodes in different parts of the linked list. The task of finding these pointers is carried out by the helper thread, which traverses the list and saves the pointers at particular intervals. Along with them, the helper thread determines the length of the linked list. Note that, the helper thread does not perform any computation but just traverses the list, which is why it can progress much faster than the main thread. Once the helper thread completes, the main thread gains access to different sublists and the length of the original list. Based on the programmer specified minimum length requirement the sublists are either distributed to different application threads, or alternatively, the main thread computes the solution sequentially by itself. The application threads perform a function similar to the original (serial) loop with the exception that they work on the sublists defined by the start and end nodes given by the main thread.

D. Discussion

We would like to emphasize that, pure static analysis based techniques cannot exploit the parallelism available in the applications described above. The lack of this capability is due to two reasons. The first problem is with the pointers being used in these programs. As discussed, the problem with pointers is that, they are dynamic entities which have meaning only at runtime and cannot be analyzed accurately at compile time. Specifically, it is very difficult for a compiler to determine which location a particular pointer is pointing to. As a result, compile time techniques cannot prove (in)dependence between program statements, and therefore, cannot discover the available parallelism. We address this problem by taking the dependence information from the programmer. The second problem with compile time analysis techniques is that, even after determining that two statements are independent, in order to execute them in parallel, we need runtime information which captures the dynamic characteristics of the program. For example, in a linked list-based application, the program needs the starting addresses of the sublists in order to start
working on them in parallel. We address this problem by running a helper thread along with the application which determines the runtime information needed to execute the code in parallel.

Clearly, our approach has the overheads of running helper thread and communication between threads. However, it should be emphasized that the helper thread in many cases does much smaller amount of work compared to the work done by the application. For example, in a tree-based application, the helper thread just traverses the top two or three levels of the tree (depending on how many parallel threads we want to spawn). The communication cost between threads is also kept minimal because of the use of shared cache based chip multiprocessors as our underlying platform. Overall, the overheads we incur are negligible compared to the savings we accomplish. A quantitative analysis of the overheads incurred by our approach is given in Section V.

IV. PROGRAMMER HINTS AND AUTOMATION

Our automation framework consists of two components: (i) an effective way of expressing parallelism in the application and (ii) a method to generate parallel code from the sequential code automatically. We implement the first component by designing simple hints that are annotated in the target (sequential) program code. The second component is realized by a source-to-source compiler that converts the user annotated program code to parallel code.

A. Programmer Hints

The first problem in parallelizing applications is the identification of independent tasks in a program. In regular programs that mostly use arrays and for loops, this can be performed through data dependence analysis [18], [19], [12], [20], [30]. However, in an irregular program involving pointer-accessed dynamic data structures, recursion, and while loops, it is a very difficult task to identify data dependencies automatically. Instead of performing alias analysis, we take an alternative approach and obtain the parallelism information from the programmer. In order to capture this information from the programmer, we need well defined programmer hints that express parallelism in the program. The design goals behind these hints are that, they should be compact with minimal essential information and simple enough to be easily used. An important information we need from the hints is the minimum threshold to parallelize the code. This information will help in deciding at runtime whether a particular code section is profitable to parallelize or not. Another use of the hints is to dictate the features of the parallel program such as the number of threads. We propose two types of hints, one for tree based applications and one for linked list based applications, which are shown in Fig. 5.

The first two fields in the first line of figure indicate that the directive is a parallelization directive and the program operates over a tree structure. This information is used to select the type of runtime system (helper thread) to build for the application. The function field indicates the name of the function to be parallelized and the threads field indicates the number of threads that will be created in the parallel version of the code. This field gives the user the capability to decide on the appropriate number of threads in the parallel program considering the number of processing elements available. The degree field indicates the number of children that each node in the tree has, which is used to customize the helper thread according to the specific tree structure used in the program. The structure and child_nodes fields indicate the name of the structure and the names of the child nodes in the tree structure, which are used to identify the subtrees that will be searched to detect parallel tasks. The threshold field indicates the minimum height of the tree for the parallelization to happen. This field gives the programmer the ability to dictate when it is profitable to parallelize and when it is not. This field is used at runtime to compare with the height of the current tree in the program. And make a decision whether to parallelize or not. Finally, the reduction field indicates the field of the tree node on which the function performs its computation. This field is optional and is required only when the function is accumulating results from individual subtrees.

Fig. 5 also shows our second type of directive. This directive is used to express parallelism in linked list-based applications. The llist field indicates that the program to be parallelized is a linked list-based application. The parameter structure defines the name of the structure corresponding to a node of the linked list. The next_node, on the other hand, identifies the field of a linked list node that points to the next node in the list. threshold indicates the minimum number of nodes in the list to parallelize the loop. The estimated number of nodes in the linked list is given by the number field, which is an optional field that in turn determines the size of the sublists the linked list will be divided into. Along with this directive, the programmer also provides another directive (#parallel) just before the loop which is to be parallelized. This additional directive is used to identify the target loops in the function.

B. Automation

Once the code is annotated with the programmer hints described in the previous section, these hints need to be parsed and leveraged to generate the parallel program. In order to achieve this, we implemented a source-to-source translator that takes the original program with parallelism hints as input, and generates a parallel version of the program that receives vital parallelization information from the helper thread attached to it. Source-to-source translation is actually performed while parsing the high level source code of the original program. For this purpose, we modified the C language grammar to include the programmer hints and provided this grammar to a parser generator. The translator generated from the parser generator parses the source code of the program and creates the parallel program while doing so. Fig. 6 shows the high level view of this process. When the parser detects a parallelization directive, it creates the functions for helper thread, application threads, and the modified parallel function.

The code for the helper thread is identical for all tree based applications except for the slight differences caused by the differences in the degrees of the trees. The application thread code is similar to the original function, which recursively calls itself over all child nodes. In some cases, the application thread has to work on multiple contexts assigned to it by the main thread. Note that, the application threads should call the original (sequential) function to perform the computations on the child nodes. The original function is modified to invoke the helper thread, receive the context information from the helper thread, distribute these contexts to different application threads, and merge their results. A parallel function in a tree based application should perform these operations. The code in the parallelized function is similar for all the tree based applications except for the merging part. The parallel code generated for the perimeter application [25] using our translator is shown in Fig. 7. The original code for the

\[\text{#parallel tree function (threads) (degree) (structure) (child_nodes) threshold reduction}\]

\[\text{#parallel list function (threads) (structure) (next_node) threshold number}\]
The suite consists of nine applications which are based on trees with sixteen core processor. The details of the platform used in our experiments are provided in Table I.

We used the Olden benchmark suite [25] to evaluate our approach. The suite consists of nine applications which are based on trees and linked lists, and are developed in C programming language. We identified the parallelism available in each application and added the appropriate hints (described earlier in Sec. IV-A) to the original programs. These modified programs are processed using the translator described in the previous section, which understands the hints and automatically generates the parallel C programs of the type shown in Fig. 7. In our experiments, we pinned each thread of the parallelized application to a different core in the underlying platform. We focused on the core functions in each benchmark and measured the running times of these functions. The contributions of these functions to overall execution times of the target applications are also given in columns five and six of the same table, respectively. Note that, the higher the execution time fraction, the better overall speedup over the whole application when that function is parallelized.

We identified the parallelism available in each application and added the appropriate hints (described earlier in Sec. IV-A) to the original programs. These modified programs are processed using the translator described in the previous section, which understands the hints and automatically generates the parallel C programs of the type shown in Fig. 7. In our experiments, we pinned each thread of the parallelized application to a different core in the underlying platform. We focused on the core functions in each benchmark and measured the running times of these functions. The contributions of these functions to overall execution times of the target applications are also given in columns five and six of the same table, respectively. Note that, the higher the execution time fraction, the better overall speedup over the whole application when that function is parallelized.
corresponding application when parallelizing it using our approach and with OpenMP. In all the graphs, the x-axis represents the number of threads (one thread per core), and the y-axis gives the speedups obtained. In most of the applications, we obtained almost linear speedups. The function find_lightest_cl in otter application traverses the linked list and finds the node with minimum weight. We could not parallelize that function (given in Fig. 9(c)) with OpenMP because of the lack of support for task reduction in OpenMP. The speedup in bisort is not significant because of the costly merge operations occurring after the parallel computations. Similarly, the speedup in otter is not significant due to the overheads in our approach. These overheads are discussed in detail in the next section.

One observation we made in these experiments is that, parallelizing with OpenMP tasks is not straightforward in many of the applications considered. Just putting an OpenMP directive before a recursive function call or a while loop might not yield any speedup. In fact, the parallelized programs can perform worse than the serial programs. We believe, the reason for this is the finer granularity of the tasks. Each recursive function call or each iteration of a loop is treated as a separate task with its own code and environment. Creating them and deploying them causes an overhead in the performance of the parallel application. The developers of OpenMP propose a patch for such cases, by manual insertion of an if statement to create tasks based on the size of the input. However, this does not work in an application working on dynamic data structures like trees and linked lists. In tree based applications, we inserted extra code in the program to track the level of the node and stop creating tasks after reaching a particular level in the tree. In list based applications, we could not find a way to control the granularity of the tasks. This resulted in suboptimal speedups from OpenMP in list based applications (Fig. 9).

Our approach does not suffer from this drawback since we first find the optimal number of independent tasks and work on them in a serial manner in different threads. Moreover, our approach provides the programmer with the ability to control the profit of parallelization with the threshold size. As a result, we obtain better speedups in all the applications compared to OpenMP versions.

Although our technique can handle the generation of parallel code, an exception in our implementation is that, in programs with complex merge (reduction) operations, it cannot produce accurate code to merge the results from different application threads. If the program is just finding the sum or product of the results from individual threads, that is automated in our translator. The difficulty in handling complex merge operations is that, in a recursive environment, the merges in deeper levels of recursion should happen before the merges in higher levels. Since our approach, in a sense, unrolls the recursion, we need a mechanism to determine the order in which the results are to be merged. This feature is not yet fully automated in our translator. Instead, in our current implementation, we generate a warning that recommends the user of the tool to examine the generated code specifically at the merging part and perform any required modifications manually. We believe that, this does not arise frequently in general programs. Even in cases it arises, the programmer should be able to easily add the code needed for the merging. In the seven benchmarks we tested our approach, three of them (bisort, tsp, and otter) required slight modifications in the merging part.

C. Overheads

The approach we proposed has the overheads of running a helper thread and synchronization between threads. In each application, the helper thread has to go over either a part of or the whole data structure solely with the purpose of finding the subproblems without doing any real work. Assuming that there are enough cores to run all the threads, executing the helper thread on a separate core can reduce this overhead. In this case, the main thread can invoke the helper thread before arriving at the parallel section and continue its work. The assumption here is that, by the time the main thread arrives at the parallel section, the helper thread would be finished with its task, effectively hiding the overhead of the task of understanding the program state. In any case, the main thread has to wait for the helper thread to complete before starting the application threads. The running times of the helper thread in each application considered in this work are shown in Table III. The second column in the table gives the time taken by helper thread working in a parallel application with two threads. The third column gives the time taken by the original function, and the fourth column shows the percentage overhead introduced by the helper thread. These results show that the helper thread overheads are negligible in the tree-based applications (the first four rows). This is because the work done in the helper thread of a tree based application is just finding the pointers to the nodes at top two or three levels of the tree, which is fairly simple. In a linked list based application, the helper thread has to traverse at least half of the list to find enough number of sublists. This results in higher helper thread overheads.

The helper thread overhead may play a significant role in the maximum achievable speedup for an application. Consider otter, for example, which is based on linked-lists. The function find_lightest_cl in this application contains a while loop which traverses the entire list and finds the minimum element in the list. The work done in a single iteration of the loop is just comparing the current minimum with the value in the current node. In the parallel version of this application, the helper thread has to traverse most of the list to find the sublists. The work done in the original loop and that in the loop in the helper thread are comparable. As a result, the helper thread takes almost 30% of the time compared to the original loop. Note that, the helper thread contains the sequential part of the parallelized application. Using Amdahl’s law, we can derive an upper bound on the achievable speedup. In the case of otter, this is 3.33 which is reflected in the results we obtained, as shown in Fig. 9(c).

VI. POSSIBLE EXTENSIONS AND FUTURE WORK

In this work, we performed an exploration of an efficient parallelization scheme for dynamic data structures. We do not claim to have a comprehensive solution that can parallelize all dynamic data structures and all usage patterns. We pointed out the benefits brought in by using programmer hints and runtime information in parallelizing applications containing these difficult-to-parallelize structures. We demonstrated the applicability of our technique to linked lists and

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Data Structure</th>
<th>Input Size</th>
<th>Core Function</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>bisort</td>
<td>Bitonic Sorting</td>
<td>Binary Tree</td>
<td>100,000 numbers</td>
<td>bisort</td>
<td>50%</td>
</tr>
<tr>
<td>treeAdd</td>
<td>Adding Numbers in Nodes</td>
<td>Binary Tree</td>
<td>22 levels</td>
<td>treeAdd</td>
<td>57%</td>
</tr>
<tr>
<td>tsp</td>
<td>Traveling Salesman</td>
<td>Binary Tree</td>
<td>10,000 cities</td>
<td>tsp</td>
<td>90%</td>
</tr>
<tr>
<td>perimeter</td>
<td>Perimeters of Regions in Images</td>
<td>Quad Tree</td>
<td>11 levels</td>
<td>perimeter</td>
<td>65%</td>
</tr>
<tr>
<td>em3d</td>
<td>Electron Microscope Tomography</td>
<td>Singly Linked List</td>
<td>10,000 n_nodes &amp; 100 d_nodes</td>
<td>compute_nodes</td>
<td>83%</td>
</tr>
<tr>
<td>mst</td>
<td>Minimum Spanning Tree</td>
<td>Singly Linked List</td>
<td>4096 nodes</td>
<td>BlueKnife</td>
<td>91%</td>
</tr>
<tr>
<td>otter</td>
<td>Theorem Proving Software</td>
<td>Singly Linked List</td>
<td>twoval.in</td>
<td>find_lightest_cl</td>
<td>20%</td>
</tr>
</tbody>
</table>

TABLE II

BENCHMARK APPLICATIONS AND THEIR IMPORTANT PROPERTIES.
trees, yet it can as well be generalized to support other dynamic data structures. For instance, although the syntax given in Fig. 5 can only handle trees with statically-known number of children, a straightforward extension can also handle trees with variable number of children that will be determined at runtime. Further, the duties of helper thread need not be restricted to the analysis of the target data structures. Instead, during idle times, the helper thread may query system performance metrics such as CPU loads or cache hit rates to reason about system-level behavior. This system-level information can then be used to determine the optimum number of threads to spawn for the target parallel loops.

VII. CONCLUSIONS

We propose a semi-automatic parallelization strategy through the cooperation between programmer, compiler and the runtime system. We show how such an approach can parallelize pointer intensive application involving dynamic data structures such as trees and linked lists. The parallelism decisions are driven by hints provided by the programmer and the runtime information. We present compact and simple programmer hints to express parallelism in the programs. We implemented a source-to-source translator which understands the hints embedded in the sequential C programs and generates the equivalent parallel programs. We compare our approach to OpenMP both qualitatively and quantitatively. We show significant speedups for the generated parallel programs compared to the original programs on a multicore machine.

REFERENCES