AutopaR: An Automatic Parallelization Tool for Recursive Calls

Mert Emin Kalender, Cem Mergenci, Ozcan Ozturk
Department of Computer Engineering
Bilkent University, Ankara, Turkey

***Abstract*—Manycore systems are becoming more and more powerful with the integration of hundreds of cores on a single chip. However, writing parallel programs on these manycore systems has become a problem since the amount of available parallel tools and applications are limited. Although exploiting parallelism in software is possible, it requires different design decisions, significant programmer effort and is error prone. Different libraries and tools try to make the transition to parallelism easier, however there is no concrete system to make it transparent to software developer. To this end, our proposed tool is a step forward to improve the current state. Our approach, Autopar, specifically aims at achieving automatic parallelization of recursive applications using static program analysis. It first decides on the recursive functions of a given program. Then, it performs analysis and collects information about these recursive functions. Our analysis module automatically collects program information without requiring any modification in the program design or developer involvement. Finally, it achieves automatic parallelization by introducing necessary OpenMP pragmas in appropriate places in the application.**

***Keywords*-parallel; recursive; manycore; automatic**

I. Introduction

As technology scales, the International Technology Roadmap for Semiconductors projects that the number of cores will drastically increase to satisfy performance requirements [1]. However, using these manycore architectures effectively is not an easy task as there are limitations in software development. Since manycore systems exploit parallel threads, we must find ways to develop parallel applications or parallelize sequential applications to achieve the full potential of these architectures.

Recent years have witnessed a tremendous growth in parallel programming tools, languages, and approaches. In contrast to mainstream general-purpose software development, these approaches have limitations. An important problem in designing parallel applications is to restructure code and/or data to make best use of the available hardware.

Various programming languages, libraries, models and tools have been created to overcome the barriers. However, development of a parallel application or parallelizing an existing program is not an easy task, but a tedious process requiring lots of programmer effort. The difficulty of parallel programming begins from the very first step of programming, clarification of the problem, because not all the problems are easily parallelizable by nature. Programmers need to consider parallel processing in details such as how current workload is distributed over parallel threads and how communication is handled among parallel threads. Therefore, automatic parallelization of sequential programs have been introduced to provide programmers with the ability to parallelize applications easily.

In this study we propose Autopar, a tool to parallelize recursive programs automatically with little programmer effort. It transforms given sequential program code into a new parallelized program by inserting necessary OpenMP pragmas.

The main contributions of this paper can be summarized as follows:

- We propose an automatic parallelization tool for recursive function calls.
- We identify and analyze recursive function calls and obtain characteristics including the number of recursive calls for each recursive function, the size of recursive function in terms of statements, and the number of statements made before recursive function calls.
- We selectively insert OpenMP pragmas to these recursive calls and convert regions of recursive calls into parallel implementations.
- We give experimental evidence showing the success of the proposed approach.

The parallelization process starts with an analysis of source code and determination of code sections to be parallelized if there is any. Autopar does not require any prior knowledge about parallel programming concepts. However, programmer should supply Autopar with a source code following the rules and restrictions of the system explained Section III.

Rest of the paper is organized as follows: Section II provides an overview of existing work on automatic parallelization. Section III explains restrictions and limitations of the proposed system. Section IV discusses Autopar system in detail and outlines our general approach. Section V presents experiment results on few benchmark recursive programs. Section VI concludes the paper.

1530-2016/14 $31.00 © 2014 IEEE
DOI 10.1109/ICPPW.2014.32

II. Related Work

There have been a number of work done in automatic parallelization, and various tools, models have been proposed.

CommSet [2] is not an automatic parallelization tool, but simplifies the job of programmer by asking a specification for commutativity of statements in the program. Programmer assigns statements that can be executed in an arbitrary order to the same set, such that CommSet can determine a parallel execution scheme. The advantage of CommSet over OpenMP is that it does not require programmer to specify a parallelization strategy, rather programmer annotates code blocks that are commutative as defined by application logic and the system handles rest of the work.

A tool, similar to CommSet, for automatic parallelization on MPSoC platforms is offered in [3]. MPA offers optimizations for parallelization and memory management of MPSoC programs by conducting an analysis over programmer written program specification and platform specification.

In [4], authors generalize parallelization concepts that are commonly offered for low-level programming languages to high-level languages that offer a broader range of data abstraction. The proposed system, ROSE, is a source-to-source transformation tool that exploits parallelization of abstract data types in C++.

Data dependence is very important for any kind of parallelization. GCD test [5] is a data dependence test for loops iterating on arrays. It states that array references in the form of $X[a\times i+b]$ and $X[c\times i+d]$ are dependent if $GCD(a,c)$ divides $(d-b)$. Omega [6] test is a more comprehensive dependence test. By formulating loops as integer linear programming it determines under which conditions two references refer to same array element.

The DOALL [7] parallelization can be applied to loops if there is no loop-carried dependency, which is the dependency between loop iterations. In the presence of loop-carried dependencies, Decoupled Software Pipelining (DSWP) [8] offers an alternative. It partitions a loop into several loops with dependencies. By considering these dependencies, it builds a pipeline of threads each corresponding to one of new loops. Parallel-Stage DSWP (PS-DSWP) combines best of both approaches by performing DOALL parallelization within loop partitions and carrying dependences between them using DSWP.

Many loop parallelization strategies assume a static environment in which loop iterates. [9] describes a method to parallelize loops that has dynamic behavior by performing a sensitivity analysis on loop parameters. Dynamic data dependences are expressed in the form of a predicate set, so that static loop parallelization can be performed on dynamic loops.

[10] introduces parallelization of call-by-value recursive functions on general recursive data structures. Parallel implementation of recursive functions is divided into two tasks.

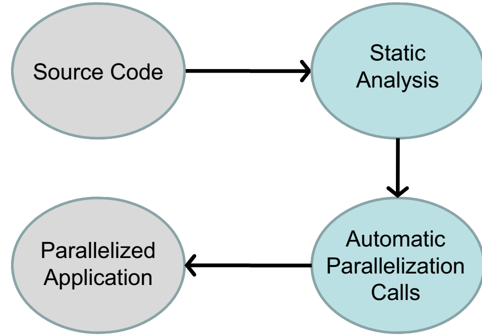

Figure 1: General overview of Autopar workflow.

Sequential functions are transformed into predefined parallel skeletons and these skeletons are implemented in parallel programs. On the other hand, [11] proposes an integrated approach to generate parallel loops. The approach is a two-staged parallelization combining profiling and mapping based on a machine-learning prediction mechanism. A sequential C program is initially extended with plain OpenMP annotations for parallel loops and reductions, and then they apply machine-learning based mapping to generate OpenMP annotated parallel programs.

While aforementioned techniques propose automatic parallelization at different levels, our approach is different in targeting recursive applications. More specifically, we insert OpenMP pragmas in recursive applications.

Automatic parallelization studies generally attack parallelization of loops. Parallelizing recursive functions can be very useful for divide and conquer algorithms. REAPAR [12] is an early work that automatically parallelizes recursive functions by creating a thread at each recursive call with pthreads library. It restricts recursive functions to be void and not to accept pointers to avoid aliasing problems. A more general approach with less restrictions on function definitions is offered in [13]. Rugina and Rinard [14] present a special compiler designed to parallelize divide and conquer algorithms whose subproblems access disjoint regions of dynamically allocated arrays. Although not fully supported, they have some recursion support and dynamic pointer optimizations. In [15], authors discuss parallelizing recursive functions automatically. They apply a quantifier-elimination-based derivation of operators to shrink function closures. Using such an operator, they split the input structure and perform computation parallelly. Our approach is different from these studies since we target OpenMP platforms and insert OpenMP pragmas automatically. In addition, our approach uses novel heuristics to identify recursive regions and selectively insert OpenMP pragmas.

III. Restrictions and Limitations

In order to avoid potential problems Autopar requires input programs to obey following properties:

Figure 2:

Algorithm 1: Automatic parallelization of recursive calls.

```
AUTOPAR(Source)
1: defs ← IDENTIFY-FUNCTION-DEFINITIONS(Source)
2: calls ← IDENTIFY-FUNCTION-CALLS(Source, defs)
3: recs ← IDENTIFY-RECURSIVE-CALLS(defs, calls)
4: anls ← ANALYZE-RECURSIVE-CALLS(recs, calls)
5: Source ← INTRODUCE-OPENMP(Source, recs, anls)
6: return Source
end
```

- Program has to be written in ANSI C containing all recursive procedures and any code calling them in a monolithic structure.
- Recursive procedures having data dependencies among recursive calls are not parallelized. Dependencies inherently block parallelism if latter recursive calls need to wait for former calls to finish.
- Recursive procedures that are going to be parallelized should have a void return type. Similar to previous point, when a dependence on return type exists subsequent recursive calls cannot be executed in parallel.

In addition to programming requirements, there are arbitrary requirements on the format of the given source code. These requirements exist to simplify the implementation of Autopar. However, they do not impose any restrictions on the expressiveness of the supplied program.

IV. Proposed Method

Autopar transforms ANSI C programs to ANSI C programs containing OpenMP pragmas through GCC compiler framework and POSIX regular expressions along with some heuristics to identify function definitions, function calls and recursive calls. Figure 1 shows the general overview of Autopar workflow.

Algorithm 2 defines the high-level tasks Autopar accomplishes. Autopar, first identifies function definitions and function calls. Using these information, it determines which of the identified calls are recursive. After recursive calls are recognized, analysis takes place and gathers information about recursive calls. Finally, OpenMP pragmas are inserted to source code to parallelize recursive function calls.

Algorithm 3 outlines the function definition identification procedure. For each line in source code, comment lines, pragmas and directives are skipped. When the regular expression matches a function definition, it checks the value of the block level. Encountered line is a function definition if current block level is 0. Otherwise, regular expression might have matched for another piece of code that looks like a definition. defs is an array of function definitions. Each function definition is expressed by its type, name, parameters, first and last line of the definition. SAVE() function saves the given information to its first argument, which is a function definition in defs array for this case.

Figure 3:

Algorithm 2: Identification of function definitions.

```
IDENTIFY-FUNCTION-DEFINITIONS(Source)
1: defs ← ∅
2: n ← blockLevel ← 0
3: lineNo ← 1
4: for all line ∈ Source do
5:     if !(line ∼ (comment or pragma or directive)) then
6:         if line ∼ definition then
7:             if blocklevel = 0 then
8:                 firstLine ← lineNo
9:                 SAVE(defs[n], type, name, parameters,
   firstLine)
10:            end if
11:            blocklevel ← blocklevel + 1
12:        else if line ∼ '}' then
13:            blocklevel ← blocklevel - 1
14:            if blocklevel= 0 then
15:                lastLine ← lineNo
16:                SAVE(defs[n++], lastLine)
17:            end if
18:        end if
19:    end if
20:    lineNo ← lineNo + 1
21: end for
22: return defs
end
```

Algorithm 4 describes identification of function calls within the source code. If the line read matches the regular expression of function calls and it is a call to a function we have previously identified, the function call is saved into $calls$. The function calls that are checked whether they are defined in the source code or not, because otherwise there is no way to distinguish between external or library function calls, and calls to the defined functions in the source code.

Algorithm 5 demonstrates the steps to find recursive calls. It takes the function definition and function call arrays as arguments. For each function call, first it is made sure that function call's definition is in the definitions array.

Algorithm 6 analyzes the recursive calls identified in previous steps. It accepts recursive function calls and function definitions, and gathers information about recursive functions to be used during parallelization. At the end of this step, the number of recursive calls for each recursive function, the size of recursive function in terms of statements, the number of statements made before recursive function calls, the number of read and write for the arguments of recursive calls within its definition, and the condition indicating whether given recursive function can be parallelizable are

Figure 4:

Algorithm 3: Identification of function calls.

```
IDENTIFY-FUNCTION-CALLS(Source, defs)
 1: calls ← ∅
 2: n ← 0
 3: lineNo ← 1
 4: for all line ∈ Source do
 5:     if line ∼ call then
 6:         if call ∈ defs then
 7:             procedureID ← GETID(name)
 8:             SAVE(calls[n++], procedureID, lineNo)
 9:         end if
10:     end if
11:     lineNo ← lineNo + 1
12: end for
13: return calls
end
```

Figure 5:

Algorithm 4: Identification of recursive calls.

```
IDENTIFY-RECURSIVE-CALLS(defs, calls)
 1: rCalls ← ∅
 2: n ← 0
 3: for all call ∈ calls do
 4:     funcDef ← SEARCH(defs, call)
 5:     if funcDef ≠ nil then
 6:         if funcDef.start < call.line < funcDef.end then
 7:             procedureID ← funcDef.id
 8:             callID ← call.id
 9:             SAVE(rCalls[n++], procedureID, callID)
10:         end if
11:     end if
12: end for
13: return rCalls
end
```

Figure 6:

Algorithm 5: Analysis of recursive calls.

```
ANALYZE-RECURSIVE-CALLS(recs, calls)
 1: anls ← ∅
 2: nfuns ← ncalls ← 0
 3: INITIALIZE-ANALYSIS-LIST(recs, anls)
 4: for all rec ∈ recs do
 5:     ANALYZE-CODE-SIZE(rec, anls)
 6:     ANALYZE-PARAMETERS(rec, anls)
 7:     ANALYZE-PARALLEL-CONDITION(rec, anls)
 8: end for
 9: return anls
end
```

Figure 7:

Algorithm 6: Introducing OpenMP pragmas.

```
INTRODUCE-OPENMP(Source, recs, anls)
 1: open ← FALSE
 2: recno ← 0
 3: PUT("#include <omp.h>")
 4: for all line ∈ Source do
 5:     if line ∼ recs.calls[recno] then
 6:         analysisid ← recs[recno].analysisid
 7:         parallel ← anls[analysisid].parallel
 8:         if !open then
 9:             open ← TRUE
10:             PUT("#pragma omp parallel sections {")
11:         end if
12:         if parallel then
13:             PUT("#pragma omp section")
14:         end if
15:         PUT(line)
16:         if recs.calls[recno+1] = nil and parallel then
17:             PUT("}")
18:             open ← FALSE
19:         end if
20:         recno ← recno + 1
21:     else
22:         PUT(line)
23:     end if
24: end for
25: return Source
end
```

saved into anls.

The parallelization of recursive calls using OpenMP pragmas is done via Algorithm 7. Set of recursive calls are encapsulated within an omp sections pragma and each recursive call is defined as an omp section that can be executed in parallel with other sections.

V. Experimental Analysis

A. Setup

In this section, we examine results for four different benchmarks. These benchmarks are bitonic, fractal, heat and knapsack. All these benchmarks are recursive implementations of scientific functions. Bitonic benchmark is an implementation of bitonic sort, whereas fractal computes different kinds of fractals. On the other hand, heat benchmark simulates heat diffusion according to thermodynamical equations. Finally, knapsack is a recursive implementation of 0-1 knapsack problem. All the benchmarks are implemented in ANSI C. The restrictions and limitations mentioned in Section III are applied to the benchmarks before experiments. Table I lists the properties of benchmarks collected by

Table I: Benchmark Properties

benchmark	recursive functions	recursive calls	statements before recursive calls	recursive function size	parameters read	parameters written
bitonic	2	4	9/37	15/42	4	1
fractal	1	4	19	23	4	0
heat	1	2	1	17	7	1
knapsack	1	2	9	15	4	1

our analysis module. After realization of recursive functions, Autopar analyzes each function and gathers different kinds of information. These results are used in making decisions in heuristics by applying cost/benefit analysis.

Experiments are carried out on a 12 core Intel Xeon server, where we parallelized the benchmarks using Autopar. Results shown are averaged over ten runs using two threads in the baseline implementation. Our initial results indicate that we can automatically generate parallel implementations of recursive applications with reasonably well performances.

B. *Results*

Figure 8 shows normalized execution times of parallel implementations over the sequential recursive implementation for various input sizes. For each benchmark, performance of the parallel implementation is depicted according to the normalized data size with respect to the base data size.

Based on these results, one can observe that some of the benchmarks are more suitable for parallelizing, while some others are not. As can be seen from Figure 8, parallel implementation of heat using Autopar is slower than its sequential version. While the sequential implementation scales up nicely with the increasing input size, the execution of parallel implementation takes longer. The analysis of heat benchmark shows that it has only one recursive function with two calls. These calls are the very first statements in the application, where seven arguments are read and one of them is written. Read and write access to the same variables is the main reason for such degradations due to synchronization. In fact, parallel implementation did not gain much from few number of sections executed in parallel as well.

Similar to heat, knapsack benchmark also performs poor. As can be seen in Figure 8, similar to heat benchmark, increasing the input size does not result in any performance gain compared to sequential implementation. Rather, parallel implementation exhibits a performance loss due to the overheads introduced by OpenMP. While this benchmark has only two recursive calls with four parameters, they cause accesses to shared variables which limit the parallelization.

On the other hand, considering the other two benchmarks, bitonic and fractal perform much better. Fractal benchmark's parallel implementation performs close to the sequential one when scaling is considered. For most of the input sizes, parallel implementation takes less time compared to the sequential baseline. Compared to both heat and kanpsack, the main difference in fractal is the number of recursive calls. That is, fractal has far more number of recursive calls compared to the other two. In addition, there is no parameter written within the recursive function calls which eliminates the potential shared variable conflicts.

Similarly, bitonic shows performance improvements since the parallel implementation performs well for different input sizes. When this benchmark is considered, there are two different recursive functions with few number of parameters to be read and written, thereby eliminating the synchronization requirements and improving the performance.

Overall, the number of recursive calls affects the performance significantly. For example, when bitonic and knapsack benchmarks are compared, their behavior is much different due to the fact that the number of recursive function calls they include widely vary. Therefore, based on our preliminary results, we conclude that this parallelization scheme may improve the performance of recursive applications with higher number of recursive calls.

In the next set of experiments, we measure the sensitivity to different number of threads. As mentioned before, we use two threads in our baseline implementation. However, the system can potentially have higher number of threads with the emerging manycore architectures. Figure 9 shows the normalized execution times with respect to the baseline implementations of the same application with two threads. As can be seen from this figure, although not optimal, bitonic and fractal scale well with higher number of threads. On the other hand, knapsack does not scale well, whereas heat is even worse. These results are expected due to the aforementioned limitations of these benchmarks.

C. *Discussion*

While our approach uses OpenMP sections for nested parallelism, we are planning to extend this framework to use more flexible OpenMP features. Specifically, we aim to implement our approach using OpenMP 3.0 *tasks* as well. This is especially important since nested parallel regions are well known not to be easy to use.

VI. CONCLUSION

In this paper, we propose an automatic parallelization technique for recursive function calls. We first analyze a given source code, extract function definitions, function calls, and identify recursive calls. We then, parallelize recursive calls by introducing OpenMP pragmas. Consecutive recursive calls are enclosed inside an OpenMP sections

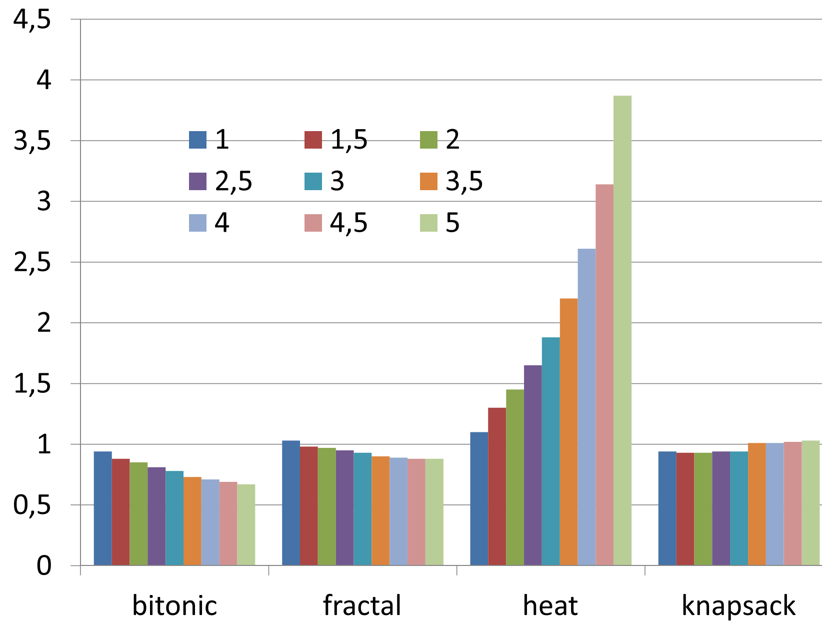

Figure 8: Execution times of parallel implementations over the sequential recursive implementation for various input sizes.

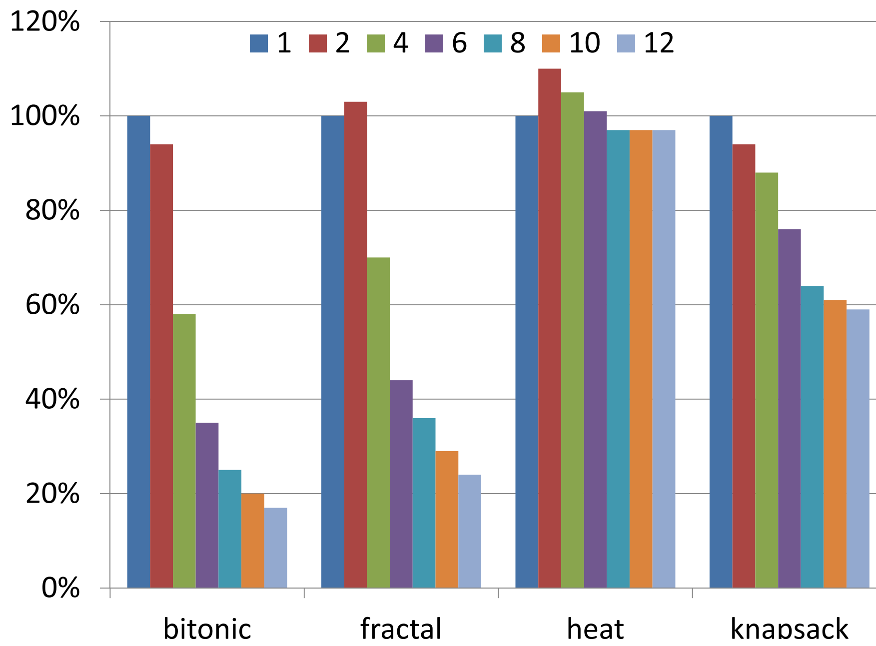

Figure 9: Sensitivity to number of threads. Results are normalized with respect to the baseline implementations of the same application with two threads.

pragma, while individual calls are annotated with parallel section pragma. Our initial experimental results show that our approach can automatically generate parallel code for recursive functions. However, parallel performance mostly depends on the nature of the recursion. Specifically, performance is dependent on the number of recursive functions parallelized, the number of recursive calls, and memory accesses of these recursive functions. While OpenMP introduces overheads due to initialization, data copies, and synchronization, these can be offset by parallel execution.

REFERENCES

[1] ITRS, "International technology roadmap for semiconductors."

[2] P. Prabhu, S. Ghosh, Y. Zhang, N. P. Johnson, and D. I. August, "Commutative set: a language extension for implicit parallel programming," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 1–11. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993500

[3] Y. Iosifidis, A. Mallik, S. Mamagkakis, E. De Greef, A. Bartzas, D. Soudris, and F. Catthoor, "A framework for automatic parallelization, static and dynamic memory optimization in mpsoc platforms," in *Proceedings of the 47th Design Automation Conference*, ser. DAC '10. New York, NY, USA: ACM, 2010, pp. 549–554. [Online]. Available: http://doi.acm.org/10.1145/1837274.1837410

[4] C. Liao, D. J. Quinlan, J. J. Willcock, and T. Panas, "Semantic-aware automatic parallelization of modern applications using high-level abstractions," *International Journal of Parallel Programming*, vol. 38, no. 5-6, pp. 361–378, 2010. [Online]. Available: http://www.springerlink.com/index/10.1007/s10766-010-0139-0

[5] M. J. Wolfe, *High Performance Compilers for Parallel Computing*, C. Shanklin and L. Ortega, Eds. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[6] W. Pugh, "The omega test: a fast and practical integer programming algorithm for dependence analysis," in *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, ser. Supercomputing '91. New York, NY, USA: ACM, 1991, pp. 4–13. [Online]. Available: http://doi.acm.org/10.1145/125826.125848

[7] K. Kennedy and J. R. Allen, *Optimizing compilers for modern architectures: a dependence-based approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

[8] G. Ottoni, R. Rangan, A. Stoler, and D. August, "Automatic thread extraction with decoupled software pipelining," in *Microarchitecture, 2005. MICRO-38. Proceedings. 38th Annual IEEE/ACM International Symposium on*, nov. 2005, p. 12 pp.

[9] S. Rus, M. Pennings, and L. Rauchwerger, "Sensitivity analysis for automatic parallelization on multi-cores," in *Proceedings of the 21st annual international conference on Supercomputing*, ser. ICS '07. New York, NY, USA: ACM, 2007, pp. 263–273. [Online]. Available: http://doi.acm.org/10.1145/1274971.1275008

[10] J. Ahn, T. Han, and C. C. Lengauer, "An analytical method for parallelization of recursive functions," 2001.

[11] G. Tournavitis, Z. Wang, B. Franke, and M. F. O'Boyle, "Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping," in *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '09, 2009, pp. 177–187.

[12] S. U. Haenssgen, "Reapar user manual and reference: Automatic parallelization of irregular recursive programs," 1998.

[13] M. Gupta, S. Mukhopadhyay, and N. Sinha, "Automatic parallelization of recursive procedures," *Int. J. Parallel Program.*, vol. 28, pp. 537–562, December 2000. [Online]. Available: http://dl.acm.org/citation.cfm?id=608713.608746

[14] R. Rugina and M. Rinard, "Automatic parallelization of divide and conquer algorithms," *SIGPLAN Not.*, vol. 34, no. 8, pp. 72–83, May 1999.

[15] A. Morihata and K. Matsuzaki, *Automatic Parallelization of Recursive Functions Using Quantifier Elimination*, M. Blume, N. Kobayashi, and G. Vidal, Eds. Springer Berlin Heidelberg, 2010.