Automatic Inter-procedural Parallelization

Ravichandhran K.M, Bhaskar. P, Annamalai.S.P and Dr.A.P.Shanthi
Department of Computer Science, College of Engineering, Guindy, Anna University

Abstract

In this paper, we introduce a novel technique for automatic parallelization of independent procedures. Our proposed technique identifies the potentially independent procedure calls and generates code for executing them in parallel without the intervention of the programmer. Our major concern is to parallelize those function calls that consume a significant portion of the execution time like recursive function calls and functions calls inside a loop. The proposed technique uses a hybrid approach, which involves both static and dynamic (or runtime) analysis. To reduce the overhead of speculative runtime analysis we extend a technique called Sensitivity Analysis (originally developed for loops) to recursive procedures. If a procedure call is identified as a candidate for parallel execution then OpenMP constructs for executing the function call in parallel are generated. Generating OpenMP constructs has two major advantages, firstly it makes the program easy to understand and debug. Secondly it increases the portability of the program.

I. Introduction

The increasing popularity of multi-core architectures have greatly increased the availability of parallel hardware resource but at the same time has increased the need for developing parallel software that can exploit the available parallel hardware. As Re-engineering the existing program to make use of the parallel hardware is both time-consuming and expensive, software that could effectively parallelize the sequential programs is the need of the hour. Moreover, such an auto-parallelizer will free a programmer from concentrating on writing parallel programs and allow him to think in his own natural way. Unfortunately, the existing automatic parallelization techniques are not effective enough to replace manual parallelization. One reason for the ineffectiveness is that most of the existing techniques concentrate on parallelizing the instructions within a single procedure and do not try to exploit the parallelism existing between the procedures. Though a few techniques do so, they fail to perform in depth analysis and also suffer from serious drawbacks such as increased analysis overhead, increased execution time in case of sequential execution and so on. To counter the above problems we propose a technique for performing advanced inter-procedural analysis with minimum overhead.

II. Related Work

A great deal of previous research in automatic parallelization has been directed towards parallelizing loops and recursive function calls in divide-and-conquer programs [1]. Manish Gupta, Sayak Mukhopadhyay and Navin Sinha have proposed a technique that uses compile-time analysis to detect the independence of multiple recursive calls in a procedure. They have also proposed speculative run-time techniques to allow parallelization of programs for which

compile-time analysis alone is not sufficient. Rugina and Rinard [2] have independently developed similar techniques to automatically parallelize divide-and-conquer applications. They achieve roughly similar results but did not perform any speculative analysis. Both these techniques are effective only for divide and conquer programs. The technique that we propose performs advanced program analysis to identify and parallelize regions in the program which consumes significant amount of execution time but at the same time reduces the runtime overhead using Sensitivity Analysis (SA). SA was originally developed for loops by Silvius Rus et all. [3]. Sensitivity Analysis (SA) is a technique that complements, and integrates with, static automatic parallelization analysis for the cases when relevant program behavior is input sensitive. When SA is extended for recursive functions it can greatly reduce the conditions that need to be evaluated at runtime.

III. Proposed Work

One of the major hurdles in inter-procedural parallelization is the complex interleaving of the function calls. In a flexible and user friendly programming language like C, there is no limit to the amount of interleaving between the function calls, for example any number of functions can be called inside a loop and each of these functions can in turn contain calls to same (recursive) or other functions and so on. Moreover, in some cases the recursion is not immediately obvious, for example a function X can call a function Y which once again calls the function X. For the above mentioned reasons our technique performs a detailed analysis of the program flow to find the potential regions in the program that consumes significant amount of execution time. Once the potential regions are identified they are analyzed for parallelism using Symbolic Array Section analysis [1] and Static analysis techniques. In case the input defies compile-time analysis, speculative run-time approaches coupled with sensitivity analysis is used. Thus, the proposed technique consist of two phases,

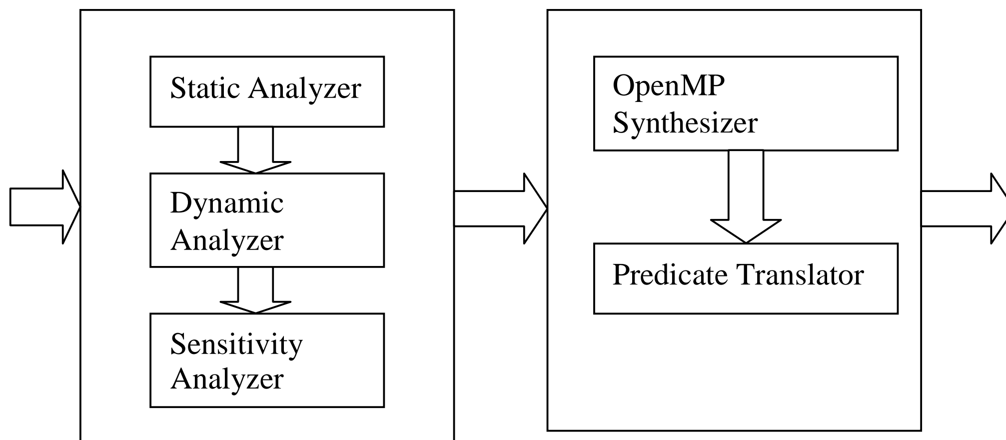
Analysis Phase

In this phase a *function dependency graph* is constructed and the potential candidates for parallelization are determined. Direct and Indirect Recursive functions, function calls within a loop are examples of potential candidates for parallelization. Once the candidates are identified, static compile-time analysis techniques like Symbolic Array Section analysis are applied. If they are not successful, the predicates (or conditions) that need to be satisfied for the parallelization to succeed are determined. These predicates are then simplified using Sensitivity Analysis. Once the parallel procedures and the minimal predicate set are determined they are stored in a convenient data structure.

Synthesis Phase

In this phase, the parallel procedures found in the analysis phase are outputted along with appropriate OpenMP constructs. The predicates, if any, are added to the OpenMP construct as conditions. Those procedures that are not parallelizable are added to the output program as it is without any modifications.

Figure 1



IV. Illustrative Example

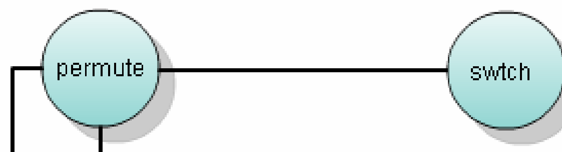
Consider the following C++ program shown in Figure 2 that prints all the permutation of a string.

Figure 2

```
string swtch(string topermute, int x, int y)
{
	string newstring = topermute;
	newstring[x] = newstring[y];
	newstring[y] = topermute[x];
	return newstring;
}
void permute(string topermute, int place)
{
	if(place == topermute.length() - 1)
	{
		cout<<topermute<<endl;
	}
	for(int nextchar = place; nextchar < topermute.length(); nextchar++)
	{
		permute(swtch(topermute, place, nextchar), place+1);
	}
}
```

The Function Dependency Graph for the above program is shown in Figure 3.

Figure 3



The self-loop in the node corresponding to the function *permute* indicates that it is a recursive function. Moreover, intra-procedural analysis of the function *permute* indicates that the function

is called inside a loop. The next step is to find if there is any loop carried dependency but this is complicated by the presence of a function call inside the loop. In general, a function is parallelizable if it does not manipulate any shared data structure and does not return any value or the return value should never be used inside the loop. It is sometimes possible to parallelize a function that manipulates a shared array if every call to the function operates on a separate portion of the array. This can be determined using *Symbolic Array Section Analysis [1].* Since the function *permute* satisfies the above constraints, the loop can be safely parallelized. The parallelization is implemented by inserting appropriate OpenMP constructs in the program. In this example, the output will contain *#pragma omp parallel for* construct before the for loop.

This kind of parallelization is termed as horizontal parallelization. In case of recursive functions another kind parallelization called vertical Parallelization can also be performed which is explained below. Assume that the above program has a single recursive call which is not enclosed in a loop as shown in Figure 4.

Figure 4

```
string swtch(string topermute, int x, int y)
{
	string newstring = topermute;
	newstring[x] = newstring[y];
	newstring[y] = topermute[x];
	return newstring;
}
void permute(string topermute, int place)
{
	if(place == topermute.length() - 1)
	{
		cout<<topermute<<endl;
	}
	permute(swtch(topermute, place,place+1), place+1);
}
```

The data dependency tests described before shows that all the recursive function calls can be executed in parallel. When *permute* is called for the first time, as many threads as the number of recursive function calls are created. As long as the terminating condition *(place == topermute.length)* is not met new threads are created and *place* is incremented for each thread. The first argument is also calculated for each thread. For the first thread the first argument is *swtch(…)*, for the second thread it is *swtch(swtch(…))*, for the third it is *swtch(swtch(swtch(…))* and so on. During the synthesis phase the code for performing the above operations are synthesized and inserted into the program. For programs where both horizontal and vertical parallelizations are possible only one of the methods is applied. This avoids too many threads from being created.

When a node in the *function dependency graph* contains more than one self loop then it implies that there exist more than one recursive call inside the function which is often the case in Divide and Conquer algorithms. These cases are dealt in a manner similar to that of function

calls within loops. The recursive function calls are checked for data dependencies and if the function calls are found to be independent they are executed in separate threads.

V. Conclusion

We have presented a sophisticated, yet practical technique for performing advanced inter-procedural parallelization with minimum overhead. This technique exploits the subtle parallelism exiting in the program that is neglected by almost all modern compilers. This technique can be made more effective if programmers adhere to good programming practices such as writing cohesive programs with little coupling between the modules. We are planning to implement our proposed technique and run it on standard benchmarks to demonstrate the improvement in the performance.

VI. References

[1] Manish Gupta, Sayak Mukhopadhyay, Navin Sinha, "*Automatic Parallelization of Recursive Procedures*", Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, Page(s):139 – 148, 1999.

[2] R. Rugina and M. Rinard, "*Automatic parallelization of divide and conquer algorithms*", In Proc. ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming, Atlanta, GA, May 1999.

[3] Silvius Rus, Maikel Pennings, Lawrence Rauchwerger , "*Sensitivity analysis for automatic parallelization on multi-cores*", Proceedings of the 21st Annual International Conference on Supercomputing ICS '07 Publisher: ACM Press.

[4] Hong-Soog Kim, Young-Ha Yoon, Sang-Og Na & Dong-Soo Han, "*ICU-PFC: An Automatic Parallelizing Compiler*", Proceedings of the High Performance Computing in the Asia-Pacific Region, Volume 1, Page(s):243 – 246, May 2000.

[5] Joonseon Ahn and Taisook Han, "*An analytical method for Parallelization of Recursive Functions*", Parallel Processing Letters, 2000.

[6] OpenMP-ARB. Openmp: A proposed standard API for shared memory programming. Technical report, OpenMP, October 1997.

[7] Ashwin Kumar, K., Aasish Kumar Pappu, Sarath Kumar, K., Sudip Sanyal, "*Hybrid Approach for Parallelization of Sequential Code with Function Level and Block Level Parallelization*" Parallel Computing in Electrical Engineering, 2006.

[8] Ambrus, W., "*A framework for automatic parallelization of sequential programs*", Telecommunications, 2003. ConTEL 2003, Proceedings of the 7th International Conference on Volume 2, 11-13 June 2003.

[9] M. Girkar and C. Polychronopoulos, "*Automatic extraction of functional parallelism from ordinary programs*", IEEE Transactions on Parallel and Distributed Systems, 3(2), March 1992.

[10] J. Gu, Z. Li, and G. Lee,"*Symbolic array dataflow analysis for array privatization and program parallelization*", In *Proc. Supercomputing '95*, San Diego, CA, December 1995.