The paper proposes an approach to exploit the parallelism available in divide and conquer algorithms.

Divide and conquer algorithms are very frequent to use recursion combined with excessive use of arrays and pointer arithmetic, which allows the efficient passing of data within a recursive function call. This means that the algorithm needs the ability to detect associations between the pointers as well as their referenced memory. Otherwise the validity of a determined parallelization couldnt be determined, since two concurrent executing procedures may could contain a pointer, which makes use of the same memory location. Therefore the memory, which a procedure uses, is statically determined and recorded as a set of symbolic region expressions. Each region expressions contains a symbolic lower and upper bound. These bounds are expressed by a reference set of variables. The reference set for a procedure contains e.g. the parameters passed to the procedure or used global variables, thus all variables which could possibly affect possibly shared memory. To compute these regions and use finally use them for parallelization, the algorithm conducts an analysis consisting of several steps:

1. Pointer Analysis

   First, a flow-, context-sensitive and interprocedural pointer analysis is performed. This providers the pointer disambiguation information, which is required by further analysis steps.

2. Bound Analysis

   This analysis is responsible for detecting the bounds of a variable. It consists out of three subanalyses:

   (a) Order analysis:

   The order analysis extracts two kinds of information. For integer variables the zero order is recorded, which means the relation of the variables value compared to zero. Additionally the relative order is determined, which means the relation of the variable's relative to other variables for each pair of pointer and integer variables. These relations are recorded at each program point and can be computed iteratively from each program point to the next.

   (b) Initial Value Analysis

   Although the previous analysis produces relations, which can already be used to determine upper and lower bounds for each variable, the region analysis needs the bounds to be expressed relative to variables of the reference set of a procedure. Therefore the initial value analysis propagates the initial values of all concerned variables in the procedure. Afterwards the analysis tries (whenever possible) to describe each variable within an expression with variables from the reference set.

(c) Correlation Analysis

The correlation analysis tries to determine correlation between variables of the form whenever p is incremented, q is also, thus detecting a strong relation between a set of variables. This is useful whenever the previous analyses fails to detect appropriate bounds for a variable.

3. Region analysis

   The region analysis makes use of the previous analyses to finally derive the symbolic region expression for each function. Therefore it uses an interprocedural fixed-point algorithm to extract the regions accessed by the entire computation of a procedure. These region expressions finally characterize the memory access of the procedure.

4. Parallelization

   Finally the extracted region information is used to determine which procedures of the program can be possibly executed in parallel. Therefore the intersection of the corresponding regions is computed, which indicates if there is a overlapped shared memory use. If this intersection is empty, its safe to assume that the procedures can be executed in parallel.

Open Questions:

The parallellization algorithm has to conservatively assume that procedures with overlapping regions cant be executed in parallel. Could these approach combined with a speculation technique and a measurement when it is profitably to take the risk and execute it anyway?

Summary I2

This week our discussion is about a paper dealing with automatic parallelization of divide and conquer algorithms. These are algorithms that divide a problem into several smaller subproblems, solve them and put their solutions together to a solution of the original bigger problem. As such algorithms often work on disjoint memory regions to solve these subproblems they can be parallelized very well. However the problem is that a parallelizing compiler has to prove that parallel tasks do not access the same memory regions which is not so easy because divide and conquer algorithms are implemented recursively in most cases. For that reason the authors have developed an analysis which is able to identify parallelizable parts of programs that access dynamically allocated memory via pointers and pointer arithmetic.
The analysis consists of four parts and starts with the pointer analysis. In this step it extracts information about memory locations that are used by pointer dereferencing statements. After that a bounds analysis is performed that includes an order analysis, an initial value analysis and a correlation analysis and generates symbolic lower and upper bounds for the pointer dereferences which are used by the region analysis in the next step. In this context it becomes not really clear to me how the order analysis works and what it does. The next step finds the regions that are accessed by each procedure and computes the region expressions by an interprocedural fixed point iteration. At the end the last step called "parallelization" identifies independant procedure calls and generates parallelized code.
Testing their approach on a divide and conquer matrix multiplication program and a sorting program the authors reached speed-ups to 7.5 (matrix multiplication) and 4.4 (sorting) and could demonstrate its benefit for parallelizing compilers.

Questions / Opinion:

1. I understood the overall topic of this analysis but I don't get the concrete differences between these steps.

2. In the benchmark section they state that the naive matrix multiplication needs 316 seconds of execution time on a single core machine and the automatically parallelized version only 28.5 seconds. But where comes this difference in execution time when there is no real parallelism because of a single core?

1 Summary

The authors propose a technique to automatically parallelize recursive divide and conquer algorithms, that use dynamic memory allocation and/or pointer arithmetic. At first, a flow-sensitive, context-sensitive, and interprocedural fixed-point pointer analysis is performed. This analysis is able to deduce invariants for pointers that form a set of possible values (e.g. an interval), here called region expressions. This information is used later to derive the memory access behaviour of a function call, regarding read- and write-operations separately. The analysis assumes, that a write to the pointer's target address writes all addresses covered by the pointer's region expression, and a read from a pointer's target address reads all addresses covered by the region expression, respectively. If the analysis does detect no more accessed memory regions, a fixed point has been reached.
In the next step, the algorithm performs a pointer analysis to detect possible memory dependences between two call sites. If no such dependences exist between the call sites, the algorithm uses the extracted information to expose the possible concurrency to a run-time system. Therefore, the parallel section algorithm detects both spawn and sync points. When it comes to code generation, the compiler inserts spawn and sync instructions from the Cilk language.
As the experiments have shown, the presented algorithm is able to automatically detect parallelism in both recursive and non-recursive divide and conquer algorithms. The improvements are up to 7.5x relative speedup on 8 processors. Besides the fact that the algorithm succeeds in introducing parallelism, the program transformation also introduces cache improvements, that led to a relative speedup of 11x for the naive matrix multiplication, running on only 1 processor.

2 Questions

I would like to now how synchronization points can have an impact on caching behaviour.
On page 2 the authors claim "This code will usually run faster than code that uses integer array indices to identify and solve subproblems.". How can this be explained?

Summary: Automatic Parallelization of Divide and Conquer Algorithms

Divide and conquer algorithms provide great scope for parallelization as independent subproblems can be executed in parallel. Even dividing the problem into subproblems and merging the solution of subproblems can be parallelized. Paper discusses how this can be achieved.

Authors describe a sequence of analyses on the divide and conquer algorithms in order to parallelize them. These analyses work as a pipeline, result of one analysis serving as input to the next analysis. Objective behind these analyses is to find the accesses to disjoint regions in memory by subproblems. If two subproblems access disjoint regions in memory, they can be executed in parallel.

Solution first performs the flow-sensitive pointer analysis. This provides pointer disambiguation information required for the subsequent analyses phases. Results of this phase is also used to identify disjoint region expressions.

Bounds analysis is divided into three sub-phases. Order analysis extracts information about values of variables relative to zero as well as relative to each other. Thus lower bound and upper bound values of variables can be determined. Initial value analysis propagates initial values of the variables through the program. Each variable is mapped to an expression containing variables in the reference set at each program point. Correlation analysis detects relationship between variables, and maps each target variable to set of correlated variables. This kind of analysis helps in determining unknown bounds of a target variable by using bounds of correlated variables.

Region analysis is divided into intra-procedural and inter-proceduaral region analyses. Intra-procedural region analysis returns the region expression for each subproblem. A region expression is nothing but the region in memory which is accesses by that subproblem. This is generally expressed in the form of a range having a lower and upper bound. Inter-procedural analysis combines region expressions from different subproblems to derive a global regon set.

Once the the subproblems with disjoint memory access patterns are discovered, spawn primitive can be used to parallize each such subproblems.

Open questions:
1. Details about lattice, transfer functions used in order analysis phase.

Automatic Parallelization of Divide and Conquer Algorithms

April 28, 2014

Summary

This paper discusses a technique to detect function calls in C code that can be executed in parallel. The overall algorithm first runs several static analyses based on which it detects the read and write areas of functions relative to their call context, i.e. the parameters they are called with.

The several steps of analysis are as follows: First, it runs a value analysis to detect the values of each pointer at each program point. The domain they use for that is some kind of heavily restricted polyhedral domain. Based on that relational information, they detect relative intervals for read and write accesses through pointers. All of that is done intraprocedural.

After that, they use this information to derive first read or write intervals for each memory access statement and finally for a function itself. In the last step, an interprocedural analysis is performed that lifts this projects this per-function-information to each of it's calls, thereby investigating how this call may or may not interfere with previous or following function calls and statements.

This infomation is finally used to introduce safe parallelism by the following mechanism: The algorithm tries to create for each function call the largest possible "window" of statements that do not overlap in their write accesses with any read or write of the function or vice versa. It then introduces a *spawn* keyword to mark those function calls that can be executed parallel to their caller context and *sync* statements at those points where the parallel "window" ends.

In the end there are given two examples (one of those with code) on which the given algorithm performs well and some evaluation results which show that fact. And while this paper focuses on the parallelisation of (potentially recurrent) function calls on source code level, there is also mentioned some related work that seems to do some similar stuff on more or less different scenarios (differing in the way of memory referencing or focusing on iterative code instead of recurrent).

Open Questions

- In section 6.3, there is a distinction between recurrent and non-recurrent functions. Is it really necessary to have two cases for this given a fixed-point algorithm that is capable of handling both?
- The effectiveness of the algorithm seems to heavily depend on the precision of value analysis to get tight interval bounds later on. Nevertheless, this part is hardly discussed in the paper. Is the choice of domain for that reasonable / always sufficient even for more complex algorithms than mergesort? Or will the use of more powerfull relational domains such as octagons improve the results for that kind of algorithms?

Summary Automatic Parallelization of Divide and Conquer Algorithms

The paper presents an approach of finding parallelims in divide-and-conquer algorithms, taking recursion and pointer operations into account. The actual parallel execution is deferred to the Cilk environment. It marks sections for parallelization if they don't violate any dependencies (such as write after write). To extract those dependencies, the authors have developed multiple analyses. Initially, they use a pointer analysis which among other things disambiguates pointers, and is used by the following analyses. The next analysis computes information for integer variables' value relative to 0, and about the relation of multiple integer and pointer variables. Building further upon this, the next step is to determine the initial values of variables to translate the bounds of the last analysis to bounds with regards to incoming function parameters and globals. To improve the accurancy of the bounds, a further correlation analysis is used, exploiting information about dependent changes happening to a set of variables. Finally, the collected information about bounds is used for a region analysis. That one yields a set of accessed memory regions, distinguishing between reads and writes. It uses a fixpoint iteration to compute those sets, coalescing the bounds of calees into a calling procedure's bounds. The authors state that for recursive procedures a fixpoint might however never reached; therefore they limit the number of iterations their compiler will do. It is not clear if this could be avoided by a better algorithm, or if the issue is inherent to the analysis. The authors did benchmark their work on two programs, obtaining speedups up to 7.5. However, as their programs are not given, it is not possible to determine if the examples were just cherry-picked to showcase the approach.

Open questions:

- The algorithm was designed with divide-and-conquer algorithms in mind; is it also beneficial for other programs?
- The order analysis does not "perform the full transitive closure" for expression of the form i j + n. Why?
- Could the analysis handle assignments of the form $i = *j$ better (not discarding all information about i), when there are multiple possible values for j, but all of them are equal?
- How well does this approach work once some functions are not available for the analysis (e.g. functions provided by an external library)?
- Why exactly is the matrix example represantive of matrix manipulation programs? What does "represantive" actually mean in this context?

Week 2 Summary

This week's paper discusses an approach to parallelize divide-and-conquer algorithms which often rely on recursion and therefore could not be parallelized by then-available methods. Divide-and-conquer algorithms, such as Mergesort, divide a problem into sub-problems until they are small enough for a simple algorithm to solve the problem. Most of the time, those sub-problems are completely independent from one another and could be easily solved in parallel. But known approaches never dealt with recursion and could not find this parallelism. The presented approach implements a compiler to detect those cases of parallelism. Especially, this compiler needs to prove that those sub-problems do, in fact, not interfere.

For that, a new method had to be developed. In short the algorithm works by calculating memory-regions. If those do not intersect, they can be parallelized (as they won't work on the same range of memory), otherwise not.

In detail the algorithm can be broke down into 4 parts: the pointer analysis, boundary analysis, calculating the memory-regions of each procedure and finally generating the parallel code for independent sections.

The pointer analysis collects data about memory locations used by pointer dereferencing which is then used by the boundary analysis to calculate some form of "lower" and "upper" bound of those memory locations. The boundary analysis combines multiple steps: order analysis, initial value analysis and a correlation analysis.

The boundaries can then be used to calculate the region of memory each procedure accesses. To get those, a fixed-point iteration needs to be performed.

The final part of the algorithm is to simply compare those regions to one another and identify parts of code which can be executed in parallel and, of course, parallelize them.

The authors report significant speedups in divide-and-conquer algorithms such as sorting or matrix multiplication.

Automatic Parallelization of Divide and Conquer Algorithms

This paper works with the technique of Divide and Conquer algorithms which are based on multi-branched recursion. A divide and conquer algorithm works by breaking a problem down recursively into sub-problems of similar type as long as they do not become simple to be directly solved. All the solutions to the sub-problems are then combined and formed into a solution for the original problem. This technique becomes very effective when used with modern parallel machines. The large amount of inherent parallelism in the divide and conquer algorithms make working with caches and deep memory hierarchies. However, parallelizing compilers is a hassle. Usually parallelizing compilers only focus on a form of concurrency involving loop nests. However this paper tries to broaden the area of attention. The parallelizing compiler represented in the paper has a broader focus on recursively generated concurrency. By exploiting pointer analysis algorithms as well as symbolic analysis algorithms recursive concurrency is implemented. Given that the divide and conquer algorithms have inherent parallelism, after dividing the problem into sub-parts, the resultant sub-problems are independent and can be solved in parallel. Divide and conquer algorithms also provide recursively generated concurrency. These algorithms are also good at cache performance. These algorithms run mostly out of the cache, thus work pretty fast. The compiler designed attempts at parallelizing algorithms using pointers so that disjointed regions of dynamically allocated arrays. Interprocedural fixed-point analysis is used. This paper also introduces a basic analysis technology which can optimize traditional parallelizing compiler technology using pointers. There are steps in the analysis, pointer analysis, bounds analysis, region analysis, parallelization. The pointer analysis extracts information used by all succeeding analyses. Then the intraprocedural bounds analysis extracts symbolic upper and lower bound values of pointer variables. Region analysis extracts the set of regions accessed by each procedure. In the parallelization step the concurrency extractor compares region expressions to find independent procedure calls. The question that arises is, can the reasoning by the compiler be done dynamically.

Automatic Parallelization of Divide and Conquer Algorithms

1 Summary

The paper proposes a technique for automatic parallelization tailored to fit the unique properties of divide and conquer algorithms. The authors define the scope of their work very strictly, being algorithms that apply divide-and-conquer and therefore make heavy use of recursion, dynamically allocated memory and pointer arithmetic.
The focus of the paper lies on defining an analysis with multiple stages that tries to judge if two function calls can be executed in parallel. Using this information they transform the source code to allow concurrent execution of the known to be independent function calls. To reduce spawning and syncing overhead they produce code for the Cilk framework which will only spawn as many tasks in parallel as are needed to keep the machine busy.
Their analysis consists of roughly three stages: At first they perform a pointer analysis proposed in another paper by the same authors, this analysis computes the abstract memory locations that pointers point to and enables them to symbolically reason about the pointers in the program in later stages. Following the pointer analysis they perform a bounds analysis which tries to find lower and upper bounds for each pointer dereference on the intraprocedural level. Using these bounds they can perform their region analysis in the third stage. They utilize a fixpoint iteration to compute what abstract memory regions are read and written by each procedure using the bounds information obtained in stage two.
After the third stage of the analysis these region expressions are used to identify groups of function calls that can be executed in parallel. To achieve this they traverse the control flow graph and put adjacent calls whose region expressions are independent into the same parallel section. If the calls are found to be dependent they insert a sync point. All parallel sections that contain at least two calls are then marked as spawn points.
They evaluate their technique by parallelizing two example algorithms and executing them multiple times with a different amount of usable cpu cores. This shows that the speedup significantly grows with the number of cpu cores.

2 Questions

- How do conditionals influence the placement of syncpoints? For example if there is a callsite C_1 followed by a conditional branch between the callsites C_2 and C_3. Further C_1 and C_2 should be independent but not C_1 and C_3. Do they place a syncpoint before the conditional and do C_1 and C_2 still form a parallel section?

Summary:
“Automatic Parallelization of Divide and Conquer algorithms”

Divide and Conquer algorithms pose a problematic challenge for parallelizing compilers. As these algorithms tend to break the problems into small sub-problems and recursively solve them, they seem to have a lot of inherent parallelism. They tend to also have a good cache performance.

However, parallelizing programs that use Divide and Conquer algorithms can be quite difficult as their natural formulation is recursive. In this paper, the researchers tried to design and implement compiler that parallelizes programs that use these algorithms. To achieve this goal they develop a new approach for parallelizing compilers and a new set of sophisticated analyses that realize this approach. A range of symbolic analysis algorithms build on the pointer analysis information to extract symbolic bounds for the memory regions accessed by procedures that use pointers and pointer arithmetic. This information allows the compiler to find procedure calls that can execute in parallel without violating data dependencies.

They conclude by showing some tests and experimental results. They test dependence and implement also the parallelizing compiler, based on the analysis algorithms from this paper, using SUIF compiler infrastructure. The results satisfy the expectations showing good program performance and good speedup.