ABSTRACT

Many computers nowadays contain several processor cores. These cores allow parallel execution to decrease the execution time of programs, but parallelism introduces challenges. How can you determine which parts of a program should be parallelized? This decision can be made explicitly by a programmer, but then business logic will be interleaved with code to parallelize the program. Implicitly running code in parallel is problematic as well because a program can be parallelized too much: For trivial computations, the overhead of running code in parallel is higher than the actual computation. Our research is to determine when a part of a program should be run in parallel. This determination is made by profiling functions and use this information to make a decision at runtime. This paper presents a metric-based approach for implicit parallelism.

Keywords

Complexity, Function Analysis, Implicit Parallelism

1. INTRODUCTION

This research investigates whether function profiling is a viable way to parallelize a program. Function applications are timed while being executed. The resulting execution time is saved in the profile along with properties of the function arguments. In this research these properties are defined as metrics. How these metrics are determined and used is described in Section 6. The resulting profile can be used at runtime to determine if certain functions should be parallelized.

Metrics have a close relation to function complexity. Function complexity is explained in Section 3.3. The execution time of a function is related to the arguments that are given to it. These measurements can be used to calculate an estimated complexity of a function.

The profiler is encapsulated in a small embedded language. This language is a strict functional language (Section 3.2). The language is interpreted and executes programs in parallel automatically. A profile can be used to decide at runtime which parts of the program should be parallelized. The syntax and semantics of the language are explained in Section 4.

Functions can be profiled at runtime (online) or beforehand (offline). For offline profiling, representative input needs to be generated for functions to be able to profile them. How the input is generated is described in Section 8.1. Offline profiling allows functions to be profiled more often and the overhead at runtime is lower, but generated input could differ from actual input during runtime which could give an inaccurate profile. Online profiling does not need to generate input because the input is real data. This gives more accurate profiles, but online profiling needs to execute the functions with real data before it can create a profile.

2. RESEARCH QUESTIONS

With this research we want to see if function profiling with metrics is a viable way to parallelize programs. To accomplish this, a few subproblems need to be solved:

– How do we represent metrics?
– When should a function be profiled?
– How can we use profiling measurements to determine if computations should be run in parallel?
– Is the time overhead that function profiling costs small enough to decrease the execution time of a program?

3. BACKGROUND

3.1 Fork/Join

The Fork/Join framework is used to actually run code in parallel. This framework uses a work-stealing [1] algorithm to schedule tasks. Fork join has two methods: fork and join. A task can spawn new tasks by using fork and can aggregate spawned tasks together using join.

3.2 Strict Functional Language

The embedded language is a strict functional language. Functional languages have their foundation at the lambda calculus [3]. The lambda calculus is a powerful notation that allows anything that is computable to be expressed in it. A lambda expression consists of lambda functions and applications. A lambda function is an anonymous function that has a single input variable. An application binds each occurrence of the input variable of a lambda function to a given expression. For example, consider the following function in pseudocode:

\[ \text{summult}(a, b, c) ::= (a + b) \times c \]

This function can also be expression in lambda calculus with a lambda function for each argument:
A lambda function introduces a lambda function. The right side of the dot is the body of the lambda function. Applying the lambda function with \((a=2, b=3, c=4)\) can be visualized as the following:

\[
\lambda a. \lambda b. \lambda c. (a + b) \times c
\]

Each lambda introduces a new variable and three applications bind the variables \(a, b\) and \(c\) to \(2, 3\) and \(4\) respectively. The result is an application of \((2 + 3) \times 4\) which is equal to 20. An application tree like this can be reduced to an answer. Functional languages are based on reducing these application trees. The language is called a strict functional language when this tree is reduced eagerly: all nodes are completely evaluated bottom up. For some functional languages, such as Haskell, only the nodes that are directly needed are reduced and whenever the same calculation is needed multiple times, it is only evaluated once. These languages are evaluated with lazy evaluation [3].

Laziness has the advantage that only the nodes that are needed are evaluated. Laziness does have problems with parallelism because expressions are only evaluated on demand[2]. Therefore, functional languages usually resort to strict evaluation when parallelism is used.

One of the advantages of functional languages is that they are referentially transparent: computations do not have side effects. Once a variable is bound to an expression, this variable cannot be bound to a different expression. Because of this, the order of the evaluation of children in an application tree is not important. This makes parallelism easier to implement.

The lambda calculus can also express data structures. To create a data structure, a constructor function is needed. This is simply a method that expects the data fields as arguments. A data structure can be viewed as an application of a data constructor function. A linked list can be represented as following:

\[
\text{Cons} ::= \lambda x. \lambda xs. \\
\text{Nil} ::= \lambda.
\]

In this definition, Cons is a node in the list that contains a value ‘x’ and a remainder list ‘xs’. Nil represents the empty list and is always the last element in the list. A list can now be represented as an application of these functions:

\[
\text{alist} ::= (\text{Cons} \ (\text{Cons} \ (\text{Cons} \text{Nil})))
\]

Note that in this example the parentheses denote applications. Writing it down with parentheses improves the readability of a function application. alist is now a linked list with values \([1, 2, 3]\). With this construct, data structures can be stored, but these structures can not be broken apart. Pattern matching is the mechanism that allows access to data constructor fields. A data structure can be matched against the named data constructor with a lambda function to bind the fields of the data structure to variables:

\[
\text{match} \ \text{alist} \ \{ \\
\text{Cons} & \lambda x. \lambda xs. f(x, \ xs) \\
\text{Nil} & \lambda g() \\
\}
\]

Whenever \(\text{alist}\) is a Cons application, \(\lambda x. \lambda xs. f(x, \ xs)\) is matched and evaluated. If \(\text{alist}\) is a Nil application, then \(\lambda g()\) is matched and evaluated. In this pattern match, the functions \(f(x, \ xs)\) and \(g()\) can be any function that operates on the data structure that is given to it. Whenever \(f(x, \ xs)\) calls the same pattern match recursively on the remainder of the list, the whole list is iterated over.

### 3.3 Function Complexity

A function that is executed takes a certain amount of time to complete. The execution time can depend on the arguments of the function. The characteristic of the execution time of a function is modelled in a complexity function. This function defines upper bounds of computation time in terms of a certain property of the data structure it operates on. This function is noted as \(O(n)\), where \(n\) is a polynomial function. As an example, consider a function that counts the amount of elements in a list. To count the amount of elements, all elements should be iterated over, so the complexity is \(O(n)\), where \(n\) is the length of the list. A function that finds an element in a balanced binary tree with \(n\) nodes has complexity \(O(\log(n))\), since it needs at most as many iterations as the depth of the tree to find an element in the tree. The problem with complexity functions is that they use certain information about the structure of the data. For this reason, it is not trivial to create a generic analyzer that defines the complexity of a function that uses data structures. The analyzer needs extra information of the important aspects that characterise a data structure. In this paper, we call these aspects metrics.

### 4. Embedded Language

In order to run a program in parallel, fine grained control is needed at runtime to identify the program structure and to run code in parallel. It would be challenging to incorporate this fine grained control into an existing language like Haskell since it has a lot of different semantic structures. Therefore, we decided to implement a simple embedded language with minimal semantic structures to simplify this control.

The embedded language is implemented in Scala. Scala was chosen because it allows programming in a functional style. It compiles to JVM bytecode and allows Java libraries to be used. This allows us to use the Java Fork/Join framework.

Our language is a strict functional language. Strictness is chosen because parallelism is easier to implement in a strict language than in lazy languages [2]. Our language supports the lambda calculus, pattern matching and data constructors. The language grammar is shown in Figure 1. For example, consider the quicksort algorithm expressed in our language:
analyzing the actual data structure, but that would only cover generic metrics that are the same for all data structures. These metrics could miss important properties of data structures and give inaccurate profiles. For that reason, we define the metrics manually. The metric of a data structure is defined as a function that calculates the metric value for an instance of that data structure. It has the following type signature:

\[
\text{Metric} :: \text{Type} \rightarrow \text{Name} \rightarrow (\text{Constructor} \rightarrow \text{Int})
\]

Recursive data structures can have their metrics defined recursively. The metrics are calculated bottom up. Because of recursion, it is important that these functions have a constant time complexity. Whenever a metric function traverses all children elements, the complexity of calculating metrics for the whole data structure with recursion depth \(n\) will become at least \(O(n!)\). Therefore, a function is defined that can extract metric data from children data structures: \text{metric}. This function has a constant time complexity and therefore a recursive metric can be defined in terms of this metric function. Consider the following metric definition for list length:

\[
\text{metric List length list} = \text{match list}
\]

This code sample creates a function called ‘qsort’. It has a single argument, a list. This list is pattern matched to extract the list type and its data fields. The ‘concat’ function appends two lists together. ‘filter’ is a higher order function. It expects a predicate function as an argument. The function \text{qsort} is defined that can extract metric data from children data structures:

\[
\text{metric}
\]

\[
\text{List length cons xs} = \text{match list}
\]

\[
\text{match list}
\]

\[
\text{Cons x xs} \rightarrow \text{concat}
\]

\[
\text{qsort(filter}(\lambda y \rightarrow \text{lte}(y, x), \text{xs}),
\text{(Cons x}
\text{qsort(filter}(\lambda y \rightarrow \text{gt}(y, x), \text{xs}))
\text{))}
\]

\[
\text{Nil} \rightarrow \text{Nil}
\]

5. PARALLELISM

Calculations have to be split up to be able to parallelize a program. Figure 2 shows a simplified Abstract Syntax Tree for the quicksort algorithm. Each branch is a calculation that can be calculated independently of the other branches at the same level. Because some computations in the tree are trivial, only the branches that need enough computation time for parallelism to be profitable need to be considered for parallelism. The nodes annotated with an ‘@’ show possible nodes that could be parallelized. These nodes are applications of named functions. Only these computations are considered to be parallelized, because these functions can be identified by name and can be reused after profiling.

6. METRICS

For function profiling, the language needs to know what properties function arguments have. Metrics need to be defined to be able to measure how complex an argument is. These metrics could be determined automatically by...
at all and instead returns the data structure as it is. When this flag is not set, the reducer reduces the constructor fields first and then calculates all metric definitions that are given in this context and appends them to the header. When all metrics are evaluated, the evaluated flag is set to true and the evaluated data structure is returned. This representation is shown in Figure 3. In this example, three metrics are defined for list: the length of the list, the maximum element in the list and the average element in the list.

7. ONLINE PROFILING

Online profiling is done at runtime. The profiling is performed on actual data. The computation is slowed down due to the overhead of profiling. Because of this, only 1 profile is allowed to run at a given time to prevent profiles from interfering with each other. The profiler needs to decide when a certain function should be profiled. In our implementation a function application has a random chance to be profiled. A random function is used because it is a quick calculation and gives a good average distribution of functions that are profiled. Once a profiled function finishes its calculation, the execution time is saved along with the name of the function and the metrics of each argument.

8. OFFLINE PROFILING

A function profile is created before execution to determine when a function could be expensive enough to be run parallel. This profiling method executes functions with different parameters to weigh the increase in parameter values against execution time. To keep this profiling in an acceptable time, there should be a threshold in execution time that a function is allowed to execute. Figure 4 shows the algorithm that the offline profiler has. Each function gets profiled for each argument it has. These timings are turned into a polynomial function by interpolating the collected data points. The polynomials for the algorithms are then added together to get a single polynomial for the whole function.

8.1 Generators

For offline profiling, the functions need generated arguments. A data structure needs to provide generators for each of the metrics that are defined. This way, the profiler can use these generators to measure the execution time using a single metric at a time. A generator is, like a metric, defined as a function. The signature of a generator function is as follows:

\[
\text{Generator} :: \text{Constructor} \rightarrow \text{Metric} \rightarrow \\
\text{Metric} \rightarrow \text{Level} \rightarrow \text{Constructor}
\]

Generators need a random function to create random inputs. This function breaks referential transparency. To keep the language referentially transparent, this random function is only allowed in the generation of random inputs during offline profiling. Since this randomness will only affect the measurements and not the outputs of the actual functions, the functions in the language remain referentially transparent.

9. FUNCTION PROFILES

Both online and offline profiling create function profiles. These profiles have to be interpreted at runtime to make decisions which computations should be parallelized. A function profile has the following format:

\[
\begin{align*}
\text{fname} & : \text{time: . . . ns} \\
\text{arguments: } & [ \\
\text{arg1} & : \text{metric1: 10, metric2: 35} \\
\text{arg2} & : \text{metric1: 34, metric2: 68} \\
\}
\end{align*}
\]

\text{fname} is the name of the profiled function. Each measurement saves the time duration in nanoseconds. Each function argument has a list of all metrics that were calculated at runtime.

We have not yet implemented a system that actually uses these function profiles for parallelism. Further research would be needed to develop an algorithm that interprets a profile and then decides which computations should be parallelized and which should be directly computed.

10. RELATED WORK

There is a lot of research in the field of parallelism. For example, EasyFJP[4]. This approach uses policies to de-
termine when a task should be split up. A function that could be parallelized needs to implement a predicate function that decides under what conditions a function call is parallelized. In contrast to our research, each profile is specific for a single function. Our metric based approach focuses on function arguments instead.

Rul et al. attempt to help the programmer implement parallelism [5]. This research analyses the code to find data dependencies and tries to find potential candidates to be executed in parallel. The programmer hand picks the parts to be executed in parallel.

11. CONCLUSION

Metrics allow functions to be profiled in terms of the structure of the data. Whether these measurements can be used to decrease computation time can not be concluded yet.

Metrics are represented by functions that map a data structure to an integer value. These metrics functions are used to calculate metrics of concrete data structures. The results of these functions are saved within a header of the data constructor as shown in Figure 3.

These metrics are then used to to profile a function. Whenever a computation is being reduced to an answer, a function profile can be started. The implementation uses a random chance function to determine if a function should be profiled, but a more sophisticated system can be used in the future to determine when a function should be profiled. Only one function is profiled at time to prevent measurements from interfering with eachother.

The resulting profile gives a mapping from functions to execution times in relation to the metrics of their arguments. These measurements are aggregated together to form an average execution time per unit of metric. These averages are then be used at runtime to make a decision whether or not a computation should be parallelized.

The metrics have a rather large overhead in regards of time and storage. When you consider the List data structure, each node in the list holds a single value. When these values are integers, they only take up a single word of memory apart from pointers. When a few metrics are defined, each metric appends another word to every node. That means that each node could take up to \( n \) times more memory, where \( n \) is the amount of metrics defined for that data structure.

The computation time of the metrics is constant. Unfortunately, this constant is rather large. When benchmarking some functions, the added time overhead of adding the metrics to the datastructures alone increased the total computation time to at least two times the normal execution time. After optimization this constant could be reduced, but the overhead is still troublesome.

The implementation is available on GitHub at the following url:

http://github.com/besuikerd/parallanguage

The current implementation gives insight in the difficulties that are faced when implementing function profiling to parallelize programs. If the runtime calculations of metrics could be optimized and a clever algorithm is developed to interpret profiles, we expect that a metric based approach for implicit parallelism is a viable method to parallelize programs.

12. FURTHER WORK

The current implementation does not yet use the profiles to parallelize computations. Currently it is native and parallelizes every function application. In future implementations this profiling should be incorporated in the code that handles the parallel execution of computations.

The overhead that metrics introduce needs to be reduced in both memory usage and computation time. A different approach is needed to compute the metrics needed. A possible alternative is to calculate the metrics on demand only when a calculation is needed and memoize the result of the current data structure so that this calculation is only computed once. The advantage is that not every node in the data structure contains metric data. The metrics do not have to be constant time functions anymore. A disadvantage is that the metric calculations could take longer than the time needed to actually calculate the function.

13. REFERENCES


