CONCURRENCY ANALYSIS IN DYNAMIC DATAFLOW GRAPHS

Seminar Report

Submitted in partial fulfillment of the requirements for the award of degree of

BACHELOR OF TECHNOLOGY

In

COMPUTER SCIENCE AND ENGINEERING

of

APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

Submitted By

NITHA FATHIMA

Department of Computer Science & Engineering
Mar Athanasius College Of Engineering Kothamangalam

CONCURRENCY ANALYSIS IN DYNAMIC DATAFLOW GRAPHS

Seminar Report

Submitted in partial fulfillment of the requirements for the award of degree of

BACHELOR OF TECHNOLOGY

In

COMPUTER SCIENCE AND ENGINEERING

of

APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

Submitted By

NITHA FATHIMA

Department of Computer Science & Engineering
Mar Athanasius College Of Engineering Kothamangalam

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
MAR ATHANASIUS COLLEGE OF ENGINEERING
KOTHAMANGALAM

CERTIFICATE

This is to certify that the report entitled **Concurrency Analysis in Dynamic Dataflow Graphs** *submitted by* **Ms.NITHA FATHIMA**, *Reg. No.* **MAC15CS043** *towards partial fulfillment of the requirement for the award of Degree of Bachelor of Technology in Computer science and Engineering from APJ Abdul Kalam Technological University for December 2018 is a bonafide record of the seminar carried out by her under our supervision and guidance.*

....	....	....
Prof. Joby George	**Prof. Neethu Subash**	**Dr. Surekha Mariam Varghese**
Faculty Guide	***Faculty Guide***	***Head of the Department***

Date: Dept. Seal

ACKNOWLEDGEMENT

First and foremost, I sincerely thank the God Almighty for his grace for the successful and timely completion of the seminar. I express my sincere gratitude and thanks to Dr. Solly George, Principal and Dr. Surekha Mariam Varghese, Head Of the Department for providing the necessary facilities and their encouragement and support.

I owe special thanks to the staff-in-charge Prof. Joby George , Prof. Neethu Subash and Prof. Joby Anu Mathew for their corrections, suggestions and sincere efforts to co-ordinate the seminar under a tight schedule. I express my sincere thanks to staff members in the Department of Computer Science and Engineering who have taken sincere efforts in helping me to conduct this seminar. Finally, I would like to acknowledge the heartfelt efforts, comments, criticisms, co-operation and tremendous support given to me by my dear friends during the preparation of the seminar and also during the presentation without whose support this work would have been all the more difficult to accomplish.

ABSTRACT

Dynamic dataflow scheduling enables effective analysis of concurrency while making parallel programming easier. Analyzing the degree of concurrency available in dataflow graphs is an important task, since it helps programmers to assess the potential performance a program can achieve via parallel execution. However, traditional concurrency analysis techniques only work for DAGs (directed acyclic graphs), hence the need for new techniques that contemplate graphs with cycles. Here we present techniques to perform concurrency analysis on generic dynamic dataflow graphs, even in the presence of cycles. In a dataflow graph, nodes represent instructions and edges describe dependencies. A set of theoretical tools for obtaining bounds and illustrate implementation of parallel dataflow runtime on a set of representative graphs for important classes of benchmarks to compare measured performance against derived bounds is provided.

Contents

List of Figures

LIST OF ABBREVIATION

DDG	Dynamic Dataflow Graphs
DAG	Directed Acyclic Graphs
PE	Processing Elements
CDDG	Conditional Dataflow Graphs
TALM	TALM is an Architecture and Language for Multi-threading

Introduction

A set of theoretical tools adequate to obtaining bounds on the performance of dataflow programs is provided through this work. Although previous work exists for obtaining such bounds on the execution of programs described as DAGs, this kind of study is yet to be done for dynamic dataflow execution. Since DAGs are not entirely suitable to represent dataflow programs we first need to introduce a generic model to represent dynamic dataflow programs.

The obvious difference between DAGs and dynamic dataflow graphs is the presence of cycles, which represent dependencies between different iterations of loops. Simply applying the techniques employed to estimate performance of DAGs to DDGs (dynamic dataflow graphs) would not succeed, since problems like finding the longest path or the maximal independent sets cannot be solved in DDGs using traditional algorithms. As follows, it is necessary to develop new theoretical ground that allows us to analyse the concurrency present in a DDG.

While static scheduling of DAGs has been treated extensively by previous researchers, dynamic dataflow has not received similar attention. In this work, we present a theoretical framework for dynamic dataflow analysis and present experimental results on scheduling dynamic dataflow graphs with bounded execution resources.

1.1 Dynamic Dataflow Graphs

This section introduces the generic definition of constricted a Dynamic Dataflow Graph which can be used for further analysis.

1.1.1 Constraints

First of all, we shall present a definition for Dynamic Dataflow Graphs (DDGs) such that the theory presented for them can be applied not only to our work, but also to any model that employs dynamic dataflow execution. For the sake of simplicity of analysis, first we consider a constrained dynamic dataflow program as follows, then we generalize the analysis to more general DDGs:

1. The only kind of conditional execution in loop bodies are loop test.
2. There are no nested loops.
3. Dependencies between different iterations only exist between two consecutive iterations (from the i-th to the (i + 1)-th iteration).

Although these may seem too restrictive at first, we will show throughout this paper that the study of DDGs allows us to analyze an important aspect that is unique to dynamic dataflow, the asynchronous execution of cycles that represent loops.

1.1.2 Definition of a DDG

Also, as mentioned, it is not hard to eliminate these restrictions and apply these methods to programs that cannot be represented in a graph that strictly follows DDG convention. A DDG is a tuple D = (I,E,R,W) where:

- I is the set of instructions in the program.
- E is the set of edges, ordered pairs (a, b) such that $\{a, b\} \subseteq I$, the instruction b consumes data produced by a and b is not inside a loop or a and b are inside a loop and this dependency occurs in the same iteration, i.e. the data b consumes in iteration i is produced by a also in iteration i.
- R is the set of return edges, they are also ordered pairs, just like the edges in E, but they describe dependencies between different iterations, consequently they only happen when both instructions are inside a loop. A return edge $r = (a, b)$ describes that b, during iteration i of the loop, consumes data produced by a in the iteration $i - 1$.
- W is the set of weights of the instructions, where the weight represents the granularity of that instruction, since granularities may vary.

Concurrency analysis

When analysing the potential performance acceleration in a DDG we are interested in two concurrency metrics: the average and the maximum degrees of concurrency. The maximum degree of concurrency is the size of the maximum set of instructions which may, at some point, execute in parallel, while the average degree of concurrency is the sum of all work divided by the critical path. In a DAG, the maximum degree of concurrency is simply the size of the maximum set of instructions such that there are no paths in the DAG between any two instructions in the set. The average degree of concurrency of a DAG can be obtained by dividing it in maximum partitions that follow that same rule and averaging their sizes. In DDGs, however, it is a little more complicated to define and obtain such metrics.

The motivation behind these metrics is straightforward. The maximum degree of concurrency describes the maximum number of processing elements that are useful to execute the graph. Allocating additional processing elements will not lead to further improvement in performance. The average degree of concurrency can be shown to be the maximum speed-up that can be obtained by executing the graph in an ideal system with infinite resources, optimal scheduling and zero overhead. We also present a lower bound for the speed-up. Arora et al. had introduced a lower bound for dataflow execution in DAGs[1], our work extends this to DDGs.

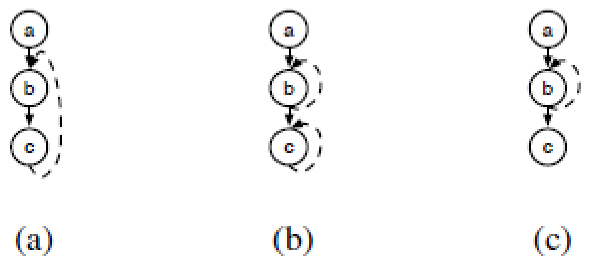


Fig. 2.1: Example of DDGs whose maximum degree of concurrency are 1, 2 and unbounded, respectively. The dashed arcs represent return edges.

2.1 Maximum degree of concurrency in a Dynamic Dataflow Graph

The maximum degree of concurrency in a DDG is called the largest set of instructions that can, at some point, be executing in parallel. In DAGs, it is direct to verify that these independent sets must be such that there is no path in the DAG between any pair of instructions in the set. Since we are dealing with DDGs it is a bit more complicated to obtain this measure, as there may be dependencies (or the lack of them) between different iterations.

2.1.1 Understanding dependencies

In order to delve deeper into the problem we should first introduce additional notation. If v is an instruction inside the body of a loop, we call $v(i)$ the instance of v executed in the $i-th$ iteration of the loop. Let us also indicate by $u(i) \rightarrow v(j)$ that $v(j)$ depends on $u(i)$ and by $u(i) \not\rightarrow v(j)$ that v(j) does not depend on $u(i)$.

In the DDG of Figure 2.1(a), $b(i) \rightarrow c(i)$ and $c(i-1) \rightarrow b(i)$ (observe the return edge, represented as a dashed arc), for $i > 1$, so, transitively $b(i-1) \rightarrow b(i)$. In this case, at any point it is only possible to have exactly one instruction running at a time. We say the maximum degree of concurrency for this DDG is 1. Now consider the DDG of Figure 2.1(b). This time, $b(i) \rightarrow c(i)$,$b(i-1) \rightarrow b(i)$ and $c(i-1) \rightarrow c(i)$, but $c(i-1) \not\rightarrow b(i)$. So, potentially, $b(i)$ and $c(i-1)$ can execute in parallel, yielding a maximum degree of concurrency of 2 for this DDG.

2.1.2 Resource allocation from dependencies

In the DDG of Figure 2.1(c) $c(i)$ depends on $b(i)$ and $b(i)$ depends on $b(i-1)$ (observe the return edge, represented as a dashed arc), but $c(i)$ does not depend on $c(i-1)$. For instance, it would be possible for $c(i)$ and $c(i+1)$ to execute in parallel. As a matter of fact, if b runs fast enough it would be possible to have potentially infinite instances of c executing in parallel. We say that DDGs with this characteristic have an unbounded maximum degree of concurrency, since, given the right conditions, it is possible to have an unbounded number of instructions executing in parallel.

The importance of this metric lies in the decision of resource allocation for the execution of the dynamic dataflow graph. If the maximum degree of concurrency C_{max} of a DDG is bounded, at any time during the execution of the DDG no more than C_{max} processing elements will be busy. Therefore, allocating more than C_{max} PEs to execute that DDG would be potentially wasteful.

2.1.3 Loop unrolling

Now that we have introduced the concept of maximum degree of concurrency in DDGs we shall present an algorithm to calculate it. Basically, the technique employed by the algorithm presented in this chapter consists of applying a transformation similar to loop unrolling to the DDG.

The way we unroll a DDG is simply by removing return edges, replicating the portions of the DDG that are inside loops k times (where k is the number of iterations being unrolled) and adding edges between the subgraphs relative to each iteration in accordance with the set of return edges. The Figure 2.2 clarifies how it is done and in the algorithm we define the process of unrolling formally.

In this example b, c, d and e are instructions inside a loop and a does some initialization and never gets executed again. In order to unroll the loop twice, we remove the return edge, replicate the subgraph that contains b, c, d, e twice, label the instructions accordingly to the iteration number, and add the edges $(e(0), b(1))$ and $(e(1), b(2))$ to represent the dependency correspondent to the return edge. Notice that the graph obtained through this process is not only a DDG, but it is also a DAG, since it has no cycles (because the return edge was removed) and the set of return edges is empty. We shall refer to the DAG obtained by unrolling D as D_k.

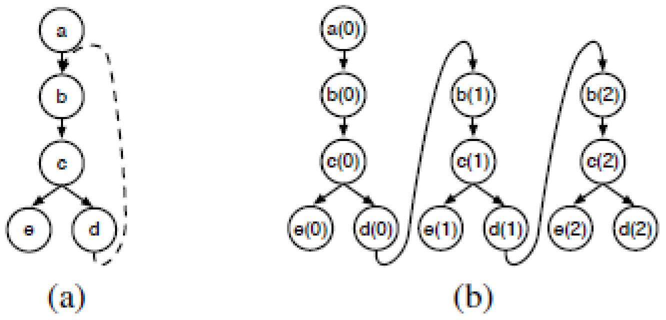


Fig. 2.2: Example of unrolling a DDG. Instructions b , c, d and e are inside a loop (observe the return edge) and a is just an initialization for the loop. The loop is unrolled twice, and the edges (d(0), b(1)) and (d(1), b(2)) are added to represent the dependency caused by the return edge.

2.1.4 Complement path graph

After unrolling the DDG k times, the algorithm proceeds to obtain the maximum degree of concurrency present in the $k + 1$ iterations. This is accomplished by creating a complement path graph of the DAG D_k obtained by unrolling the DDG. A complement-path graph of a DAG $D_k = (I, E)$ is defined

as an undirected graph $C = (I, E^1)$ in which an edge (a, b) exists in E^1 if and only if a and b are in I and there is no path connecting them in D_k. An edge (a, b) in the complement-path graph indicates that a and b are independent, i.e.,$a \not\rightarrow b$ and $b \not\rightarrow a$. Note that as the complement-path graph is constructed as a transformation of D_k, this relation of independence might refer to instances of instructions in different iterations.

In the Figure 2.3 we present as an example the complement-path graph obtained from a DAG. The only thing left for obtaining the maximum degree of concurrency in the $k + 1$ iterations is to find the largest set of independent instructions. Recalling that an independent set of instructions is defined as a set S such that, for all pair of instructions $a, b \in S, a \not\rightarrow b$ and $b \not\rightarrow a$. It follows from this definition that the largest set of independent instructions in D_k is simply the maximum clique in C. Consequently, the maximum degree of concurrency is $\omega(C)$, the size of the maximum clique in C.

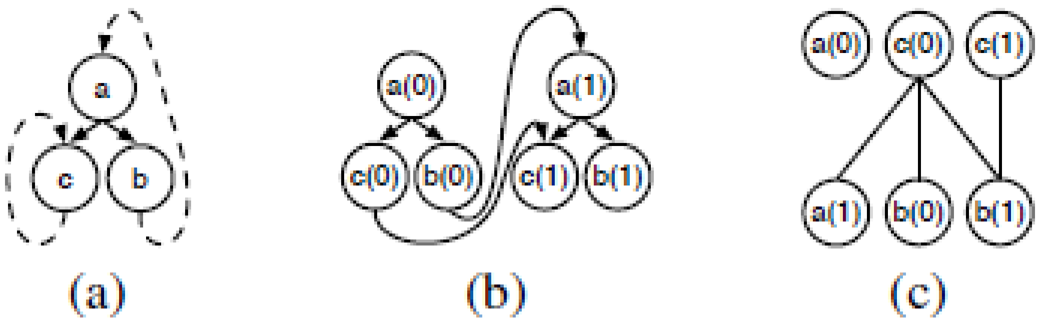


Fig. 2.3: Example of execution of the second iteration of the algorithm (k = 1). In (a) we have the original DDG, in (b) the DAG obtained by unrolling it once and in (c) we have the complement-path graph. The maximum degree of concurrency is 2, since it is the size of the maximum cliques.

2.1.5 Maximum degree of concurrency: Algorithm

The algorithm then proceeds to repeat the procedure described above, replacing D_k with D_{k+1}. The algorithm stops if:

1. The maximum degree of concurrency remains the same as the last step (i.e. the largest independent set in D_k and D_{k+1} have the same size)
2. The maximum degree of concurrency is greater then the number of instructions in D, in this case the maximum degree of concurrency is unbounded, i.e. it will always grow with the number of iterations.

Input: A DDG D = (I,E,R,W)

1. Initialize $k := 0$.
2. Define L as the subset of I that contains only instructions that are inside loops.
3. Unroll D k times by replicating L k times (resulting in $I_k = I \cup \{l_0(1), l_1(1), .., l_(k), .., l_n(k)\}$).
4. Create $E_k = \{(a(i), b(i)) : (a, b) \in E, (a, b), 0 < i < k\} \cup \{(a(i), b(i + 1)) : (a, b) \in R, 0 < i < k - 1\}$
5. Define $D_k = (I_k, E_k, W)$.
6. Create a complement-path graph C_k of D_k defined as $C_k = (I_k, \overline{E}_k; W)$ where $\overline{E}_k = \{(a, b) : a, b \in I_k,$ there is no path in D_k between a and b and vice-versa $\}$.
7. Find a maximum clique in C_k, and define the new maximum degree of concurrency as $\omega(C_k)$, which is the size of a maximum clique in C_k.
8. If $\omega(C_k)$ increased the maximum degree of concurrency, and if $\omega(C_k) < |I| + 1$, make $k := k + 1$ and go back to (2).
9. If $\omega(C_k) = |I| + 1$, the maximum degree of concurrency is infinite (unbounded), otherwise it is $\omega(C_k)$.

2.1.6 Conclusions from the algorithm

Algorithm that returns the maximum degree of concurrency C_{max} of a graph or decides that the degree of concurrency is unbounded.

Lemma 1: If in the $i - th$; $i > 0$ iteration of Algorithm there is an instruction such that $a(i-1) \not\rightarrow a(i)$, the maximum degree of concurrency in the DDG is unbounded.

Proof: If there is an instance $a(i); i > 0$, it means a is inside a loop and was added as part of the unrolling process.If $a(i-1) \not\rightarrow a(i)$, an instance of a does not depend on its execution from the previous iteration, so given the right conditions there can be potentially an unboundend number of instances of a executing in parallel.

Lemma 2: If Algorithm reaches an iteration k such that $\omega(C_k) = \omega(C_{k-1})$, it will not find a clique with size greater than $\omega(C_k)$ and therefore can stop.

Proof: Suppose, without loss of generality, that a maximum clique K in C_{k-1} does not have an instance of an instruction b. If $b(k)$ can not be added to K to create a greater clique, we have that there is an instance $a(i); i < k$; in K such that $a(i) \rightarrow b(k)$, since $b(k) \not\rightarrow a(i)$, because dependencies toward previous iterations are not possible. Consequently, another clique could be constructed such that, in iteration i of the algorithm, $b(i)$ was inserted instead of $a(i)$ and in iteration k there will be a clique where all $u(j); j \geq i$; were replaced by $u(j+1)$ and this clique is greater than K. But, according to the hypothesis, the $\omega(C_k) = \omega(C_{k-1})$, so we must have that $b(i) \rightarrow a(j); i < j$ as well. In that case, it is not possible to obtain a clique with more instructions than K.

Theorem 1: Algorithm finds the maximum degree of concurrency of a DDG, if it is bounded, or decides that it is unbounded if it in fact is.

Proof: Algorithm stops at iteration k if either $\omega(C_k) > |I|$ or if $\omega(Ck) = \omega(C_{k-1})$. In the former case, it decides that the maximum degree of concurrency of the DDG is unbounded and in the latter it returns that $\omega(Ck)$ is the maximum degree of concurrency. First, suppose that for a DDG $D = (I, E, R, W)$ the algorithm obtained $\omega(C_k) > |I|$. In this case, there is at least one instruction a in I such that $a(i) \not\to a(i+1)$, and thus, according to *Lemma 1* the DDG is indeed unbounded.

Now consider the other stop criteria, where $\omega(Ck) = \omega(C_{k-1})$. According to Lemma 2, it is not possible to have a clique with a greater number of instructions than $\omega(Ck)$, thus there is no point in continuing to run the algorithm and $\omega(Ck)$ really is the number of instructions in a maximum clique possible for that DDG.

2.2 Average degree of concurrency (Maximum - speedup)

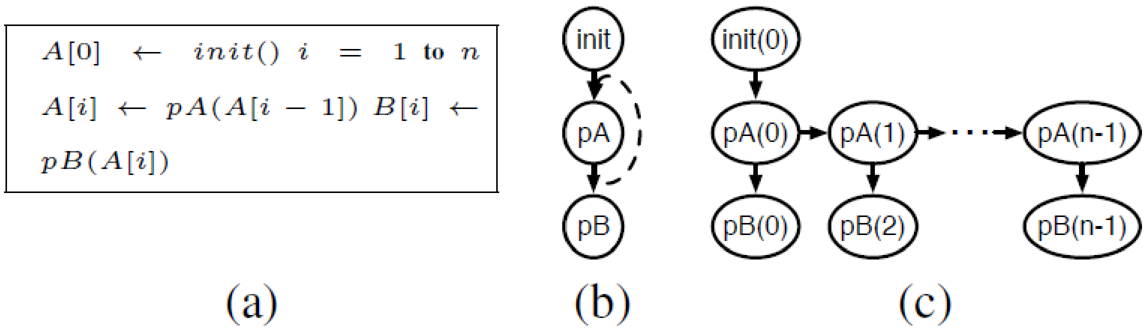
The other important metric we adopt in our analysis is the average degree of concurrency, which is also the maximum possible speed-up. The average degree of concurrency of a program is the average amount of work that can be done in parallel along its critical path. Following the notation adopted in [2] and [3], we say that T_n is the minimum number of computation steps necessary to execute a DDG with n PEs, where a computation step is whatever unit is used for the instructions weights. T_1 is thus the sum of all instruction weights, since no instructions execute in parallel when there is only one PE. T_1 is the minimum number of computation steps it takes to execute a DDG with an infinite number of PEs. T_1 is also the critical path of that DDG, since in a system with infinite PEs the instructions in the critical path still have to execute sequentially. We can thereby say that the maximum speed-up S_{max} of a DDG is T_1/T_∞.

Figure 2.4(a) shows an example of peseudocode, while its corresponding DDG is represented in the Figure 2.4(b). Since we know that the number of iterations executed is n, we can represent the entire execution as a DAG by unrolling the loop n times the same way we did in Algorithm. This process gives us the DAG of Figure 2.4(c). Let us represent with w_{init}, w_a and w_b the weights of the instructions that execute procedures init, procA and procB, respectively. The critical path of the execution is the longest path in the DAG (considering weights), which is $w_{init} + w_a.n + w_b$ and, consequently, the maximum speed-up is:

$$\frac{w_{init} + (w_a + w_b).n}{w_{init} + w_a.n + w_b} \qquad \text{(Equ: 2.1)}$$

Now suppose we do not know the number of iterations that are going to be executed a priori. In this case an exact value for n to calculate maximum speedup is unavailable. On the other hand, if we define a function $S(n)$ equal to the above equation for the maximum speedup, we find that the speedup of this DDG corresponds to a function with asymptotic behaviour, since:

$$\lim_{x \to \infty} S(n) = \frac{w_a + w_b}{w_a} \qquad \text{(Equ: 2.2)}$$



```
$A[0] \leftarrow init()$ $i = 1$ to $n$
$A[i] \leftarrow pA(A[i-1])$ $B[i] \leftarrow$
$pB(A[i])$
```

Fig. 2.4: Example corresponding to a loop that executes for n iterations. In (a) we present the pseudo-code, in (b) the corresponding DDG and in (c) the DAG that represents the execution.

This means that if the maximum speed-up is bounded and converges to a certain value as the number of iterations grows. As will be shown in Theorem 2, this property is true for any DDG, even if the maximum degree of concurrency of the DDG is unbounded. And it is a very important property since it is often the case in which loops will run for a very large number of iterations and this property allows us to estimate the maximum speed-up for them. We shall thus enunciate this property in a theorem and then prove it.

Theorem 2: Given a dynamic dataflow graph $D = (I, E, R, W)$, let D_k be the $k-th$ graph generated by unrolling D as described in Algorithm, W0 the weights of instructions that are inside loops and let L_k be the longest path in D_k, we have

$$S_{max} = \lim_{x\to\infty} \frac{(\sum_{w_i \in W^1} w_i).k}{L_k} \qquad \text{(Equ:2.3)}$$

and S_{max} converges even if C_{max} is unbounded.

Proof: The equation comes directly from the definition of maximum speed-up. Since $L_k \geq min(W).k$, we get

$$\lim_{x\to\infty} \frac{(\sum_{w_i \in W^1} w_i).k}{L_k} \leq \lim_{x\to\infty} \frac{(\sum_{w_i \in W^1} w_i).k}{min(W).k} \qquad \text{(Equ:2.4)}$$

and since the right-hand part of the inequality is bounded, S_{max} must be bounded.

The property enunciated in Theorem 2 is also important because it shows us that one can not expect to get better performance just by adding processing elements to the system indefinitely, without making any changes to the DDG, since at some point the speed-up will converge to an asymptote. Now that we have an upper bound for speed-up, it can also be useful to have a lower bound for it as well, considering that it would provide us the tools to project the expected performance of a DDG. Earlier we discuss an upper bound on the number of computation steps it takes to execute a DDG, which will give us a lower bound for speed-up in an ideal system.

2.3 Lower bound for speedup

Blumofe et al. [5] showed that the number of computation steps for any computation with total work T_1 and critical path length T_1 is at most $T_1 = P + T_\infty$ in a system with P processors and $greedy\ scheduling$. The definition of greedy scheduling is that at each computation step the number of instructions being executed is the minimum between the number of instructions that are ready and the number of processors available.

This bound fits our target executions, since we know how to calculate T_1 and T_∞ for DDGs and dataflow runtimes with work-stealing can be characterized as having greedy scheduling policies. We therefore shall use that upper bound on computation steps to model the performance of DDGs.

Actually, since we are interested in modeling the potential speed-up of a DDG, the expression we must use is the total amount of work T_1 divided by the upper bound on computation steps, which will give us a lower bound on speed-up:

$$S_{min} = \frac{T_1}{\frac{T_1}{P} + T_\infty} \tag{Equ:2.5}$$

The upper bound on time proved by Blumofe et al. takes into consideration the number of computation steps it takes to execute the work using a greedy scheduling. Basically, it means that overheads of the runtime are not taken into account and there is the assumption that the system never fails to schedule for execution all instructions that are ready, if there is a processor available. Consequently, this lower bound on speed-up S_{min} will serve us more as a guideline of how much the dataflow runtime overheads are influencing overall performance, while the upper bound S_{max} indicates the potential of the DDG. If the performance obtained in an experiment falls below S_{min} that indicates that the overheads of the runtime for that execution ought to be studied and optimized.

Evaluating concurrency metrics

3.1 Comparing different pipelines

In this section we will apply the theory introduced to analyze DDGs that describe an import parallel programming pattern: pipelines. We will take the DDGs of two different pipelines that have the same set of instructions and calculate the maximum degree of concurrency C_{max}, the maximum speed-up S_{max} and the minimum speed-up S_{min} for both of them. In the end of the section we will see that the theoretical bounds obtained correspond to the intuitive behaviour one might expect just by reasoning about both pipelines.

3.1.1 Pipeline 1

Figure 3.1(a) shows the DDG for the first pipeline and figures 3.1(b) and 3.1(c) represent the stages of the last iteration of Algorithm for this DDG. In Figure 3.1(b) the DDG was unrolled four times and in Figure 3.1(c) the complement-path graph was obtained. Since the clique number of C_4 is the same as the clique number of C_3, the maximum degree of concurrency is $C_{max} = \omega(C_3) = 3$. To obtain S_{max} we apply the same principle as dicussed before. If we consider $w_b > w_a > w_c$, where w_u is the weight of an instruction u:

$$S_{max} = \lim_{x \to \infty} \frac{(w_a + w_b + w_c).k}{w_a + w_b.k + w_c} = \frac{w_a + w_b + w_c}{w_b} \qquad \text{(Equ:3.1)}$$

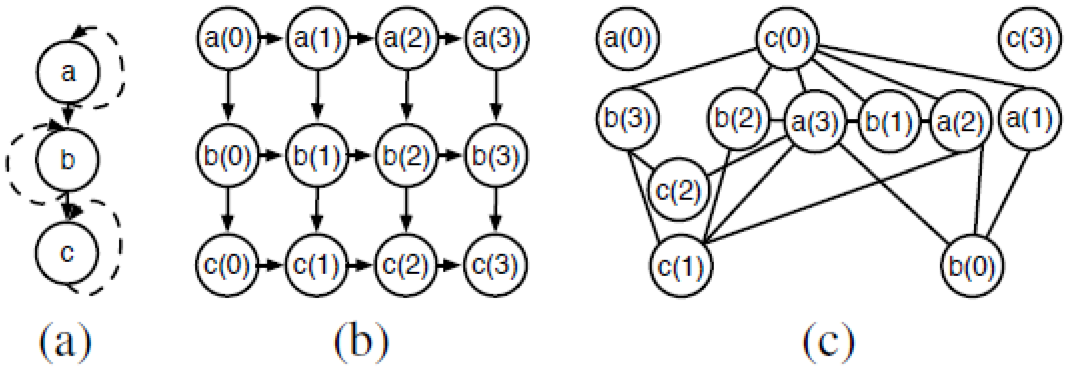


Fig. 3.1: Example of execution of Algorithm in which the maximum concurrency converges.

3.1.2 Pipeline 2

Now we consider, in Figure 3.2, the same instructions connected in a pipeline, but this time the second stage of the pipeline (instruction b) does not depend on its previous execution. In this case, Algorithm will reach $\omega(C_3) = 4$ and decide that C_{max} is unbounded. We then calculate S_{max}, obtaining:

$$S_{max} = \lim_{x \to \infty} \frac{(w_a + w_b + w_c).k}{w_a.k + w_b + w_c} = \frac{w_a + w_b + w_c}{w_a} \qquad \text{(Equ:3.2)}$$

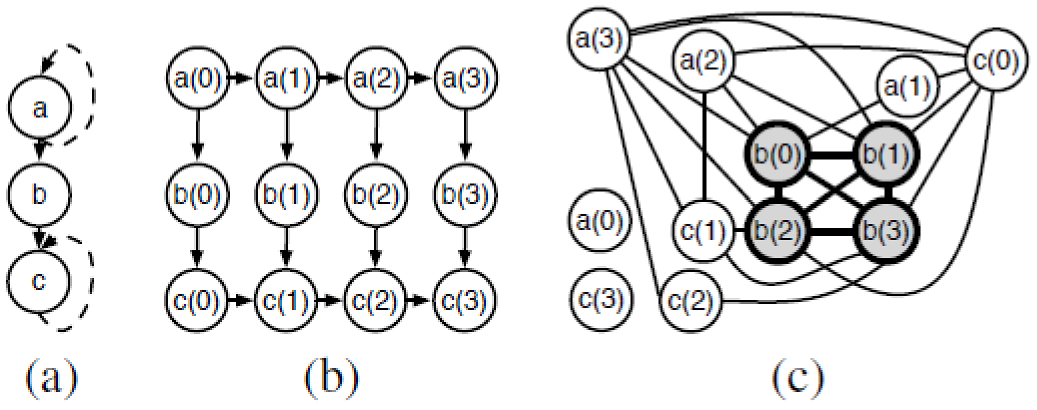


Fig. 3.2: Example of execution of Algorithm in which the maximum degree of concurrency does not converge.

And since $w_a < w_b$, the S_{max}, the average degree of concurrency for this DDG is greater than the average degree of concurrency in the the DDG of Figure 3.1. This corroborates the conclusion one might come to intuitively, since it makes perfect sense that a pipeline in which the slowest stage does not depend on its previous execution will most likely execute faster than one that is identical but has this restriction.

3.2 Performance evaluation

In this section we are going to present a series of experiments in which we compared the actual speed-up obtained for benchmark executions with the theoretical bounds for the DDG.

In a first set of experiments we devised three artificial benchmarks representing different categories of DDGs. The choice for the adoption of artificial benchmarks in these experiments was due to the precise parametrization of the tasks weights and the freedom to define the topology of the graphs. Such decision was important since our focus was to also provide empirical proof of the theory presented in this paper and artificial benchmarks allow us to put to test various dataflow scenarios, which would take a long survey of real applications to find ones that are adequate to the points we want to prove. Next we perform experiments with two real applications: Ferret, from the PARSEC Benchmark Suite [6] and a classic implementation of Conways Game of Life. All results presented in this section came from executions of each benchmark in the Trebuchet Dataflow Runtime [7] running on a Intel Xeon R machine with 36 cores.

Static scheduling: A static scheduling algorithm is presented for off-line scheduling of tasks in distributed hard real-time systems. The tasks considered are instances of periodic jobs and have deadlines, resource requirements and precedence constraints.

The Trebuchet runtime acts as facilitator for the parallelized execution in accordance to the dataflow paradigm. However, the same results can be expected if the benchmarks are implemented and executed using any other runtime or library that orchestrates parallel computation in a dataflow fashion. The only variation in performance for different implementations would be due to the overheads of the orchestrator itself, however, the trends in the behaviour of the results should be the same.

3.3 Eliminating the restrictions

The reason we chose not to relax the restrictions in the previous sections was to ease the formalization of the model and the related proofs. In this section, however, we will introduce how each of the three restrictions can be eliminated.

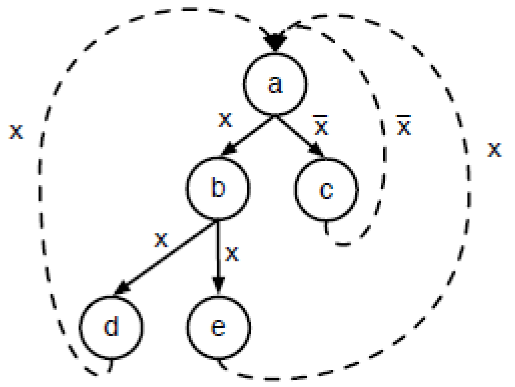
3.3.1 Conditional execution

The first restriction has to do with conditional execution. In order to allow DDGs to have conditional execution, just like any dataflow graph, we will have to take into consideration each possible execution path when calculating T_1 and T_∞ for the DDG. Basically, the critical path T_∞ and the total work done T_1 both depend on the boolean control expressions evaluated during execution (i.e. the equivalent of the branches taken/not taken in Von Neumann execution). To address this first restriction we extend the DDGs with the concept of Conditional DDGs (CDDG).

Let a CDDG be a tuple $D = (I, E, R, W, X, \phi)$, where:

1. I, E, R, W are the same as in the original DDG definition
2. X is a set of boolean variables used to express the logical conditions for data to be sent through each edge.
3. $\phi : E \rightarrow F(X)$ attributes a logical expression to each edge of the graph.

With this definition, we have that the logical expression for an edge $e \in E$ is the boolean function $\phi(e) \in F(X)$, where $F(X)$ is the set of all boolean functions using the variables in X. This way, if $\phi(e)$ evaluates to True, data will be carried through edge e. Figure 3.3 shows an example of a loop with an if statement inside its body, which causes the subgraphs corresponding to the then and else clauses to be mutually exclusive.



e	$\phi(e)$
(a, b)	x
(a, c)	$\overline{x}$
(b, d)	x
(d, a)	x
(e, a)	x
(c, a)	$\overline{x}$

Fig. 3.3: Example of conditional DDG representing a loop with an if statement inside its body. Since y = x, we have mutually exclusive sets of instructions in each iteration, which represent the then and else clauses.

In order to apply our methodology to Conditional DDGs, we also introduce the concept of projections of CDDGs. A projection of a CDDG $D = (I, E, R, W, X, \phi)$ is the attribution of a value $\alpha \in \{0, \}$ to a variable $x \in X$ in an iteration i (represented as $D \mid_{x(i)=\alpha}$). For instance, if we unroll the CDDG k times, we can attribute a boolean value to a variable $x \in X$ in each of the k iterations. A complete projection of a CDDG unrolled k times is a projection such that all variables have boolean values attributed to them. Notice that complete projections can be obtainted by accumulating the projection operation, i. e., $D \mid_{x0=\alpha0}\mid_{x1=\alpha1} .. \mid_{xn=\alpha n}$, where each xi is a variable in the unrolled CDDG. For the sake of simplicity, we define the function $\psi : X_k \rightarrow \{0, 1\}$ which attributes a boolean value to each variable in X_k (the set of variables obtained when unrolling the CDDG k times). This way, we say that, if $\phi \mid_{\psi} (e_k) = 1$, data will be carried through edge e_k in the unrolled CDDG because its condition evaluates to true.

With this definition of complete projections, we can reduce unrolled Conditional DDGs to unrolled DDGs simply by applying a complete projection to D_k, a CDDG unrolled k times, and then removing all edges e_i such that $\phi \mid_\psi (e_i) = 0$ and all instructions I_k that become unreacheable in the complete projection (meaning that such instance of the instruction would not be executed in the scenario represented by the attribution function ψ).

Now, we can adopt these definitions to adapt Algorithm III for Conditional DDGs. The basic idea is to consider all possible complete projections $D_k \mid_\psi$ in each iteration and reduce the conditional graph to a simple dataflow graph by removing edges whose expression $\phi \mid_\psi$ evaluates to 0. Having done that, we pick the unrolled graph with the maximum $\omega(C)$ and consider that the maximum degree of concurrency for the current iteration. The same rules for continuing onto the next iteration of the algorithm or for terminating it apply, as we will show below.

3.3.2 Dependencies over multiple iterations

The restriction which states that dependencies can only be carried from one iteration to the next one can be relaxed by starting with the DDG unrolled as many times as the longest distance (in iterations) among the loop carried dependencies in the graph. Consider the loop and the corresponding DDG in Figure 3.4. The integers used as labels for the return edges represent the distance (in iterations) corresponding to the edge , i.e., if the label is d the dependency is carried from iteration i to iteration $i + d$.

Figure 3.4 shows the first step for adapting both Algorithm III and the calculation of the bounds for speed-up to overcome the restriction on the distance of loop-carried dependencies. Since, the longest distance in inter-iteration dependencies is 3 (the return edge from c to c itlsef), we unroll the DDG 3 times.

For nested the loops, a simple variation of the same approach can be adopted. Since nested loops can be represented as flat loops that carry dependencies over a iteration distance equal to the number of iterations in the inner loops.

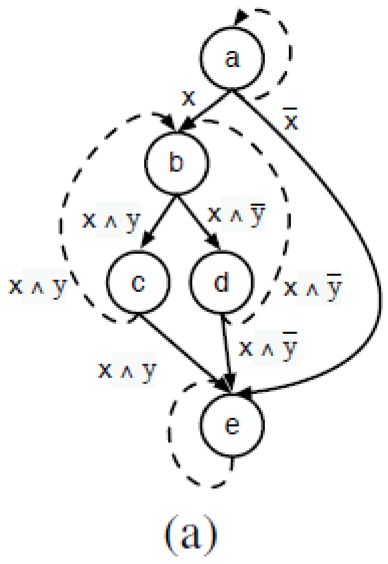
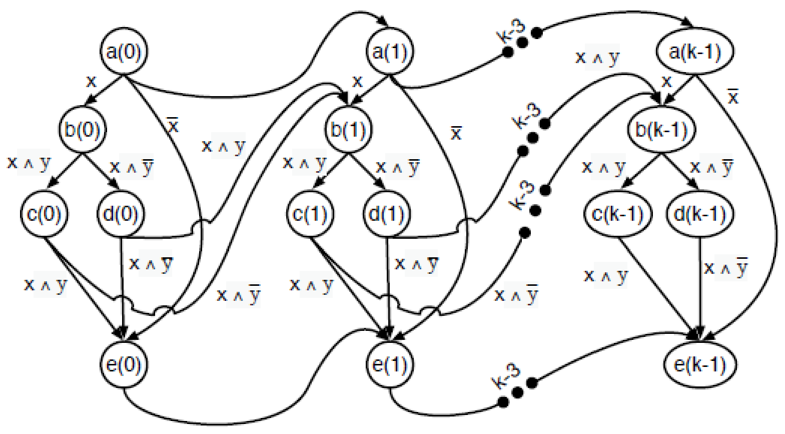


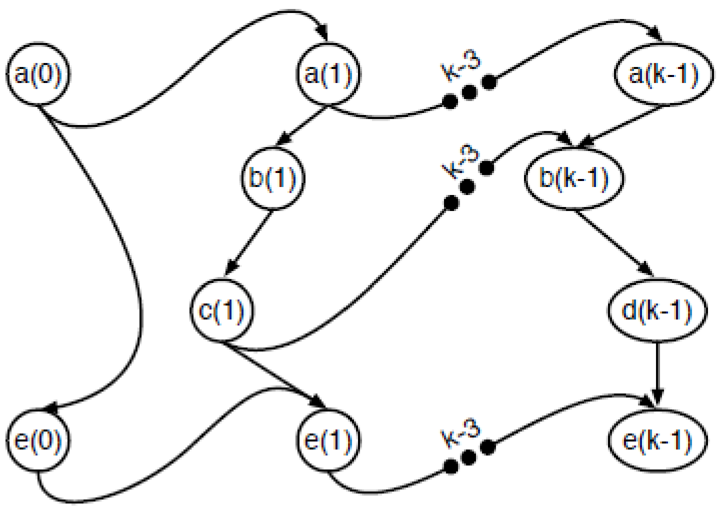
Fig. 3.4: A Conditional DDG with two nested loops, the innermost inside the then-clause of the outmost.

3.4 Working with unrestricted Dynamic Dataflow Graph

Considering the figure 3.5(b), the DAG is formed from it by unrolling it k times. That results in a DAG after applying the conditions. Hence a completely unrolled graph of the DDG is obtained.



(b)



(c)

Fig. 3.5: Graph obtained after unrolling it k times

Figure 3.5 is another example of a dynamic dataflow graph that is considered in which the dependencies are carried over a distance more than one iteration. Hence implying the lifted restriction over the DDG definition.

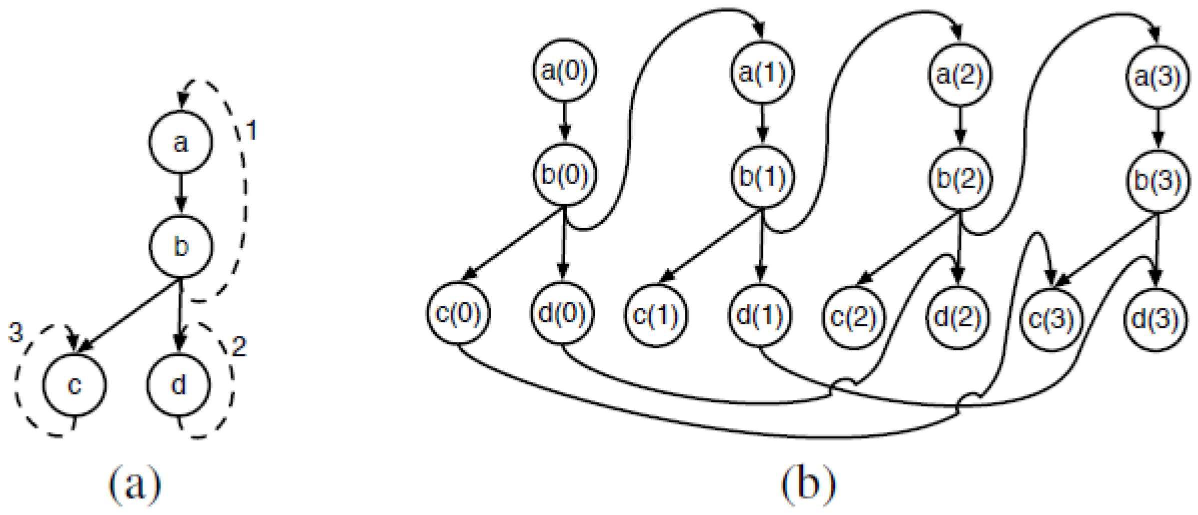


Fig. 3.6: DDG distance carried over more than one iteration.

Conclusion

In this work we present concurrency analysis techniques that extend prior work that focused only on DAGs. We presented a definition of Dynamic Dataflow Graphs (DDGs), which we used as our model of computation for the analysis. This new model of computation allowed us to abstract details of dataflow implementation, since DDGs are more simple and easier to analyze than TALM graphs, for instance, and gain important insight into the behaviour of dataflow parallelization. We showed how to obtain two important metrics via analysis of DDGs: the maximum speed-up (or average degree of concurrency) Smax and the maximum degree of concurrency Cmax. We also were able to present an important result, the fact that the former is bounded even if the latter is not. Our experiments with a benchmark that represented the streaming parallel programming pattern showed us that our analysis indeed is able to model and predict the performance behaviour of applications parallelized with dataflow execution. An important lesson learned with this work: all the information that describes the parallel programming pattern in the program is in the dataflow graph, i.e. the data dependencies in the program. Therefore, high-level annotations or templates that specify what programming pattern is adopted are not necessary. For instance, using our techniques it is not necessary to specify (or use a special template) that a certain application has a pipeline pattern, the programmer just has to describe the dependencies and the static scheduling algorithm will be able to determine to number of PEs to allocate (using Cmax) and how to spread the pipeline stages among them.

REFERENCES

[1] R. D. Blumofe and C. E. Leiserson, Scheduling multithreaded computations by work stealing, J. ACM, vol. 46, no. 5, pp. 720748, Sep. 1999.

[2] D. Eager, J. Zahorjan, and E. Lazowska, Speedup versus efficiency in parallel systems, IEEE Transactions on Computers, vol. 38, no. 3, pp.408423, Mar. 1989.

[3] B. Kumar and T. A. Gonsalves, Modelling and analysis of distributed software systems, in Proceedings of the Seventh ACM Symposium on Operating Systems Principles, ser. SOSP 79. New York, NY, USA: ACM, 1979, pp. 28.

[4] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, Thread Scheduling for Multiprogrammed Multiprocessors, Theory of Computing Systems,vol. 34, no. 2, pp. 115144, Jan. 2001.

[5] W. F. Boyer and G. S. Hura, Non-evolutionary algorithm for scheduling dependent tasks in distributed heterogeneous computing environments, Journal of Parallel and Distributed Computing, vol. 65, no. 9, pp. 10351046, Sep. 2005.

[6] M. Lombardi and M. Milano, Scheduling conditional task graphs, in Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming, ser. CP07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 468482.

[7] G. C. Sih and E. A. Lee, A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures, IEEE Transactions on Parallel and Distributed Systems, vol. 4, no. 2, pp. 175187, Feb 1993.

[8] M. T. Anticona, A GRASP algorithm to solve the problem of dependent tasks scheduling in different machines. Boston, MA: Springer US, 2006, pp. 325334.