ON THE LOSS OF PARALLELISM BY IMPOSING SYNCHRONIZATION STRUCTURE

Arturo González Escribano¹, Valentín Cardeñoso Payo¹, Arian J.C. van Gemund²

¹ Dept. de Informática, Universidad de Valladolid.
Prado de la Magdalena s/n, 47011 - Valladolid, Spain
Phone: +34 83 423162  eMail: arturo@infor.uva.es

² Dept. of Electrical Engineering, Delft University of Technology.
P.O.Box 5031, NL-2600 GA Delft, The Netherlands
Phone: +31 15 2786168  eMail: a.vgemund@et.tudelft.nl

Keywords: parallel programming paradigms, task synchronization, parallelizing compilers, graph theory.

Abstract

Recently a new parallel programming model has been presented that imposes synchronization restrictions in order to allow for fully automatic, retargetable program optimization. The motivation for the model is the conjecture that in practice the loss of parallelism due to the inherent synchronization restrictions is less than a factor of 2. In this paper we provide compelling evidence in favor of this conjecture, based on the results of a largely empirical investigation into the ratio between the critical paths of unstructured task graphs and their approximations under the restricted synchronization model.

1 INTRODUCTION

A long-term goal in program compilation for parallel and distributed systems is the development of compile-time techniques that perform optimizations automatically, without requiring costly user interaction, thus providing portability as well as performance. For example, in the case of a distributed-memory target machine the compiler should be able to automatically determine an acceptable data partitioning without having the average user supply the distribution directives which would require a major understanding of the complex interplay between program and machine. In essence, the compiler must be able to reason about the effects of program optimizations merely based on a performance model of the machine, rather than through heuristics that are usually hard-wired in the compiler.

Although currently a range of fine, compile-time cost prediction methods exist (e.g., [1, 2, 4, 9]), either the underlying analysis (and the associated parameter space) is targeted to a particular type of parallel architecture, or the technique is not intended to produce reliable estimates for the temporarily erratic solutions that may be generated in the course of the optimization. The generally limited ability of compile-time cost estimations to be extremely low-cost and fully reliable at the same time, is closely related to the choice of programming model. Traditionally, parallel programming models are focused on expressiveness, i.e., the ability to express the inherent parallelism within the algorithm to the ultimate, rather than performance analyzability, which in the automatic optimization context means the ability to derive sufficiently cheap yet reliable cost estimates.

Aimed to achieve a better trade-off between expressiveness of parallelism and performance analyzability, a new parallel programming model has been introduced, called SPC [5]. One of SPC's features is that the algorithm (and machine) must be described in terms of a series-parallel (SP) computation, which implies structure with respect to the synchronization patterns that are possible (only SP task graphs). By imposing these specific restrictions in the synchronization structure, a performance analyzability is achieved that allows for reliable, closed-form, analytic cost estimation [6]. This, in turn, unlocks the potential of automatic program optimization which is the ultimate objective [5]. The aforementioned trade-off is based on the following conjecture [5] which states that the loss of parallelism when programming according to the SPC model is typically limited to a constant factor of 2, compared to the unrestricted case.

Conjecture 1.1 Let G be a practical parallel computation. Let $T_G$ denote the minimum critical path of G when expressed in a programming model that does not impose restrictions with respect to the synchronization constraints inherent in the problem. Let $G'$ be an SPC...
program that also computes $G$ and which has a critical path $T_G$. It is conjectured that

$$\forall G \exists G': \frac{T_{G'}}{T_G} \leq 2$$

The idea behind this conjecture is briefly summarized in the following (see [5] for an elaborate treatment). First, SP Cencloses all data parallel computations (i.e., that correspond with task graph representations that have SP form) which represent a large group of applications. Second, the group of algorithms that necessarily correspond to task graph representations that have a non-SP (NSP) form can be transformed to SP versions that do not violate the original synchronization constraints and have a critical path less than a factor 2 of the original path for practical workloads.

For example, consider some parallel computation of which the corresponding task graph is shown in Figure 1(a). Expression of this computation in SPC re-

![Figure 1: NSP problem and its SP approximation](image)

quires the NSP graph to be approximated by an SP graph, such as the one shown in Figure 1(b). One can easily verify that the increase of the critical path is at most a factor 2 (for an extremely improbable workload distribution). This property also holds for computations involving larger numbers of tasks.

In this paper we present the results of a largely empirical investigation into the truth of the above conjecture. More specifically we present the following contributions:

- We present a polynomial-time/space algorithm that approximates any NSP task graph in terms of an SP graph that is “close” to the original in terms of its critical path.
- We show for a number of well-known topological graph classes that the algorithm yields SP graphs that are well within a factor 2 critical path increase.
- We show that for some graph topologies the critical path increase will only occur under pathological workload distributions that will never occur in practice.

These results provide compelling evidence in favor of the above conjecture and to the best of our knowledge have not been presented before. It is important to note that the factor 2 upper bound associated with the use of SPC may seem as a drastic performance reduction when compared to alternative programming models with an expressiveness comparable to, e.g., CSP. We believe, however, that this sacrifice is justified by the potential of fully automatic, cost-driven optimization, which does not come with alternative models that have larger expressiveness. In many cases the consequent portability will greatly outweigh the effort spent on realizing $T_G$ every time through manually porting the code to the latest machine architecture.

The paper is organized as follows. In Section 2 we present the algorithm. The experimental results are presented in Section 3. Section 4 concludes the paper.

2 THE ALGORITHM

2.1 CONCEPTS AND NOTATIONS

Before we describe the algorithm we first present some notations and basic concepts about NSP and SP graphs. For a complete treatment see [3].

2.1.1 BASIC CONCEPTS AND REDUCTION OPERATIONS

A directed graph (dg) is $G = (V, E)$; where $V$ is a finite set of vertices (or nodes) and $E$ is a finite set of directed edges or tuples $e = (v, v')$. Multiple edges between the same nodes are allowed. A two-terminal directed acyclic graph or stdag is a dg without cycles and with only one root and only one leaf. Any task graph can be presented as an stdag. We classify the nodes of task graphs in three broad categories depending on their synchronization role in the graph:

- $F_{nodes}(G)$ or nodes with more than one successor.
- $J_{nodes}(G)$ or nodes with more than one predecessor.
- $JF_{nodes}(G)$ or nodes that are F and J nodes at the same time.

Two mapping operators that reduce the SP (series-parallel) structures in a graph to a single edge have been proposed [8]. We call them $Series reduction operator$ or $\dagger$ and $Parallel reduction operator$ or $\ddagger$. The $\dagger$ operator is used in a node that has only one arriving edge and one leaving edge, and substitutes the node and both edges for only one new edge between the predecessor and the successor on the original node. The $\ddagger$ is used on a set of direct edges between two nodes, and maps the whole set to only one edge between the same nodes.

The Minimal Reduction Graph of an stdag is the graph that results after using all the possible series and
parallel reductions on the graph. We write it as \([G]\).

When we use reduction operations in a graph, each edge can represent an SP structure. We use an annotation system regarding the edges to keep track of the last edges of the structure that a new one represent. So, we can reallocate the end point of the structure to resynchronize it as a whole.

A Trivial Graph or \(G_1\) is a graph with only two nodes and only one edge. An stdag is an SP graph iff it can be reduced to a trivial graph (only one edge), through series and parallel reductions (using \(\uparrow\) and \(\downarrow\) operators). The SP Branches (of an stdag \(G\)) are the subgraphs of \(G\) that are themselves SP graphs. The Longest SP Branches or LSP branches are the largest subgraphs of the graph that can be reduced to an edge. In a minimal reduction graph, they are reduced to an edge.

### 2.2 NSP PROBLEMS

An NSP problem arises when there is an edge (or SP branch) that crosses from one SP branch to another, thus causing a non-SP (NSP) synchronization.

We distinguish only two different kinds of NSP problems. All the NSP problems are compositions of these basic ones and can be solved by a combination of the resolution principles associated with the two basic cases.

![Figure 2: Second basic NSP problem and its SP approximation](image)

The first case is a simple crossing NSP problem, see Figure 1(a), which can be recognized because there is an F node which has branches to J nodes that are in transitive relation. The way to approximate the structure to an SP version is to resynchronize (move) the branches of this F node to a new dummy synchronization node (with zero workload), and add an edge from this new node to the first J node related, in topological order (see Figure 1(b)).

![Figure 2(a) and 2(b)](image)

The second case is a multiple crossing NSP problem, see Figure 2(a), where there is at least one F node with J nodes related that are not in transitive relation. In this multiple case, the problem involves all the J nodes that receive branches from that F node, and all the F nodes that are origins of branches to any J node involved, recursively. We solve the problem yielding an SP approximation by synchronizing all the branches of each related F node by a new dummy synchronization node, and then, adding edges from this node to all J-nodes involved (see Figure 2(b)).

### 2.3 ALGORITHM PRINCIPLES

#### 2.3.1 STRATEGY

The algorithm transforms any NSP graph into an SP graph approximation purely based on the SP topology. No knowledge on the delay times of the nodes is exploited.

In each iteration the algorithm resolves one NSP problem as follows (a formal description is presented in [3]):

- **a. Obtain LSP branches information.** We compute the minimal reduction graph from the original one, with annotated edges.

- **b. Select an NSP problem.** We search in the minimal reduction graph for an F node of which all its LSP branches go to J nodes (see [3] for more details).

- **c. Locate the F and J nodes related.** We call these nodes F and J handles of the problem. We use two sets (F set and J Set) to detect them. Initially, we assign the chosen F node to the F set. Then, we add every J node at the end of LSP branches of each F node in the F set to the J set. In addition, we add the F or J nodes at the start of the branches of each node in the J set to the F set. We mark any explored node for avoiding use it again. This procedure is repeated until all the F nodes on the F set and all the J nodes on the J set have been processed. Some nodes are removed from the sets in special circumstances. See the example below and [3] for a full description.

- **d. Resolve the NSP problem.** We synchronize all the branches of each F node in the F set over a new dummy synchronization node. Then, this node is connected to each J node on the J set.

The complexity of this algorithm is \(O(n^2)\) in space, and \(O(n^5)\) in time. See [3] for a complete description of the algorithm, and its implementation.

#### 2.3.2 EXAMPLE

We demonstrate the way our algorithm works with an example graph. The minimal reduction graph of the graph is shown in Figure 3(a).

The evolution of the J and F sets during the problem handle detection phase can be seen in Table 1.
The first column of this table (checkpoint number N) describes the event sequence. The algorithm would then proceed as follows:

We can choose as initial NSP problem either nodes 2, 3, 4 or 5. All of them have only LSP branches to J nodes. If we suppose that node 5 is the initial NSP problem, we would add it to the F set and explore it to locate its related J nodes 7 and 9, which should be included in the J set (checkpoints 1 → 2). In checkpoint 3, we explore the J nodes just added in the previous step and their related F nodes 3, 4 and 8—which is taken as an F handle because it is also the origin of an LSP branch. In checkpoint 4, we explore the next unexplored node in the F set, e.g. 3, and a new J node is obtained for the J set, namely node 5. In checkpoint 5, we test transivities in J set, which implies the elimination of nodes 7 and 9, since node 5 represents their transitive closure and is the only one kept in the J set. In checkpoint 6, we detect how node 5 is also present in the F set, which represents a J-F combination to be ruled out from the F set. In checkpoint 7, the F handle 8 is taken out from the F set because there is no J node related to it in the J set. In checkpoint 8, we explore the next F node (4) and introduce a new J node in the J set (6). As a consequence, a new F node has to be added to the F set (2) after the exploration of this last J node; when this new F node (2) is explored, a new J node is added to the J set (8) which is then ruled out because of the transitivity relation with node 6. When we reach this point, we are at checkpoint 11 and there are no more J or F nodes unexplored, which concludes the search of the handles.

As a final result, the F handles for this problem are nodes 3, 4, and 2. So we synchronize the LSP branches of these nodes over a new dummy node, from which we add edges to the J handles detected, i.e., the J nodes 5 and 6. Thus the solution is the graph in Figure 3(b).

Since this is already an SP graph no more iterations are needed.

### Table 1: Detecting the Problem Handles

<table>
<thead>
<tr>
<th>N</th>
<th>F set</th>
<th>J set</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>7,9</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>7,9</td>
</tr>
<tr>
<td>3</td>
<td>5,4,8</td>
<td>7,9</td>
</tr>
<tr>
<td>4</td>
<td>5,4,8</td>
<td>7,9,5</td>
</tr>
<tr>
<td>5</td>
<td>5,4,8</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>3,4,8</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>3,4</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>3,4</td>
<td>5,6</td>
</tr>
<tr>
<td>9</td>
<td>3,4,2</td>
<td>5,6</td>
</tr>
<tr>
<td>10</td>
<td>3,4,2</td>
<td>5,6,8</td>
</tr>
<tr>
<td>11</td>
<td>3,4,2</td>
<td>5,6</td>
</tr>
</tbody>
</table>

3 EXPERIMENTAL RESULTS

3.1 RANDOM EXPERIMENTS

The first experiment that has been performed is a measurement of the ratio in the critical path between randomly generated NSP graphs and the SP approximations as generated by the algorithm. For these experiments we have chosen a uniform distribution for the delay values of the nodes. The experiment comprises four runs using 20,000 random graphs consisting of 10, 20, 50, and 100 nodes.

The results clearly show the following facts:

1. For the majority of NSP graphs the relative loss of parallelism due to imposing SP structure is far less than 2.
2. The percentage of NSP graphs, for which the critical path increase is negligible after the transformation grows with the number of nodes.
3. There are still a few cases in which the algorithm can not produce an SP version where the critical path increase is less than a factor 2. However, the percentage of graphs is negligible and pertain to cases that will not occur in practice (this phenomenon is discussed later on).

3.2 SPECIFIC TOPOLOGIES

The second experiment has been performed using a number of specific topologies originating from typical parallel computations [7]. The task graphs, shown in Figure 4, correspond to

a) **neighbor synchronization** as typically found in iterative solutions for linear equation systems.

b) **macropipelining**, typically associated with wavefront computations and various other forms of task parallelism.

c) **fork-join/broadcast-reduction**, typical for periodically parallel solution techniques such as LU factorization.
d) paired synchronization, associated with the parallelization of loops where a part of the loop body contains loop-carried dependencies.

![Diagram](image)

**Figure 4: Specific topology types**

For this experiment, random uniformly distributed delay times have been used for the task nodes on one example graph of each type. The number of random delay experiments was 5000 for each one. Some topologies are able to show pathologically bad synchronization conditions for at least 15 nodes, as we discuss later on, so we have chosen examples with about 20 nodes each. The critical path increment has been scored both for the solution proposed by the algorithm and by a well-known manual solution [7].

### 3.2.1 RESULTS

The neighbor synchronization problem (Figure 4(a)) has clearly defined layers of multiple NSP problems. The algorithm detects them and effectively synchronizes each layer by including a barrier synchronization between them just like in the well-known manual solution when using an SP programming model [7]. The performance loss due to the additional synchronizations (Figure 5(a)) is quite small, although in principle there are unbalanced load configurations for the tasks that can make the solution have a critical path more than twice the original, in practical computations it will never happen. (This is discussed later on).

In the macropipeline (Figure 4(b)) the algorithm produces a strange synchronization. The algorithm synchronizes branches only to J nodes, so it does not detect correctly the levels in the macropipeline. We are studying a new version of the algorithm that uses all branches to produce more logical results. Yet, the performance loss is not much higher than in the manual SP solution, which includes barrier synchronizations between each layer. Again, the performance loss is quite small (Figure 5(b)).

For the fork-join/broadcast-reduction graph we have chosen an example for a generic LU-decomposition graph of a 6x6 matrix. The graph has multiple NSP problems clearly organized in layers. As a consequence, the solutions synchronize each layer in a natural way, producing very short increments in the critical path (Figure 5(c)). In this model the algorithm even produces a slightly better solution than the manual SP solution. Some inherent parallelism is detected automatically that it was not exploited in the manual solution.

Finally, the paired synchronization graph (Figure 4(d)) presents only one multiple NSP problem. It is solved in only one pass of the algorithm (i.e., by including only one barrier) so it is theoretically impossible to generate a solution with a critical path more than twice the original [3]. All the graphs of this kind are reduced through series and parallel reduction operations to the same multiple NSP problem. As expected, the performance loss due to imposing SP structure is quite small (Figure 5(d)). In this case the automatic solution has more chances to slightly increase the critical path of the graph than the manual one.

A more elaborated analysis of all results for these examples and other tests, appear in [3].

### 3.2.2 DISCUSSION

Although the above results clearly indicate that the performance loss due to imposing SP structure is indeed quite small, there exist cases where the critical path ratio is larger than a factor 2. The most notable case is the macropipeline where for very contrived workload distributions the ratio is even unlimited (zero delays on all nodes except for a small subset within a very specific topological setting). Apart from the fact that these cases in no way represent practical computations, it should be noted that, especially in the macropipeline case, the critical path ratio between the NSP and SP version does not accurately reflect an answer to the original question, i.e., what is the performance loss between the intended synchronization structure and an SP solution. In the case of the macropipeline the intended synchronization structure is dynamic whereas the NSP graph is an overspecified static version [5]. In contrast to the SP graph, the appropriate (dynamic) solution can be readily expressed by a simple SPC program. Thus, instead of representing evidence to possibly falsify the conjecture, the macropipeline is an excellent example in favor of the conjecture as the actual performance loss is in fact zero [5]. Of course, there still exist NSP graphs that pertain to static computations of which the NSP graph is indeed an appropriate representation of the intended synchronization structure (e.g., the neighbor synchronization topology). Again, however, the cases in which the critical path increment is greater than a factor 2 correspond to work load delays which in no way reflect practical computation —note that many nodes represent similar computations and, thus, have "comparable" workloads.
4 CONCLUSION

In this paper we have studied the conjecture that the practical loss of parallelism due to imposing the SPC programming model is limited (by far) within a factor 2. By developing an algorithm that automatically transforms NSP graphs to SP graphs, and by applying this algorithm to tens of thousands of random graphs as well as specific topologies, we have found compelling evidence in favor of the conjecture. The cases where the factor 2 is exceeded either correspond to dynamic computations where the NSP graph representation is inherently inappropriate, or to cases where the workload distribution over the nodes is extremely improbable (i.e., practically non-existent). As mentioned in the previous section, the relationship between (static) NSP and SP graphs represents only a subspace of the general question whether SPC has enough expressive power to capture the (dynamic) parallelism inherent in practical computation. Hence, future work will include an investigation into the inherent synchronization structure of problems at a more abstract level in order to better determine the actual performance loss associated with mapping problems onto parallel programming models.

References


Figure 5: Performance loss distributions for specific topologies