An Efficient Heuristic for Code Partitioning

Moez Ayed and Jean-Luc Gaudiot

CENG 96-37

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213) 740-4484

November 1996

An Efficient Heuristic for Code Partitioning

Moez Ayed and Jean-Luc Gaudiot

Department of EE-Systems
University of Southern California
Los Angeles, CA 90089-2563

email: {*mayed, gaudiot*}*@usc.edu*

November 29, 1996

Abstract

In this paper, we propose a heuristic for code partitioning for Distributed Memory Multiprocessors (DMMs). Our method is data-flow based where all levels of parallelism are exploited. Given a weighted Directed Acyclic Graph (DAG) representation of the program, our partitioning algorithm automatically determines the granularity of parallelism by partitioning the graph into tasks to be scheduled on the DMM. The granularity of parallelism depends only on the program to be executed and on the target machine parameters. The output of our algorithm is passed on as input to the scheduling phase. Finding an optimal solution to this problem is NP-complete. Due to the high cost of graph algorithms, it is nearly impossible to come up with close to optimal solutions that do not have very high cost (higher order polynomial). Therefore, we propose a heuristic that gives good performance and that has relatively low cost.

1 Introduction

Programming DMMs has been a very difficult and complicated problem. High-level programming abstractions for these machines are almost non-existent, leaving the programmers the task of explicitly programming these architectures using machine-dependent, low-level abstractions. This approach is error-prone and forces the programmer to deal with many details outside of the application domain. More precisely, the programmer has to deal with all parallel processing tasks required to program the parallel machine. These tasks include explicit partitioning of the program into parallel tasks, scheduling these tasks on the Processing Elements (PEs), synchronization, and explicit distribution of data among the PEs

and insertion of the appropriate message passing calls needed to exchange data from one remote memory to another.

Much effort is being done to make the parallel processing tasks mentioned above be done automatically by the compiler of the parallel machine. This way, the user does not have to know the details of the architecture of the machine. His/her main concern is the specification of the algorithm for solving the problem. Two of the main phases of the compiler are the code partitioning and scheduling phases. Many solutions have been proposed regarding the scheduling phase. However, very little or almost nothing has been done regarding the code partitioning phase. Most existing work regarding the partitioning problem falls into 2 classes. In the first class, researchers consider a specific application and try to come up with an efficient partitioning scheme for that application (i.e. no automatic partitioning). For an example of that, refer to [6]. In the second class, researchers come up with a general solution (automatic partitioning) that is too simple and therefore not efficient (e.g. exploit only one kind of parallelism level). For examples of such research, refer to [3, 4, 5, 8, 9, 10, 11].

Our research deals with the code partitioning phase of the compiler. We propose a data-flow based partitioning method where all levels of parallelism are exploited. Given a Directed Acyclic Graph (DAG) representation of the program, we propose a procedure that automatically determines the granularity of parallelism by partitioning the graph into tasks to be scheduled on the DMM. The granularity of parallelism depends only on the program to be executed and on the target machine parameters. The output of our algorithm is passed on as input to the scheduling phase. Finding an optimal solution to this problem is NP-complete. Due to the high cost of graph algorithms, it is nearly impossible to come up with close to optimal solutions that do not have very high cost (higher order polynomial). Therefore, we propose a heuristic that gives good performance and that has relatively low cost.

The rest of the paper is organized as follows. In section 2, we define the partitioning problem. In section 3, we talk about the task merging process. In section 4, we do some analysis of the task graph in order to come up with some criteria to choose pairs of tasks to be merged. In section 5, we describe our partitioning heuristic. Section 6 deals with the performance of our partitioning algorithm using some regular DAGs. Finally, concluding remarks and future research are the topic of section 7.

2 Defining the Partitioning Problem

2.1 Assumptions

- We assume that we have a weighted and flat (non-hierarchical) DAG representation of the program. The nodes represent instructions (primitive or compound statements), and the edges represent data movement.
- All inputs of the DAG are assumed to be ready when program starts execution.
- The target DMM is assumed to have point to point communication links (no busses).

- *Convexity Constraint:* A task receives all inputs before starting execution, then it executes to completion without interruption (i.e. no partial task execution).
  For this not to cause any deadlock situations, we have to make sure that the task graph is acyclic at all times.

2.2 Definitions

Partition: Given a multiprocessor M and a program graph g to be executed on M, a partition of g is a set $\Pi = \{\tau_1, \tau_2, \ldots, \tau_n\}$, where each τ_i is a non-empty set consisting of nodes of g that have to be executed on the same PE, and where all the τ_i's are disjoint, and their union forms all the nodes in g. Each set τ_i is called a *task*.

Trivial Partition: It is the partition for which each task τ_i is a singleton set (i.e. this is the partition that puts each node in a single task).

Input and Output Nodes: Given a DAG g, an *input (entry)* node in g is any node for which an input edge carries an input data, and an *output (exit)* node in g is any node for which an output edge carries an output data. There is no predecessor node to an input node and there is no successor node to an output node. Input nodes are also called *root* nodes, and output nodes are also called *leaf nodes.*

Execution Path: Given a DAG g, an execution path of g is any path from an input node to an output node.

Critical Path: Given a DAG g, a critical path of g is any longest execution path in g. The critical path length of g (CPL) is the length of a critical path of g.

Independent Nodes: Given a DAG, two nodes are dependent if and only if there is a path between them. Otherwise they are independent.

Independent and Dependent Sets: Consider a DAG g. An independent set is a set of nodes in g in which each pair of nodes are independent (i.e. all nodes are pairwise independent). A dependent set is a set of nodes in g in which each pair of nodes are dependent (i.e. all nodes are pairwise dependent).

Task Graph: We define the *task dependency graph* (or *task graph* for short) of a partition to be the *directed* graph whose nodes are the tasks in the partition, and where the arcs between nodes represent the data dependency between tasks (i.e. there is an edge from task τ_i to task τ_j *if and only if* data has to be transmitted from τ_i to τ_j).

2.3 Problem Statement

Sarkar [8, 9, 10] defines the partitioning problem as follows:
Find the optimal partition[1] Π_∞, assuming that the DMM satisfies the following 2 properties:

1. Infinite number of PEs.
2. Minimum non-zero communication overhead[2] between the PEs.

Why Π_∞?

Consider a task τ_i in Π_∞. Let $\tau_i = \{a_1, a_2, \ldots, a_n\}$, where the a_j's are actors in the input graph. Since under the ideal case of infinite number of PEs and minimum non-zero communication overhead actors $a_1, a_2, \ldots, a_n$ belong to the same task, then under the realistic case of finite number of PEs and actual communication overhead they have to belong to the same task as well. Hence all the actors that belong to the same task in Π_∞ belong to the same task in the optimal partition for the realistic case. Therefore, by doing some further merging of the tasks in Π_∞, we can obtain the optimal partition for the realistic case. If our scheduling algorithm is optimal, then the tasks that should be merged together will be assigned to the same PE. In our approach, Π_∞ is passed as the input to the scheduling phase, and we rely on the scheduling algorithm to assign the tasks that should be merged together to the same PE.

Overall Procedure

- Start with the *trivial partition.*
- Perform a sequence of partitioning refinements.
- At each step, the algorithm tries to improve on the previous partition by choosing a pair of tasks to be merged *using some heuristic*[3].
  We record PARTIME corresponding to the new partition.
- Stop when the singleton partition is reached.
- Choose partition with lowest PARTIME.

---

[1] The optimal partition is the one that results into minimal parallel execution time PARTIME.

[2] Here we assume that the communication overhead between any 2 processors is minimal. In other words, we assume that the distance between any two processors is one hop (i.e. all processors are directly linked with one another). Also, we assume that the total communication load in the multiprocessor network is always negligible and does not affect the communication time between processors, and therefore we can ignore it.

[3] Merging 2 tasks τ_i and τ_j means replacing them by a new task τ_k which contains all the actors in τ_i and τ_j: $\tau_k = \tau_i \cup \tau_j$.

Equivalent Problem Statement

At each step of the algorithm, we assume that each task of the current partition is mapped to a separate PE. Also, we assume that all inputs of the program graph are ready before the program starts execution. Therefore, we can estimate the Parallel Execution Time (PARTIME) to the CPL of the task graph of the current partition. Hence, the optimal partition is the one for which the corresponding task graph has the shortest CPL among all partitions. The task graph corresponding to the optimal partition is called *optimal task graph.*

A Note on Scheduling

Note that we distinguish between the partitioning and the scheduling problems. We define the scheduling problem to be the assignment of the tasks in the partition obtained from the partitioning phase to the available processors so as to minimize PARTIME. Hence, for each task, we have to decide on *when* and *where* (on which processor) to execute the task. Clearly, for the tasks that are assigned to the same processor, we have to decide on the order of execution of these tasks. The output of our partitioning analysis is therefore the input to the scheduling phase.

A Comparison with DSC

The scheduling problem as defined by Tao Yang [2, 12, 13] (a sequence of task clustering) has some similarities with the way we define the partitioning problem (a sequence of task merging). Mainly both assume the availability of an infinite number of PEs and non-zero communication overhead between PEs. As a consequence, both problems use the CPL of the task graph as the parallel execution time.

However, there is a major difference, since merging tasks involves changes in the task graph[4] whereas task clustering doesn't[5].

Because of this, there is a major difference in the effect of task merging and task clustering on the length of execution paths and as a consequence on the CPL of the task graph.

Task Merging

Task merging could increase the CPL of the task graph. As an example, consider the task graph in figure 1. Before the merger, the critical path is (n_4, n_5) and the CPL is 15. After the merger, both execution paths (n_1, n_3, n_5) and (n_2, n_3, n_5) increase in length by 4. The CPL increases to 18. This is a side effect of the merger.

[4] When two tasks are merged, they are replaced by a new task and some edges are replaced by new ones.

[5] Clustering simply means that all tasks in the same cluster are executed in the same PE. The only change in the task graph is the addition of pseudo-edges between independent tasks in the same cluster to impose an execution order in the PE. Also all weights of edges between tasks in the same cluster are zeroed.

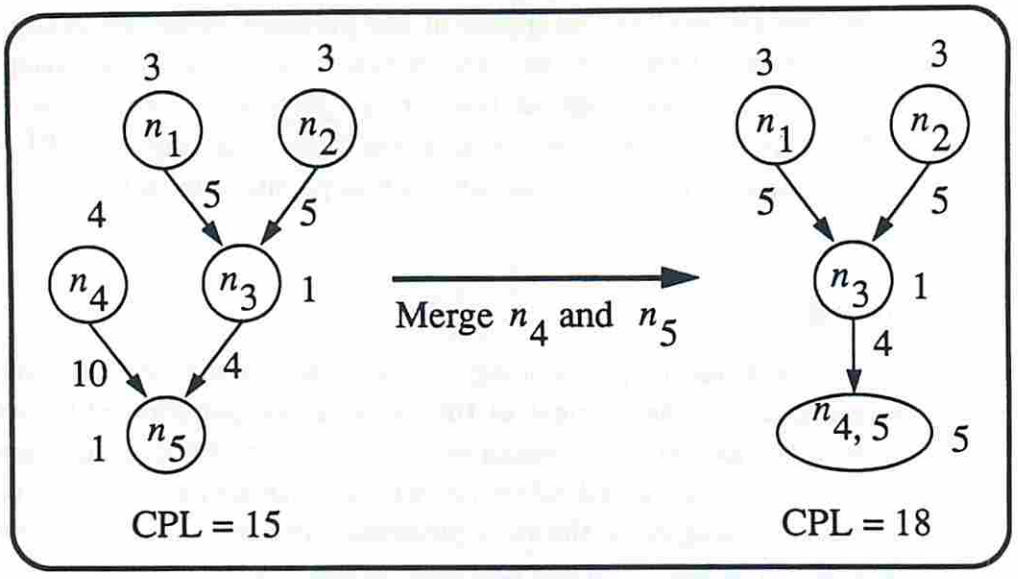

Figure 1: Example of task merging

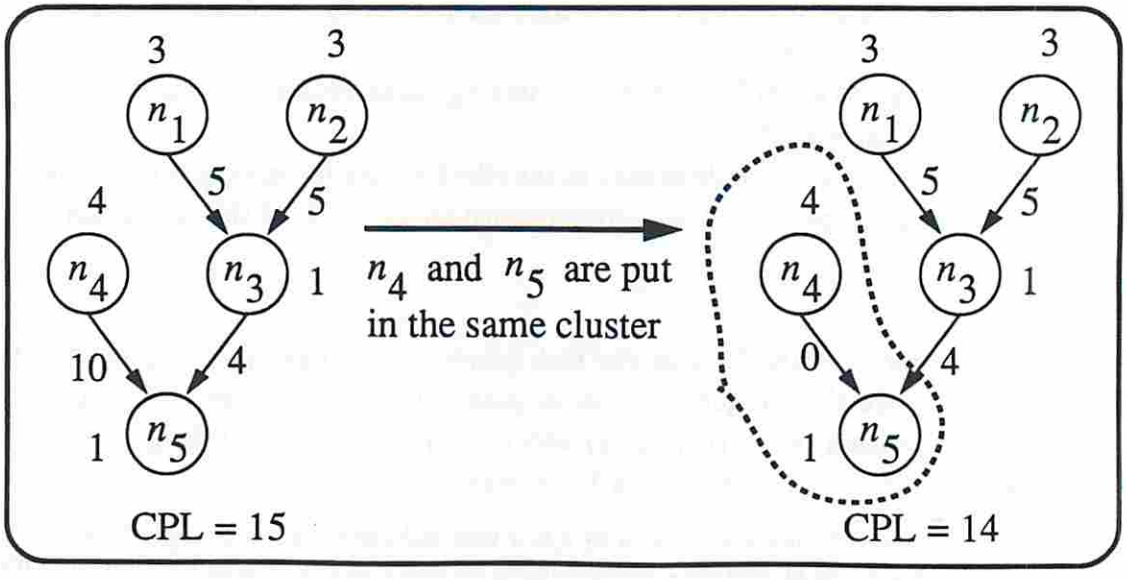

Figure 2: Example of task clustering

Task Clustering

Task clustering rarely causes the CPL of the task graph to increase. As an example, consider the task graph in figure 2. Before the merger, the critical path is (n_4, n_5) and the CPL is 15. After n_4 and n_5 are put in the same cluster, the execution paths (n_1, n_3, n_5) and (n_2, n_3, n_5) are not affected[6]. The CPL decreases to 14.

Consequence

Because of the side effect caused by task merging, reducing the CPL using task merging is much harder than using task clustering. This makes the partitioning problem (as defined in this work) much harder than the scheduling problem (as defined by Tao Yang).

The Algorithm

An informal description of the algorithm for the code partitioning follows:

```
ALGORITHM PartitionGraph
BEGIN
  Partition := Trivial_Partition  /* Start with the trivial partition.
  PARTIME := CPL of current task graph  /* Parallel Execution Time
                                        /* corresponding to current
                                        /* partition.
  best_partition := Partition  /* Best partition found so far.
  best_time := PARTIME  /* Best parallel execution time found so far.
  WHILE |partition| >= 2 DO
    BEGIN  /* Perform a merging iteration.
      Partition := Merge(H)  /* Choose a pair of tasks to be merged
                             /* using Heuristic H, and merge them.
                             /* Partition = partition after merger.
      PARTIME := CPL of new task graph
      IF (PARTIME < best_time)
        THEN
          BEGIN
            best_partition := Partition
            best_time := PARTIME
          END
    END
END
```

At the end of the execution of the algorithm, variable *best-partition* is the partition chosen by the algorithm, and variable *best-time* is its corresponding PARTIME.

[6]The only case when execution paths could increase in length is when pseudo-edges are added.

3 Task Merging

PARTIME := Parallel execution time of the program on the DMM.
PARTIME = $T_c + T_o$, where
T_c := Computation Time Component,
T_o := Overhead Component (communication overhead only, no scheduling overhead).
There is a trade-off between computation component and overhead component. The more parallelism we exploit, the smaller T_c and the larger T_o will be, and vice versa.
In general, when we merge tasks, T_c increases (loss in parallelism and more sequentialization) and T_o decreases (reduction in communication overhead).

A Merging Rule

The idea behind merging tasks is to reduce the communication cost. However as a side effect, we loose in the amount of parallelism available in the task graph. If the 2 tasks merged are independent, then there is no reduction in communication overhead and there is loss in parallelism. Therefore, there is no gain and possible loss in doing so. If on the other hand the two tasks merged are connected by an edge, then there is reduction in communication overhead and possible loss in parallelism. Thus, there is possible gain in doing so. Hence, the rule is to *merge only pairs of tasks connected by an edge.*

3.1 Merging Procedure

Initially, each actor in the input graph is put in a separate task. Therefore, the task graph has one node for each actor in the input graph. The edges in the task graph are determined by the edges between the actors in the input DAG.

Each time 2 nodes n_1 and n_2 in the task graph connected by an edge (n_1, n_2), are merged into a node $n_{1,2}$, we do the following:

- Nodes n_1 and n_2 are replaced by node $n_{1,2}$.
- Edge (n_1, n_2) is deleted.
- Any edge going from n_1 to any other node n_i $(i \neq 2)$ is replaced by an edge going from $n_{1,2}$ to n_i, and any edge going from n_i to n_1 is replaced by an edge going from n_i to $n_{1,2}$.
- Any edge going from n_2 to any other node n_i $(i \neq 1)$ is replaced by an edge going from $n_{1,2}$ to n_i, and any edge going from n_i to n_2 is replaced by an edge going from n_i to $n_{1,2}$.
- If there is an edge from n_1 to n_i and an edge from n_2 to n_i $(i \neq 1$ and $i \neq 2)$, then edges (n_1, n_i) and (n_2, n_i) are replaced by **one** edge $(n_{1,2}, n_i)$.

- If there is an edge from n_i to n_1 and an edge from n_i to n_2 ($i \neq 1$ and $i \neq 2$), then edges (n_i, n_1) and (n_i, n_2) are replaced by **one** edge $(n_i, n_{1,2})$.

Merging an Edge in the Task Graph

Let $e = (n_1, n_2)$ be an edge in the task graph. Merging the edge e means merging tasks n_1 and n_2 together.

3.2 Notations and Definitions

Given a task graph g. Let n be a node in g. Let $e = (n_1, n_2)$ be an edge in g connecting 2 nodes n_1 and n_2. Let p be a path in g.
comp(n) := Cost of computation in node n.
data(e) := Amount of data communicated along edge e during 1 execution of the program.
data(n_i, n_j) := 0, if there is no edge (n_i, n_j).
comm(e) := Cost of communicating data on edge e during 1 execution of the program.
comp(e) := comp(n_1) + comp(n_2).
L(p) := Length of path p, L(p) = $\sum_{n \in p} comp(n) + \sum_{e \in p} comm(e)$.
$L_b(p)$:= Length of path p before the merger, $L_a(p)$:= Length of path p after the merger.
CPL_b and CPL_a are defined to be the CPL of the task graph before and after the merger respectively.

Consider two PEs PE_1 and PE_2 belonging to the DMM. Let f_c be the cost to communicate a message from PE_1 to PE_2. f_c is a function of the message size s only, because of the characteristics of our DMM.
$f_c(s)$:= Cost to communicate a message of size s from PE_1 to PE_2. $f_c(s)$ has 2 components: a message start-up component T_{start} and a delay component $delay(s)$. $f_c(s) = T_{start} + delay(s)$. $delay(s)$ is proportional to s. Since the message start-up component is a constant and does not depend on the message size, then $f_c(s_1 + s_2) \neq f_c(s_1) + f_c(s_2)$, i.e. $f_c(s)$ is not proportional to s.
$f_c(s_1 + s_2) = T_{start} + delay(s_1 + s_2) = T_{start} + delay(s_1) + delay(s_2)$.
Hence, $f_c(s_1 + s_2) = f_c(s_1) + delay(s_2)$.
We define $delay(s) := 0$, if $s = 0$.
$comm(e) = T_{start} + delay(data(e))$.

3.3 Updating Task Graph Weights as a Result of the Merger

Consider 2 nodes n_1 and n_2 in the task graph connected by an edge (n_1, n_2) and merged into a node $n_{1,2}$.

3.3.1 Node Weights

$comp(n_{1,2}) = comp(n_1) + comp(n_2)$.
Here we assume that all PEs are simple and are not capable of doing parallel computations[7].

3.3.2 Edge Weights

1. If an edge e' replaces **one** edge e:
   $data(e') = data(e)$.

2. If an edge e' replaces **two** edges e_1 and e_2:
   $data(e') = data(e_1) + data(e_2)$.
   Since all messages inside a task that have the same destination task are combined together into a bigger message then:
   $comm(e') \neq comm(e_1) + comm(e_2)$.
   $comm(e') = T_{start} + delay(data(e_1)) + delay(data(e_2))$.
   $comm(e') = comm(e_1) + delay(data(e_2))$.
   $comm(e') = comm(e_2) + delay(data(e_1))$.

3.4 Creation of Cycles as a Result of Task Merging

Since We assume that our execution model obeys the *Convexity Constraint*, the task graph should be acyclic at all times, so that we guarantee that no deadlock situation occurs. Initially the task graph is acyclic since the program graph is acyclic. When we merge tasks, we have to make sure that no cycles are created as a result of the merger.

Theorem: Given an acyclic task graph, consider 2 nodes n_1 and n_2 in the graph, connected by an edge (n_1, n_2) and merged into a node $n_{1,2}$. A cycle is created after the merger *if and only if* there exists a path from n_1 to n_2 other than (n_1, n_2) before the merger.

Proof

1. There exists a path from n_1 to n_2 other than (n_1, n_2) before the merger
   $\Longrightarrow$
   A cycle is created after the merger:
   The proof of this is quite obvious. Refer to figure 3.

2. A cycle is created after the merger
   $\Longrightarrow$
   There exists a path from n_1 to n_2 other than (n_1, n_2) before the merger:
   For a cycle to be introduced, a newly created edge has to have created it. Since any new edge is connected to $n_{1,2}$, then any created cycle contains the node $n_{1,2}$ (see figure

[7] Even simple computations are done sequentially.

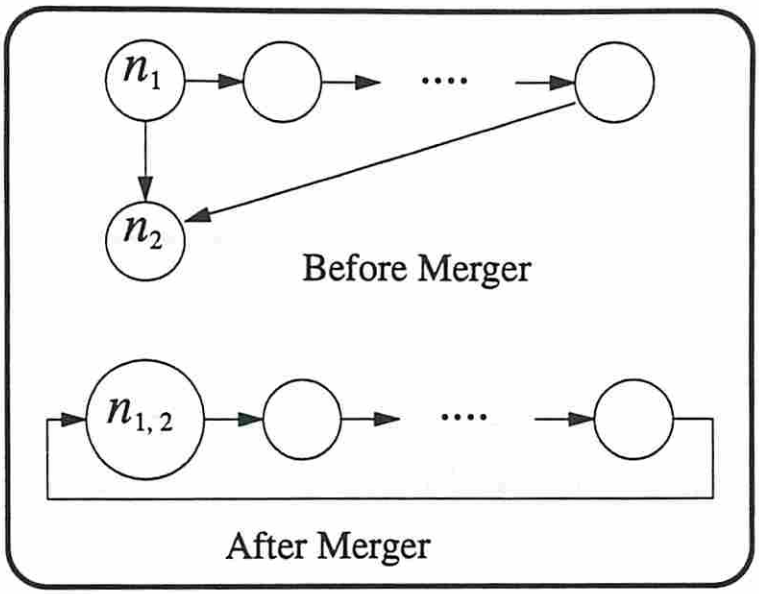

Figure 3: Cycle creation

4-a).
Before the merger, the portion of the graph in figure 4-a used to be the one shown in figure 4-b.
We cannot have edges going from both nodes n_1 and n_2 to n_a, because that would create a cycle, and we no that the graph is acyclic before the merger.
Also we cannot have edges going from n_b to both nodes n_1 and n_2, because that would also create a cycle.
Assume that we have an edge from n_1 to n_a. Then we cannot have an edge from n_b to n_1 because that would create a cycle. Therefore, we can only have an edge from n_b to n_2 (see figure 4-c).
Assume that we have an edge from n_2 to n_a. Then having an edge from n_b to n_1 or an edge from n_b to n_2 would create a cycle. Hence we cannot have an edge from n_2 to n_a.
In conclusion, the only possibility is the one shown in figure 4-c.

Another Merging Rule

If there is a path from n_1 to n_2 in the task graph other than (n_1, n_2), then nodes n_1 and n_2 should not be merged, otherwise we create a cycle in the graph. Keeping the task graph free of cycles guarantees that no deadlock situation occurs because of the convexity constraint.

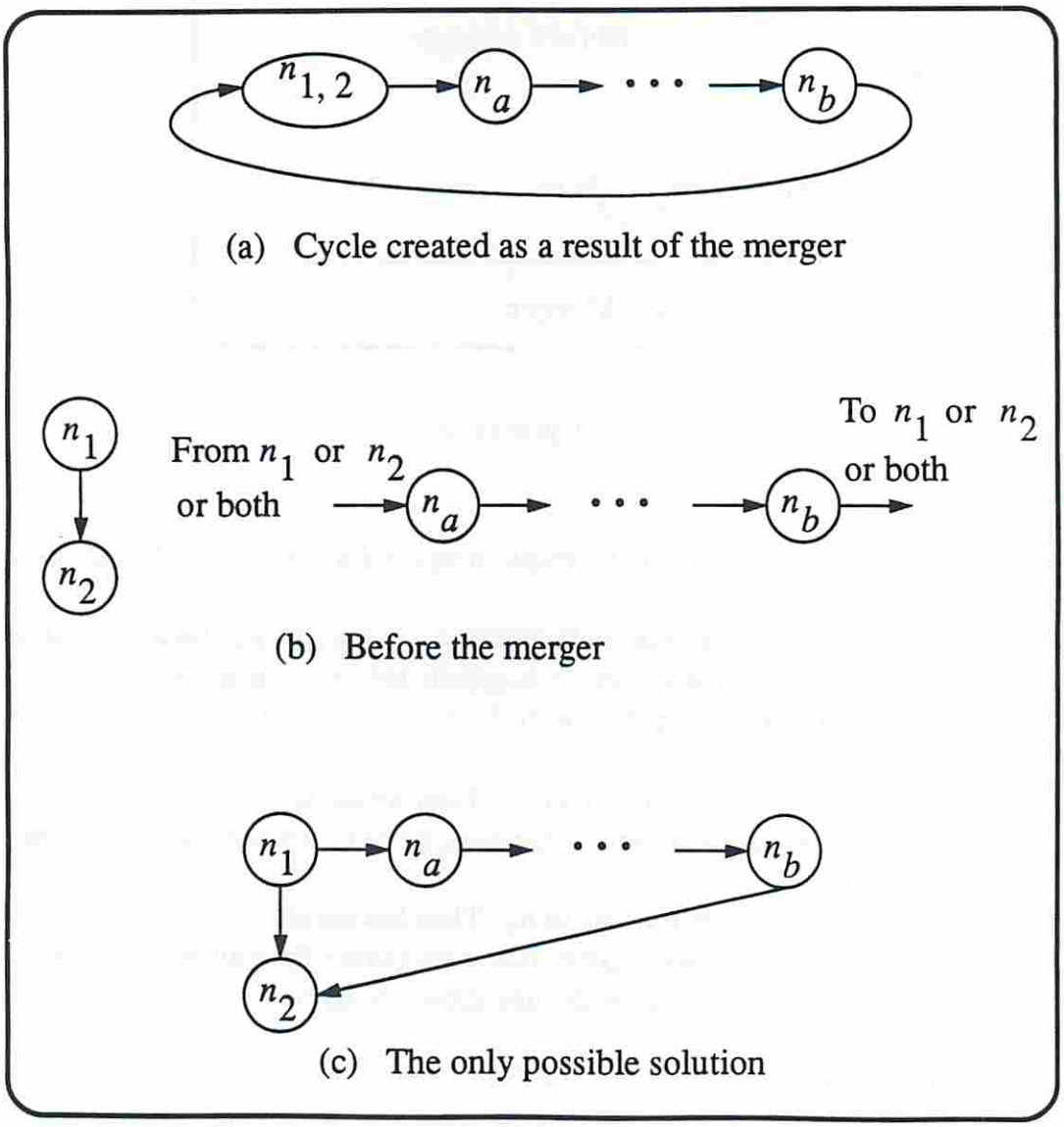

Figure 4: Cycle creation

4 Some Analysis

In this section, we do some analysis of the task graph in order to determine some criteria used by our heuristic for choosing the edge to be merged.

4.1 Parallelism Loss Due to Task Merging

In this section, we study the effect of task merging on the available parallelism in the task graph.

Clearly, given a task graph, there could be loss in parallelism when two nodes connected by an edge are merged.

4.1.1 Definitions

Parallel Set of a Node: Given a node n in a DAG g, the Parallel Set of n is ParSet(n) := $\{n' \in g \ / \ n$ and n' are independent$\}$. These are the nodes that can be executed in parallel[8] with n.

Dependent Set of a Node: Given a node n in a DAG g, the dependent set of n is DepSet(n) := $\{n' \in g \ / \ n$ and n' are dependent$\}$.

4.1.2 Defining Parallelism

Parallelism with respect to a Node: Given a node n in a DAG g, we define the Parallelism with respect to n to be $|ParSet(n)|$[9].

Parallelism Loss with respect to a Node: Given a node n in a DAG g, we say that there is parallelism loss with respect to n as a result of merging nodes *if and only if* the parallelism with respect to n after the merger is strictly smaller than the parallelism with respect to n before the merger.

Condition for Parallelism Loss

Consider two nodes n_1 and n_2 in the task graph, connected by an edge (n_1, n_2) and merged into a node $n_{1,2}$. We say that some parallelism is lost in the task graph as a result of the merger *if and only if* some parallelism is lost with respect to n_1 or n_2. Thus if we guarantee that no parallelism is lost with respect to n_1 **and** no parallelism is lost with respect to n_2, then we guarantee that no parallelism is lost in the task graph as a result of the merger.

$ParSet(n_1)$ represents all nodes which can be executed in parallel with n_1 before the merger. Any node belonging to $DepSet(n_2)$ cannot be executed in parallel with n_1 after

[8] n may not be executed in parallel with all nodes in ParSet(n) simultaneously, since ParSet(n) is not necessarily an independent set.

[9] Given a set S, $|S|$ is the number of elements in S.

the merger. Therefore, the set ParSet(n_1) $\cap$ DepSet(n_2) represents all nodes which could execute in parallel with n_1 before the merger, and no longer can be executed in parallel with n_1 after the merger.

Hence:

1. Some parallelism will be lost with respect to n_1 as a result of the merger *if and only if* ParSet(n_1) $\cap$ DepSet(n_2) $\neq \emptyset$.

2. Some parallelism will be lost with respect to n_2 as a result of the merger *if and only if* ParSet(n_2) $\cap$ DepSet(n_1) $\neq \emptyset$.

The same analysis can be applied to node n_2.

Amount of Parallelism Lost

Consider 2 nodes n_1 and n_2 in the task graph, connected by an edge (n_1, n_2) and merged into a node $n_{1,2}$.

1. The amount of parallelism lost (if any) with respect to n_1 as a result of the merger is $|ParSet(n_1) \cap DepSet(n_2)|$.

2. The amount of parallelism lost (if any) with respect to n_2 as a result of the merger is $|ParSet(n_2) \cap DepSet(n_1)|$.

The amount of parallelism lost in the task graph as a result of the merger is defined to be the sum of the amount of parallelism lost with respect to n_1 and the amount of parallelism lost with respect to n_2.

Theorem

Let g be a task graph. Let n_1 and n_2 be 2 nodes in g connected by an edge $e = (n_1, n_2)$.
Let $S_1 := DepSet(n_1) - \{n_2\}$.
Let $S_2 := DepSet(n_2) - \{n_1\}$.
$ParSet(n_1) \cap DepSet(n_2) = \emptyset$ AND $ParSet(n_2) \cap DepSet(n_1) = \emptyset$
$\Longleftrightarrow$
$ParSet(n_1) = ParSet(n_2)$
$\Longleftrightarrow$
$S_1 = S_2$.

Proof

1. Assume that: $ParSet(n_1) \cap DepSet(n_2) = \emptyset$ AND $ParSet(n_2) \cap DepSet(n_1) = \emptyset$.

(a) $ParSet(n_1) \cap DepSet(n_2) = \emptyset$:
$DepSet(n_2) = \{n_1\} \cup S_2$.
$\forall n \in S_2, n \notin ParSet(n_1) \Rightarrow$
$\forall n \in S_2, n \in DepSet(n_1)$ (since $n \neq n_1$) $\Rightarrow$
$\forall n \in S_2, n \in S_1$ (since $n \neq n_2$) $\Rightarrow$
$S_2 \subset S_1$ (1)
$\forall n \in ParSet(n_1), n \notin DepSet(n_2) \Rightarrow$
$\forall n \in ParSet(n_1), n \in ParSet(n_2)$ (since $n \neq n_2$) $\Rightarrow$
$ParSet(n_1) \subset ParSet(n_2)$ (2)

(b) $ParSet(n_2) \cap DepSet(n_1) = \emptyset$:
$DepSet(n_1) = \{n_2\} \cup S_1$.
$\forall n \in S_1, n \notin ParSet(n_2) \Rightarrow$
$\forall n \in S_1, n \in DepSet(n_2)$ (since $n \neq n_2$) $\Rightarrow$
$\forall n \in S_1, n \in S_2$ (since $n \neq n_1$) $\Rightarrow$
$S_1 \subset S_2$ (3)
$\forall n \in ParSet(n_2), n \notin DepSet(n_1) \Rightarrow$
$\forall n \in ParSet(n_2), n \in ParSet(n_1)$ (since $n \neq n_1$) $\Rightarrow$
$ParSet(n_2) \subset ParSet(n_1)$ (4)

(1) AND (3) $\Rightarrow S_1 = S_2$.
(2) AND (4) $\Rightarrow ParSet(n_1) = ParSet(n_2)$.

2. Assume that:
$ParSet(n_1) = ParSet(n_2)$.
$\forall n \in DepSet(n_2), n \notin ParSet(n_2) \Rightarrow$
$\forall n \in DepSet(n_2), n \notin ParSet(n_1) \Rightarrow$
$ParSet(n_1) \cap DepSet(n_2) = \emptyset$ (5)
$\forall n \in DepSet(n_1), n \notin ParSet(n_1) \Rightarrow$
$\forall n \in DepSet(n_1), n \notin ParSet(n_2) \Rightarrow$
$ParSet(n_2) \cap DepSet(n_1) = \emptyset$ (6)
(5) AND (6) $\Rightarrow S_1 = S_2$.

3. Assume that:
$S_1 = S_2$.
$\forall n \in ParSet(n_1), n \notin DepSet(n_1) \Rightarrow$
$\forall n \in ParSet(n_1), n \notin S_1 \Rightarrow$
$\forall n \in ParSet(n_1), n \notin S_2 \Rightarrow$
$\forall n \in ParSet(n_1), n \notin DepSet(n_2)$ (since $n \neq n_1$) $\Rightarrow$
$\forall n \in ParSet(n_1), n \in ParSet(n_2)$ (since $n \neq n_2$) $\Rightarrow$
$ParSet(n_1) \subset ParSet(n_2)$ (7)
$\forall n \in ParSet(n_2), n \notin DepSet(n_2) \Rightarrow$
$\forall n \in ParSet(n_2), n \notin S_2 \Rightarrow$

$\forall n \in ParSet(n_2), n \notin S_1 \Rightarrow$
$\forall n \in ParSet(n_2), n \notin DepSet(n_1)$ (since $n \neq n_2$) $\Rightarrow$
$\forall n \in ParSet(n_2), n \in ParSet(n_1)$ (since $n \neq n_1$) $\Rightarrow$
$ParSet(n_2) \subset ParSet(n_1)$ (8)
(7) AND (8) $\Rightarrow ParSet(n_1) = ParSet(n_2)$.

Corollary

Let g be a task graph. Let n_1 and n_2 be 2 nodes in g connected by an edge $e = (n_1, n_2)$.
No parallelism is lost in the task graph as a result of the merger
$\Longleftrightarrow$
$ParSet(n_1) = ParSet(n_2)$.

Proof

No parallelism is lost as a result of the merger *if and only if* no parallelism is lost with respect to n_1 and no parallelism is lost with respect to n_2.
This is true *if and only if*
$ParSet(n_1) \cap DepSet(n_2) = \emptyset$ AND $ParSet(n_2) \cap DepSet(n_1) = \emptyset$.
From the above theorem, that is true *if and only if*
$ParSet(n_1) = ParSet(n_2)$.

4.2 Effect of Task Merging on CPL

4.2.1 Problem Statement

In all what follows, we assume that 2 nodes n_1 and n_2 in the task graph connected by an edge $e = (n_1, n_2)$, are merged into a node $n_{1,2}$.
Let $p_c = P_{crit}$ of task graph before the merger.
$l_b :=$ length of P_{crit} of task graph before the merger.
$l_b = L_b(p_c)$.
$CPL_b := l_b$.
$l_a :=$ length of P_{crit} of task graph after the merger.

4.2.2 Effect on Path Length

Let p be any path in the task graph.
We have 3 possibilities:

1. **None** of the two nodes merged belongs to p.
2. **Only one** of the two nodes merged belongs to p.
3. **Both** nodes merged belong to p: $\Rightarrow e \in p$.
   To see why this is true, assume that $e \notin p$.
   $\Rightarrow$ there are two possibilities:

(a) There is a path from n_1 to n_2 other than (n_1, n_2).
$\Rightarrow$ After a merger, a cycle will be created.
Therefore n_1 and n_2 cannot be merged together.

(b) There is a path from n_2 to n_1.
$\Rightarrow$ There is a cycle before the merger, because of edge (n_1, n_2), which is not possible since we have a DAG.

The length of p is affected by the merger *if and only if* at least one of the following conditions is true:

1. A node in p is replaced by another node that has more computations (this is the case when only one of the 2 merged nodes belongs to p).
   Assuming that $n_1 \in p$,
   $L(p)$ is increased by $comp(n_2)$.

2. An edge in p is deleted (this is the case when $e \in p$).
   $L(p)$ is reduced by $comm(e)$.

3. An edge in p is replaced by another edge which carries more data (this is the case when **two** edges e_1 and e_2 are replaced by **one** edge e', and either e_1 or e_2 belongs to p).
   Assume $e_1 \in p$, then
   $L(p)$ is increased by $delay(data(e_2))$.

There are 3 cases:

Case 1 None of the two nodes merged belongs to p:
$L_a(p) = L_b(p)$.

Case 2 Only one of the two nodes merged (say it is n_1) belongs to p:
Let n_p be the predecessor of n_1 in p (if any).
Let n_s be the successor of n_1 in p (if any).
$L_a(p) = L_b(p) + comp(n_2) + delay(data(n_p, n_2)) + delay(data(n_2, n_s))$.
Note that if (n_p, n_2) and (n_2, n_s) don't exist (or if n_p and n_s don't exist), then
$L_a(p) = L_b(p) + comp(n_2)$.
This increase in length of p represents a loss in parallelism and increase in sequentialization by the amount $comp(n_2) + delay(data(n_p, n_2)) + delay(data(n_2, n_s))$ relative to path p.
The terms involving the *delay* function are due to the fact that some inter-PE communication has to be sequentialized are a result of the merger. For instance, the increase by the amount $delay(data(n_p, n_2))$ is due to the fact that before the merger, n_p used to send the data on edges (n_p, n_1) and (n_p, n_2) to separate virtual PEs in parallel. After the merger, the data on these 2 edges is combined and sent to the same virtual PE. Clearly this takes more time.

In conclusion, we could have an increase in the CPL, and as a consequence the parallel execution time could increase.

Case **3 Both** nodes merged belong to p:
Let n_p be the predecessor of n_1 in p (if any).
Let n_s be the successor of n_2 in p (if any).
$L_a(p) = L_b(p) - comm(e) + delay(data(n_p, n_2)) + delay(data(n_1, n_s))$.
Note that if (n_p, n_2) and (n_1, n_s) don't exist (or if n_p and n_s don't exist), then
$L_a(p) = L_b(p) - comm(e)$.
This decrease in the length of p represents a reduction in communication overhead by the amount $x = comm(e) - delay(data(n_p, n_2)) - delay(data(n_1, n_s))$ relative to path p (assuming that $x > 0$, which is true for most cases).
Again, the terms involving the *delay* function are due to the fact that some inter-PE communication has to be sequentialized are a result of the merger.

4.2.3 Merging an Edge Belonging to the Critical Path

In what follows, we omit the terms involving the function *delay* in the expressions giving the length of a path after merging two nodes, in terms of its length before the merger.

Let's assume that $e \in P_{crit}$ of task graph.
Thus, $L_a(p_c) = l_b - comm(e)$.

Effect on Execution Paths

Clearly for any execution path p in the task graph, $L_b(p) \leq l_b$, since l_b is the CPL before the merger.

1. Any execution path p that doesn't go through any of the 2 nodes merged:
   $L_a(p) = L_b(p)$.
2. Any execution path p that goes through n_1 and not n_2:
   $L_a(p) = L_b(p) + comp(n_2)$.
3. Any execution path p that goes through n_2 and not n_1:
   $L_a(p) = L_b(p) + comp(n_1)$.
4. Any execution path p that goes through edge e:
   $L_a(p) = L_b(p) - comm(e)$.

Effect on Critical Path

Case **1 There is no execution path that goes through only one of the 2 nodes merged:**
Thus for any execution path p, $L_a(p) \leq L_b(p)$.

Also we know that $L_b(p) \leq l_b$. Therefore, $L_a(p) \leq l_b$.
Hence, *CPL will either decrease or remain unchanged as a result of the merger.*
P_{crit} *could change as a result of the merger.* We have 2 cases:

1. If all execution paths p go through e:
   $L_a(p) = L_b(p) - comm(e)$.
   In this case, all execution paths including p_c will be reduced in length by the same amount.
   Hence, P_{crit} will not change and CPL decreases by $comm(e)$ as a result of the merger.
2. At least one execution path doesn't go through e:
   Let $p_1, p_2, \ldots, p_k$ be such execution paths, where $k \geq 1$.
   $L_a(p_i) = L_b(p_i)$, $1 \leq i \leq k$.
   There are 2 cases:
   (a) If at least one of the p_i's is such that
   $L_b(p_i) = l_b$:
   P_{crit} will change and CPL will remain unchanged.
   (b) If $L_b(p_i) < l_b$, $1 \leq i \leq k$:
   CPL will decrease.
   P_{crit} could change. There are 2 possibilities:
   i. $L_b(p_i) \leq L_a(p_c)$, $1 \leq i \leq k$:
   CPL will decrease by $comm(e)$.
   P_{crit} will not change.
   ii. At least one of the p_i's is such that $L_b(p_i) > L_a(p_c)$:
   P_{crit} will change.
   CPL will decrease by an amount smaller than $comm(e)$.
   Let p_m be the p_i such that $L_b(p_m)$ is the largest among all p_i's.
   After the merger,
   $P_{crit} = p_m$ and $CPL = L_b(p_m)$.
   CPL will decrease by $l_b - L_b(p_m)$.

Case 2 There is at least one execution path p that goes through only one of the 2 nodes merged:
P_{crit} *could change as a result of the merger, and CPL could increase*, since the length of p increases after the merger.
Let $p_1, p_2, \ldots, p_k$ be all execution paths that go through only one of the 2 nodes merged, where $k \geq 1$.
Let $n_{i,1}$ and $n_{i,2}$ be the two nodes merged, and let $n_{i,1}$ be the node that belongs to p_i, and let $n_{i,2}$ be the other node[10], $1 \leq i \leq k$.
$L_a(p_i) = L_b(p_i) + comp(n_{i,2})$, $1 \leq i \leq k$.
We know that $l_b \geq L_b(p_i)$, $1 \leq i \leq k$, since l_b is the CPL before the merger.

[10] If p_i goes through n_1 then $n_{i,1}$ is n_1 and $n_{i,2}$ is n_2. If p_i goes through n_2 then $n_{i,1}$ is n_2 and $n_{i,2}$ is n_1.

There are so many possibilities, depending on the length of the execution paths before the merger, l_b, the value of $comp(n_{i,2})$, the value of $comm(e)$, etc.

Since we already studied the case where no execution path goes through only one of the 2 nodes merged, **let's assume that all execution paths (p_c excluded) go through only one of the 2 nodes merged**. This will simplify our analysis.
In this case, the execution paths are $p_1, p_2, \ldots, p_k$ and p_c.
There are 2 possible situations:

1. If $L_a(p_i) \leq L_a(p_c)$, $1 \leq i \leq k$:
   P_{crit} will not change.
   CPL will decrease by $comm(e)$.
2. If there is at least one execution path p such that $L_a(p) > L_a(p_c)$:
   P_{crit} will change, but CPL does not necessarily increase. We have 2 cases:
   (a) If $L_a(p_i) \leq l_b$, $1 \leq i \leq k$:
      i. If at least one of the p_i's is such that $L_a(p_i) = l_b$:
         Let p_o be this p_i.
         After the merger,
         $P_{crit} = p_o$.
         CPL remains unchanged.
      ii. If $L_a(p_i) < l_b$, $1 \leq i \leq k$:
         CPL will decrease by an amount less than $comm(e)$.
         Let p_m be the p_i such that $L_a(p_m)$ is the largest among all p_i's.
         After the merger,
         $P_{crit} = p_m$ and $CPL = L_a(p_m)$.
         CPL will decrease by $l_b - L_b(p_m) - comp(n_{m,2})$.
   (b) If there is at least one execution path p such that $L_a(p) > l_b$:
      CPL will increase.
      Let p_m be the p_i such that $L_a(p_m)$ is the largest among all p_i's.
      After the merger,
      $P_{crit} = p_m$ and $CPL = L_a(p_m)$.
      CPL will increase by $L_b(p_m) + comp(n_{m,2}) - l_b$.

Conclusion

- Merging an edge that belongs to P_{crit} of task graph does not guarantee a decrease in CPL.
- The maximum decrease in CPL is $comm(e)$.
- Merging an edge that belongs to all execution paths guarantees the maximum decrease in CPL.

- If none of the execution paths go through only one of the 2 nodes merged, then CPL will either decrease or remain unchanged.
  Also the maximum decrease in CPL could be achieved here.

- If at least one execution path goes through only one of the 2 nodes merged, then CPL will either increase, remain unchanged or decrease.
  Also the maximum decrease in CPL could be achieved here.

4.2.4 Merging an Edge Not Belonging to the Critical Path

In what follows, we omit the terms involving the function *delay* in the expressions giving the length of a path after merging two nodes, in terms of its length before the merger.

Let's assume that $e \notin P_{crit}$.
Clearly, e belongs to at least one execution path[11].
There are 2 possible cases:

Case 1 P_{crit} goes through only 1 of the 2 nodes merged:
Let n_{in} be the node merged which belongs to p_c, and let n_{out} be the node merged which does not belong to p_c.
$L_a(p_c) = l_b + comp(n_{out})$.
After the merger, CPL will increase by at least $comp(n_{out})$ and P_{crit} *might change.*
There are 2 possibilities:

1. If none of the execution paths p (p_c excluded) go through only one of the 2 nodes merged:
   $L_a(p) \leq L_b(p)$.
   Since $L_b(p) \leq l_b$, then $L_a(p) \leq l_b$.
   Hence CPL will increase by $comp(n_{out})$ and P_{crit} will not change after the merger.
2. If at least one execution path (p_c excluded) goes through only one of the 2 nodes merged:
   Let $p_1, p_2, \ldots, p_k$ be all the execution paths that go through only one of the 2 nodes merged (p_c excluded), $k \geq 1$.
   Let $n_{i,1}$ be the node merged which belongs to p_i, and $n_{i,2}$ be the node merged which does not belong to p_i.
   $L_a(p_i) = L_b(p_i) + comp(n_{i,2})$, $1 \leq i \leq k$.
   If $comp(n_{i,2}) \leq comp(n_{out})$ then P_{crit} will not change and CPL will increase by $comp(n_{out})$.
   If $n_{i,2} = n_{out}$ then P_{crit} will not change and CPL will increase by $comp(n_{out})$.
   Let p_m be the p_i such that $L_a(p_m)$ is the largest among all p_i's.

[11] Any edge in the graph belongs to at least one execution path.

There are 2 possible cases:

(a) If $L_a(p_i) \leq L_a(p_c)$, $1 \leq i \leq k$:
P_{crit} will not change and CPL will increase by $comp(n_{out})$.

(b) If at least one p_i is such that $L_a(p_i) > L_a(p_c)$:
After the merger,
$P_{crit} = p_m$ and $CPL = L_a(p_m)$.
CPL will increase by an amount greater than $comp(n_{out})$.
The increase in CPL is $L_b(p_m) + comp(n_{m,2}) - l_b$.

Case 2 P_{crit} does not go through any of the 2 nodes merged:
$L_a(p_c) = L_b(p_c) = l_b$.
CPL will either increase or remain unchanged.
P_{crit} *might change.*
There are 2 possibilities:

1. If no execution path p (p_c excluded) goes through only one of the 2 nodes merged:
$L_a(p) \leq L_b(p)$.
Since $L_b(p) \leq l_b$ then $L_a(p) \leq l_b$.
Thus, P_{crit} and CPL will not change.

2. If at least one execution path goes through only one of the 2 nodes merged:
Let $p_1, p_2, \ldots, p_k$ be all the execution paths that go through only one of the 2 nodes merged (p_c excluded), $k \geq 1$.
Let $n_{i,1}$ be the node merged which belongs to p_i, and $n_{i,2}$ be the node merged which does not belong to p_i.
$L_a(p_i) = L_b(p_i) + comp(n_{i,2})$, $1 \leq i \leq k$.
Hence, P_{crit} could change and CPL could increase.
There are 2 cases:

   (a) If $L_a(p_i) \leq l_b$, $1 \leq i \leq k$:
   P_{crit} and CPL will not change.

   (b) If at least one p_i is such that $L_a(p_i) > l_b$:
   Let p_m be the p_i such that $L_a(p_m)$ is the largest among all p_i's.
   After the merger,
   $P_{crit} = p_m$ and $CPL = L_a(p_m)$.
   CPL will increase by $L_b(p_m) + comp(n_{m,2}) - l_b$.

Conclusion

- If $e \notin P_{crit}$ then CPL never decreases (it will either increase or remain unchanged) after the merger.

- If P_{crit} goes through only one of the 2 nodes merged, then CPL will increase by at least

$comp(n_{out})$ after the merger, where n_{out} is the node merged which does not belong to P_{crit}.

- If P_{crit} does not go through any of the 2 nodes merged, then CPL will either increase or remain unchanged after the merger.

4.3 Merging Tasks: Effect of Parallelism Loss on CPL

In this section, we study the effect of parallelism loss on the CPL of the task graph. We consider two nodes n_1 and n_2 belonging to the task graph and connected by an edge $e = (n_1, n_2)$. We study the effect of merging nodes n_1 and n_2 on the CPL when the merger causes parallelism loss and when it doesn't.

4.3.1 No Parallelism Loss

Theorem: Assume that $ParSet(n_1) = ParSet(n_2)$, so that there is no parallelism loss when we merge n_1 and n_2.
Then the CPL of the task graph never increases as a result of the merger.

Proof

Let's use proof by contradiction. We assume that the CPL increases as a result of the merger. Therefore, there should exist at least one execution path p such that $L_a(p) > CPL_b$. Clearly, $L_b(p) \leq CPL_b$.

There are 3 possible cases:

1. p goes through both nodes n_1 and n_2.
2. p goes through n_1 but not n_2.
3. p goes through n_2 but not n_1.

The situation where p doesn't go through any of the 2 nodes n_1 and n_2 is not possible, since in that case $L(p)$ is not affected by the merger.

Let's investigate the 3 possible cases.

Case1 p goes through both nodes n_1 and n_2:
Let $p = (n_i, \ldots, n_p, n_1, n_2, n_s, \ldots, n_f)$.
For p to have the maximum increase in length after the merger, we have to have an edge (n_p, n_2) and an edge (n_1, n_s) (see figure 5).
Let $\Delta L := L_a(p) - L_b(p)$.
$\Delta L = delay(data(n_p, n_2)) + delay(data(n_1, n_s)) - comm(n_1, n_2)$.
The *comm* function includes both the start-up component and the delay component.

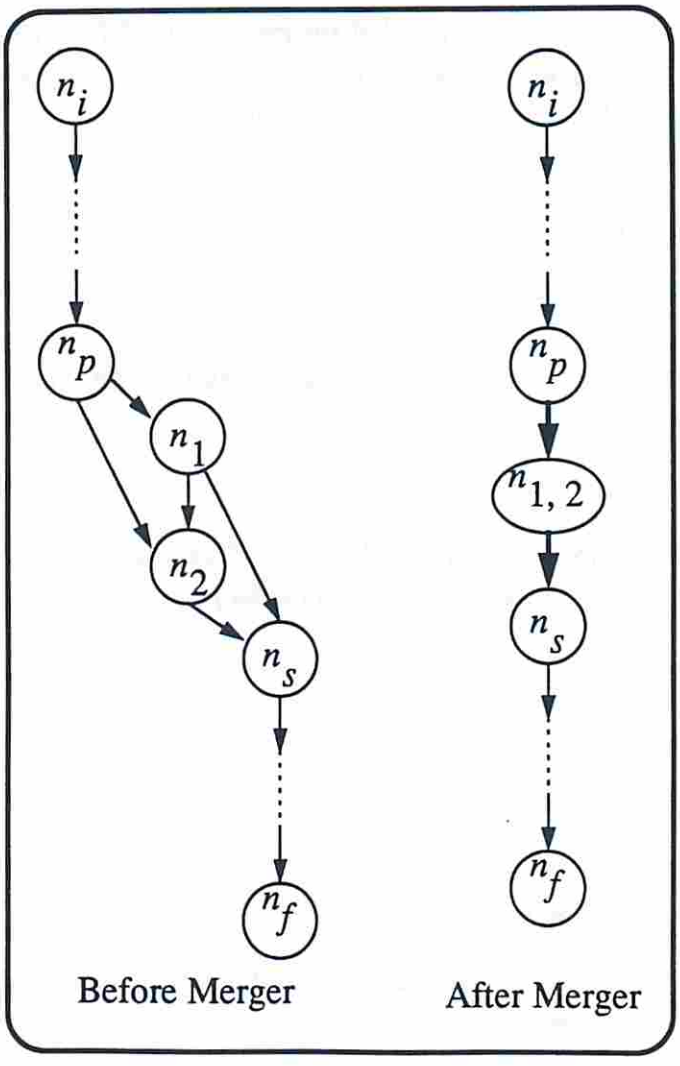

Figure 5: No parallelism loss

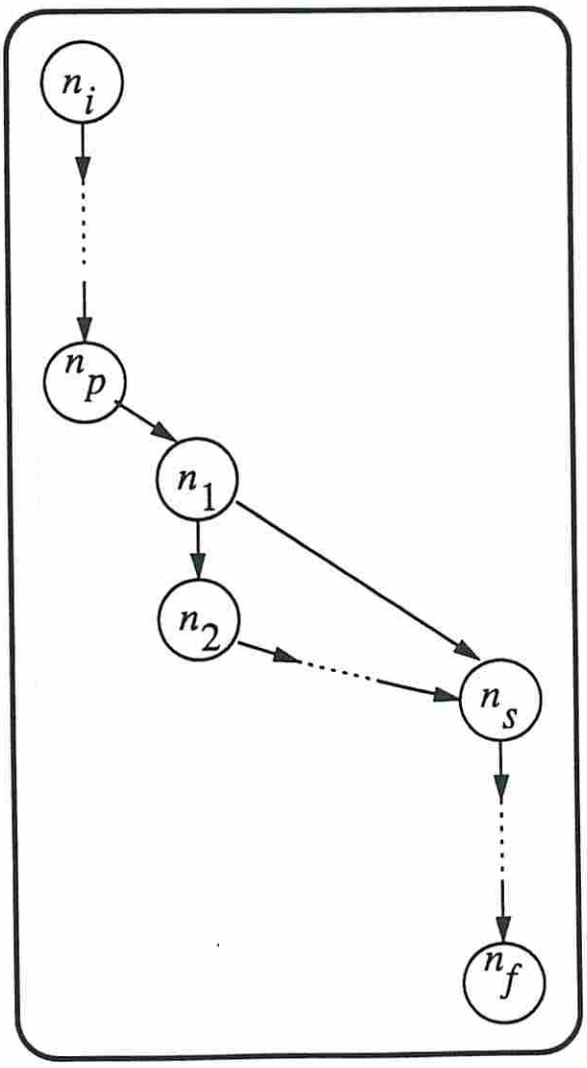

Figure 6: No parallelism loss

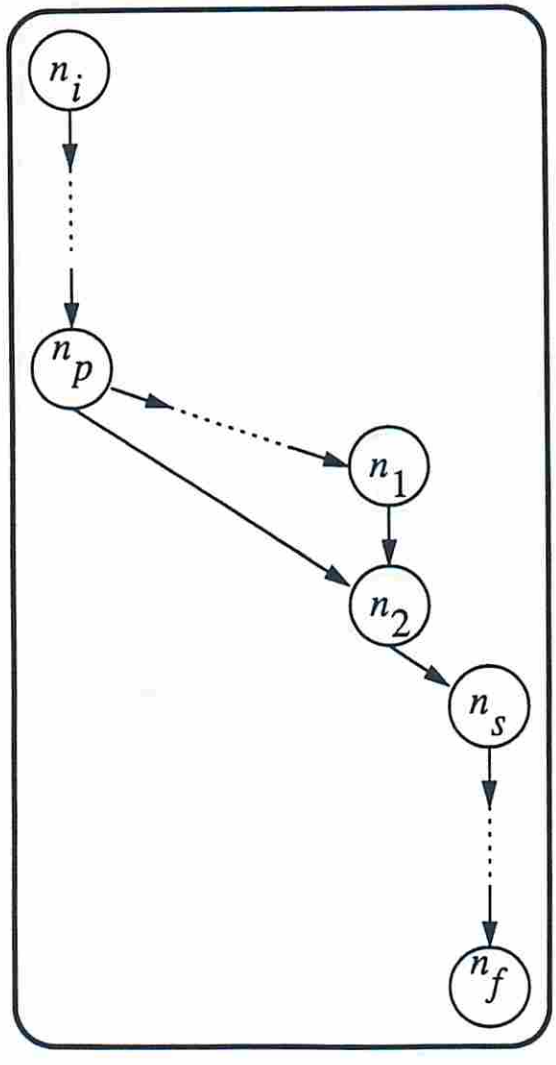

Figure 7: No parallelism loss

Since in general the start-up component is much larger than the delay component, ΔL must be negative. Furthermore, in practice the edges (n_p, n_2) and (n_1, n_s) are most likely not to exist.
This means that $L_a(p) < L_b(p)$. Since $L_b(p) \leq CPL_b$, then $L_a(p) < CPL_b$. This is a contradiction since we assumed that $L_a(p) > CPL_b$.

Case **2** p goes through n_1 and not n_2:
Let $p = (n_i, \ldots, n_p, n_1, n_s, \ldots, n_f)$.
Since $ParSet(n_1) = ParSet(n_2)$ and nodes n_1 and n_s are dependent, then nodes n_2 and n_s have to be dependent as well. Thus either there exists a path from n_2 to n_s or there exists a path from n_s to n_2. If there exists a path from n_s to n_2 then there exists a path from n_1 to n_2 other than (n_1, n_2). Therefore we will have a cycle in the task graph after the merger. Hence we have to disregard this case, which means that there exists a path from n_2 to n_s (see figure 6).
Let $p' = (n_i, \ldots, n_p, n_1, n_2, \ldots, n_s, \ldots, n_f)$.
Let $\Delta L := L_a(p) - L_b(p')$.
For ΔL to have its maximum value, $L_a(p)$ has to be maximized and $L_b(p)$ has to be minimized. Hence, the path from n_2 to n_s has to be constituted of the single edge (n_2, n_s).
Also for $L_a(p)$ to be maximized, we have to have an edge (n_p, n_2) (see figure 5).
Therefore, $p' = (n_i, \ldots, n_p, n_1, n_2, n_s, \ldots, n_f)$.
Hence,
$\Delta L = delay(data(n_p, n_2)) + delay(data(n_1, n_s)) - comm(n_1, n_2)$.
Again, ΔL must be negative. Furthermore, in practice the edges (n_p, n_2) and (n_2, n_s) are most likely not to exist.
This means that $L_a(p) < L_b(p')$. Since $L_b(p') \leq CPL_b$, then $L_a(p) < CPL_b$. This is a contradiction since we assumed that $L_a(p) > CPL_b$.

Case **3** p goes through n_2 and not n_1:
Let $p = (n_i, \ldots, n_p, n_2, n_s, \ldots, n_f)$.
Since $ParSet(n_1) = ParSet(n_2)$ and nodes n_2 and n_p are dependent, then nodes n_1 and n_p have to be dependent as well. Thus either there exists a path from n_1 to n_p or there exists a path from n_p to n_1. If there exists a path from n_1 to n_p then there exists a path from n_1 to n_2 other than (n_1, n_2). Therefore we will have a cycle in the task graph after the merger. Hence we have to disregard this case, which means that there exists a path from n_p to n_1 (see figure 7).
Let $p' = (n_i, \ldots, n_p, \ldots, n_1, n_2, n_s, \ldots, n_f)$.
Let $\Delta L := L_a(p) - L_b(p')$.
For ΔL to have its maximum value, $L_a(p)$ has to be maximized and $L_b(p)$ has to be minimized. Hence, the path from n_p to n_1 has to be constituted of the single edge (n_p, n_1).
Also for $L_a(p)$ to be maximized, we have to have an edge (n_1, n_s) (see figure 5).
Therefore, $p' = (n_i, \ldots, n_p, n_1, n_2, n_s, \ldots, n_f)$.

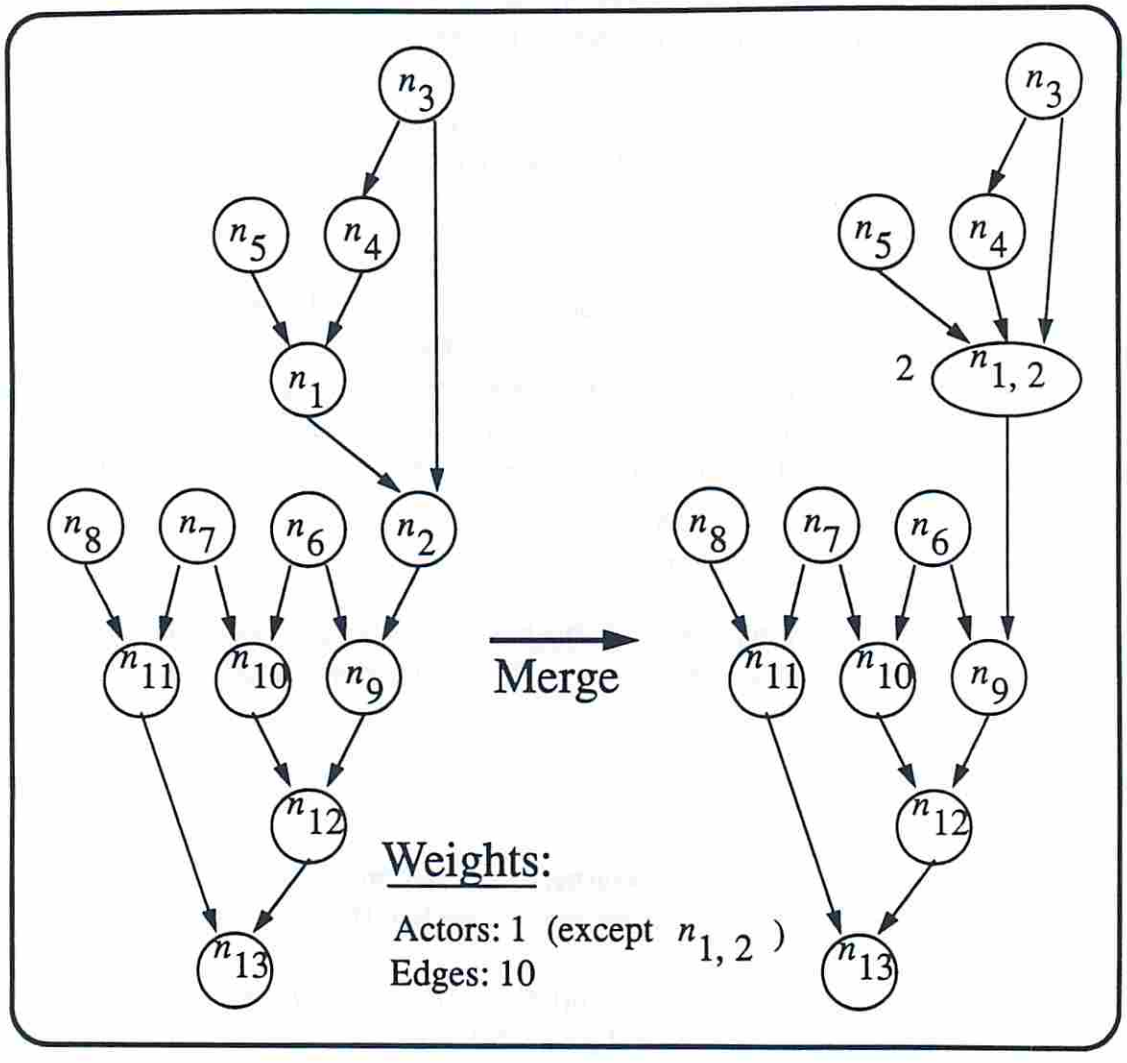

Figure 8: Example: no parallelism loss

Hence,
$\Delta L = delay(data(n_p, n_2)) + delay(data(n_1, n_s)) - comm(n_1, n_2)$.
Again, ΔL must be negative. Furthermore, in practice the edges (n_p, n_1) and (n_1, n_s) are most likely not to exist.
This means that $L_a(p) < L_b(p')$. Since $L_b(p') \leq CPL_b$, then $L_a(p) < CPL_b$. This is a contradiction since we assumed that $L_a(p) > CPL_b$.

Hence, there cannot exist an execution path p such that $L_a(p) > CPL_b$

Examples

- Figure 8 shows a task graph before and after the merger.
  $ParSet(n_1) = \{n_6, n_7, n_8, n_{10}, n_{11}\}$

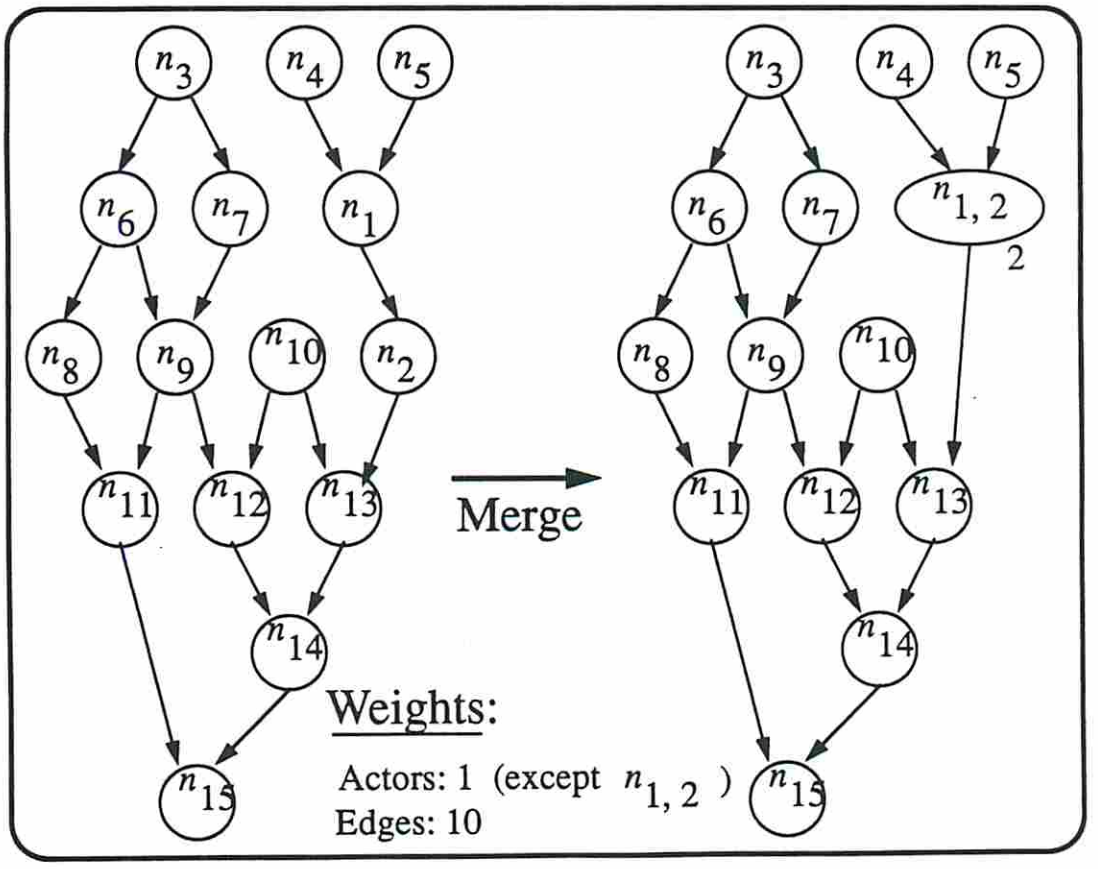

Figure 9: Example: no parallelism loss

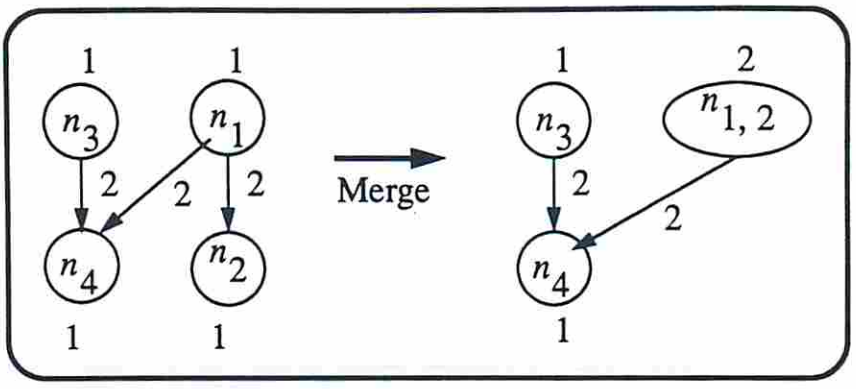

Figure 10: Example: there is parallelism loss

$ParSet(n_2) = \{n_6, n_7, n_8, n_{10}, n_{11}\}$
$ParSet(n_1) = ParSet(n_2)$
Hence, there is no parallelism loss as a result of the merger.
Before merger: CPL = 67.
After merger: CPL = 57.
The CPL has decreased.

- Figure 9 shows a task graph before and after the merger.
  $ParSet(n_1) = \{n_3, n_6, n_7, n_8, , n_9, n_{10}, n_{11}, n_{12}\}$
  $ParSet(n_2) = \{n_3, n_6, n_7, n_8, , n_9, n_{10}, n_{11}, n_{12}\}$
  $ParSet(n_1) = ParSet(n_2)$
  Hence, there is no parallelism loss as a result of the merger.
  Before merger: CPL = 56.
  After merger: CPL = 56.
  The CPL has not changed.

4.3.2 There is Parallelism Loss

Theorem: Assume that there is parallelism loss when we merge n_1 and n_2. Then the CPL of the task graph could increase as a result of the merger.

Proof

Let's prove the claim in the theorem by studying some examples.
In all the figures used in the following examples, the number next to a node represents its execution time, and the number next to an edge represents the communication time caused by the edge.

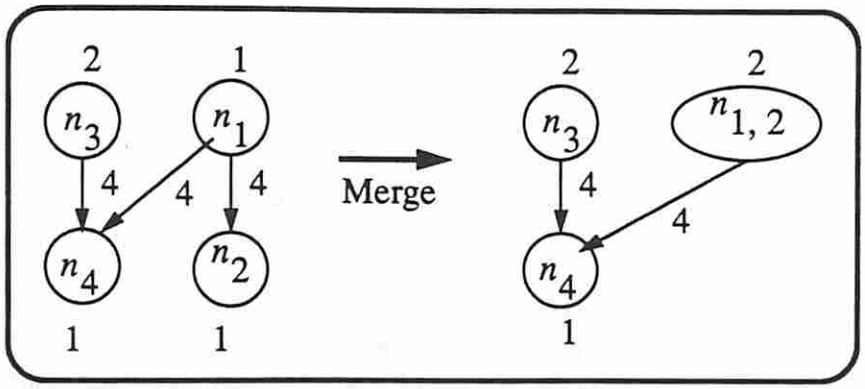

Figure 11: Example: there is parallelism loss

- Figure 10 shows a task graph before and after the merger.
  $ParSet(n_2) = \{n_3, n_4\}$
  $DepSet(n_1) = \{n_2, n_4\}$
  $ParSet(n_2) \cap DepSet(n_1) = \{n_4\}$
  Hence, there is parallelism loss with respect to n_2.
  Before merger: CPL = PARTIME = 4.
  After merger: CPL = 5.
  Thus the CPL has increased.
  Note that before the merger, the graph had a critical path which contained n_1 and not n_2 ((n_1, n_4)), and that is why we have an increase in the CPL.

- Figure 11 shows the same graph as in figure 10, except for the weights.
  Before merger: CPL = 7.
  After merger: CPL = 7.
  Thus the CPL did not change.

4.4 Criteria for Merging

In this section, we list some criteria that will be used by the partitioning heuristics to choose the edge to be merged. We use the results of the previous analysis from the previous sections to obtain these criteria.

- Edge has to belong to a critical path.
- Edge that belongs to all execution paths (if such an edge exists).
- Edge e such that none of the execution paths go through only one of the 2 nodes connected by e.

- Edge $e = (n_1, n_2)$ such that $ParSet(n_1) = ParSet(n_2)$ (no parallelism loss as a result of the merger).
- Edge e with the largest $comm(e)$. This way all execution paths which go through e will decrease in length by a maximum quantity.
- Edge $e = (n_1, n_2)$ such that the merger causes the minimum loss in parallelism: $(|ParSet(n_1)| - |ParSet(n_{1,2})|) + (|ParSet(n_2)| - |ParSet(n_{1,2})|)$ is the smallest.
- Edge e such that the merger causes the least amount of execution paths to increase in length. For instance, we could choose edge $e = (n_1, n_2)$ such that the number of execution paths that go through only one of the 2 nodes n_1 and n_2 is minimum.
- Edge e such that the merger causes the largest number of execution paths to decrease in length. In other words, we look for edge e such that the number of execution paths that go through e is maximum.

We have to make sure that the criteria used in our partitioning algorithm are not too costly. For instance, the 2 last criteria mentioned above require a large time complexity.

5 The Partitioning Heuristic

5.1 Definitions

Safe Edges: Let g be a task graph (DAG). An edge $e = (n_1, n_2)$ is said to be a *safe edge* if merging nodes n_1 and n_2 does not cause any cycles to be created in g. Otherwise, e is said to be an *unsafe edge.*

Perfect Edges: We define a *perfect edge* to be one that belongs to all execution paths in the DAG. Otherwise, the edge is said to be an *imperfect edge.*

Risky Edges: An edge $e = (n_1, n_2)$ is said to be a *risky edge* if there exists at least one execution path that goes through only one of the nodes n_1 and n_2.

5.2 The Heuristic

1. Find heaviest safe edge e in task graph which is *perfect.*
   If there is more than one such edge e, choose the one such that $comp(e)$ has the minimal value. If there is still more than one edge that satisfies that, then choose one randomly.
   If no such edge go to 2, else go to 5.
2. Find heaviest safe edge e in critical path (P_{crit}) which is not *risky.* If there is more than one such edge e, choose the one such that $comp(e)$ has the minimal value. If there is still more than one edge that satisfies that, then choose one randomly.
   If no such edge go to 3, else go to 5.

3. Find Heaviest safe edge $e = (n_1, n_2) \in P_{crit}$ such that
   $\text{ParSet}(n_1) = \text{ParSet}(n_2)$.
   If there is more than one such edge e, choose the one such that $comp(e)$ has the minimal value. If there is still more than one edge that satisfies that, then choose one randomly.
   If no such edge go to 4, else go to 5.

4. Find safe edge $e = (n_1, n_2) \in P_{crit}$ such that
   $(|ParSet(n_1)| - |ParSet(n_{1,2})|) + (|ParSet(n_2)| - |ParSet(n_{1,2})|)$
   is the smallest among all safe edges in P_{crit}.
   If there is more than one such edge e, choose the one such that $comp(e)$ has the minimal value. If there is still more than one edge that satisfies that, then choose one randomly.

5. Merge 2 tasks linked by e.

5.3 Some Properties

The following properties enable us to reduce the time complexity of the partitioning heuristic, by making it easier and quicker to find the edge to be merged.

Lemma: Given a task graph (DAG) and a safe edge $e = (n_1, n_2)$ in the graph.
e is not a *risky* edge $\Longleftrightarrow$
Node n_1 has *only one* output edge (e), and node n_2 has *only one* input edge (e).

Proof

1. Assume that No execution path goes through *only one* of the 2 nodes connected by edge e.

   - If n_1 has more than one output edge (let the other output edge be $e' = (n_1, n_3)$), then there is at least one execution path p that goes through edge (n_1, n_3). Clearly, p cannot go through edge (n_1, n_2) (otherwise p will have a cycle, which means that the graph is not acyclic). Thus, p cannot go through n_2 (otherwise p goes through both nodes n_1 and n_2, which implies that it goes through edge e). Hence p goes through n_1 and not n_2. This contradicts our assumption. Therefore, n_1 has only one output edge.
   - If n_2 has more than one input edge (let the other input edge be $e' = (n_3, n_2)$), then there is at least one execution path p that goes through edge (n_3, n_2). Clearly, p cannot go through edge (n_1, n_2) (otherwise p will have a cycle, which means that the graph is not acyclic). Thus, p cannot go through n_1 (otherwise p goes through both nodes n_1 and n_2, which implies that it goes through edge e). Hence p goes through n_2 and not n_1. This contradicts our assumption. Therefore, n_2 has only one input edge.

2. Assume that node n_1 has *only one* output edge (e), and node n_2 has *only one* input edge (e).

- Any execution path p that goes through n_1 has to go through edge e (since n_1 has only one output edge).
- Any execution path p that goes through n_2 has to go through edge e (since n_2 has only one input edge).

Hence, no execution path goes through only n_1 or only n_2.

Lemma: Given a task graph (DAG) and an edge $e = (n_1, n_2)$ in the graph.
e is a *perfect* edge $\Longrightarrow$

Node n_1 has *only one* output edge (e), and node n_2 has *only one* input edge (e)
AND $ParSet(n_1) = ParSet(n_2) = \emptyset$

Proof

Assume that e is a perfect edge.
If n_1 has more than one output edge or n_2 has more than one input edge, then clearly there exists at least one execution path that doesn't go through e. Hence, e is not a perfect edge, which contradicts our assumption. Therefore, node n_1 has only one output edge, and node n_2 has only one input edge.
Now, let's prove that $ParSet(n_1) = ParSet(n_2) = \emptyset$.
We know that all execution paths go through e. For any node n in the graph other than n_1 and n_2, n belongs to at least one execution path p. Since p goes through e, then p goes through n_1 and n_2. Hence, n and n_1 are dependent and n and n_2 are dependent. Therefore $ParSet(n_1) = ParSet(n_2) = \emptyset$.

5.4 Time Complexity of Partitioning Algorithm

Let E be the number of edges and N be the number of nodes in the program graph. Clearly, the initial task graph will have N nodes and at most E edges.

In the next section, we describe some general DAG traversal techniques. As will be seen later, graph traversal is necessary to determine some notions used by the partitioning algorithm, such as CPL, perfect and safe edges, etc.

5.4.1 DAG Traversal

Traversal of general DAGs is different from tree traversal. With general DAGs, if we are not careful, a node might be visited more than once. Clearly, this is not the case for trees. The reason for this is that for general DAGs, a node may have more than one input edge. In order to avoid visiting nodes more than once, when a node is put in the queue Q, it is marked as *queued*. After a node is visited, only its children which are not marked *queued* are inserted in Q. The algorithm is as follows:

Let Q be a queue (could be implemented as a linked list).
$Q \leftarrow \emptyset$.
Insert all root nodes in Q (in any order)
% No need to mark root nodes as *queued.*
Repeat until $Q = \emptyset$

$n \leftarrow$ Front of Q.
Delete n from Q.
visit(n) % Node n is visited here.
n is marked *visited.* % This marking may not be needed.
IF n is not a leaf node THEN

Insert all children of n that are not marked *queued* in Q, and mark them as *queued* % So that nodes are not visited more than once.
% The way insertion is done depends on the traversal
% type (e.g. breadth-first, depth-first).

Deadlock Situations: We can easily show that the algorithm for DAG traversal listed above never leads to deadlock situations (deadlock means that the algorithm ends and there are still nodes not visited). Refer to [1] for more details.

Depth-First Traversal: For depth-first traversal, the children of the node just visited are inserted at the Front of Q.

Breadth-First Traversal: For breadth-first traversal, the children of the node just visited are inserted at the Rear of Q.

Remark: We can easily show that Breadth-first and depth-first traversals take at the most $O(E + N)$ time complexity.

Parents-First Traversal

In parents-first traversal, a node is not visited until all of its parent nodes are visited. The idea here is to keep a counter for each node in the graph (except the root nodes). This counter is used to keep track of the number of parent nodes of a given node that are already visited. When the counter of some node n is equal to the total number of parents of node n, then we know that all the parent nodes of n are already visited. A child node is inserted in the queue Q only when all of its parents are already visited.
The procedure is as follows:

Let Q be a queue (could be implemented as a linked list).
$Q \leftarrow \emptyset$.
Insert all root nodes in Q (in any order).
FOR all non-root nodes n in the graph DO

% Initialize the counters of the nodes.
$n.count \leftarrow$ Number of parent nodes of n

Repeat until $Q = \emptyset$

$n \leftarrow$ Front of Q.
Delete n from Q.
visit(n) % Node n is visited here.
n is marked *visited.* % This marking may not be needed.
IF n is not a leaf node THEN

FOR all children nodes n' of n DO
$n'.count \leftarrow n'.count - 1$ % One more parent visited.
IF $n'.count = 0$ % All parents of n' are visited.
THEN Insert n' at the Rear of Q.

Remarks

1. Deadlock situations: We can easily show that the algorithm for parents-first traversal never leads to deadlock situations.
2. Time Complexity: $O(E + N)$.

5.4.2 Determining the Notions Used by the Partitioning Algorithm

In this section, we explain very briefly how we determine the notions used by the heuristic of the partitioning algorithm. For the details of this, refer to [1].

CPL

We use a slight variation of parents-first traversal of the task graph. The time complexity is $O(E + N)$.

Perfect Edges

Consider an edge $e = (n_1, n_2)$. From a previous lemma, we know that if n_1 has more than one output edge or n_2 has more that one input edge, then e is not a perfect edge. This check can be done in constant ($O(1)$) time. However, if n_1 has exactly one output edge and n_2 has exactly one input edge, then e could be either perfect or imperfect. In this case, we do a special kind of graph traversal. The idea behind the above procedure is to traverse the graph without going through edge e. If a leaf node is reached, then there must exist at least one path from an input node to an output node which does not go through e. This means that there much exist at least one execution path which does not go through e. If none of the leaf nodes is reached, then there cannot be any path from an input node to an output node which does not go through e. This means that all execution paths must go through e. The time complexity to do that is $O(E + N)$.

Safe Edges

The procedure here is almost the same as the one for perfect edges described above. The time complexity to do that is $O(E + N)$.

Risky Edges

From a previous lemma, we know that an edge $e = (n_1, n_2)$ is not risky *if and only if* n_1 has only one output edge and n_2 has only one input edge. This check can be done in constant time. Hence we need $O(1)$ time to find out whether an edge is risky or not.

DepSet(n)

Let n be a node in the DAG. First, we do a traversal of the graph starting from node n . This will give us all nodes n' such that there is a path from n to n'. Second, we do a backwards traversal of the graph starting from node n (we follow the opposite direction of the edges). This will give us all nodes n' such that there is a path from n' to n. This takes $O(E + N)$ time complexity (complete graph traversal).

ParSet(n)

Let n be a node in the DAG. One way to determine $ParSet(n)$ is to first determine $DepSet(n)$. By doing that, all nodes n' in the DAG such that n and n' are dependent are marked *visited*. Initially, we set $ParSet(n)$ to $\emptyset$. Then we do a complete traversal of the graph, and any node which was not marked *visited* from the traversal to determine $DepSet(n)$ is added to $ParSet(n)$. Note that we have to distinguish between the nodes that are marked *visited* during the graph traversal to determine $DepSet(n)$, and during the complete graph traversal to determine $ParSet(n)$. It takes $O(E + N)$ time complexity to determine $ParSet(n)$.

5.4.3 Time Complexity of Partitioning Algorithm

The following table summarizes the time complexity of each step in the partitioning heuristic. For more details refer to [1].

step 1	step 2	step 3	step 4	step 5
$O(E(E+N))$	$O(E)$	$O(E.N^2)$	$O(E.N^2)$	$O(N)$

Each iteration of the partitioning algorithm consists of choosing the edge to be merged using some heuristic, then the edge chosen is merged[12]. A merging iteration using our heuristic costs at the most $O(E(E + N^2))$. Since there are $N - 1$ merging iterations in the partitioning algorithm, the total cost of the partitioning algorithm is $O(EN(E + N^2))$.

[12]This is called a merging iteration.

We can easily show that $0 \leq E < N(N-1)$. Since $E < N^2$, the time complexity of the partitioning algorithm can be written as $O(E.N^3)$.

Over-Estimation of Time Complexity

In the previous analysis, we over-estimated the time complexity of the partitioning algorithm because we had to assume the worst case scenario. For instance, we used E and N for the numbers of edges and nodes respectively along the critical path. Also we used N for the number of elements in $ParSet(n)$ for various nodes n. In addition, we used E for the number of safe edges along the critical path. Clearly for real applications, the actual numbers are usually much smaller than that. Finally as was mentioned before, merging 2 tasks takes $O(N)$ time in the worst case, but it takes a constant amount of time in the average case (for real applications).

If we assume that the average number of nodes along the critical path, the average number of edges along the critical path, and the average number of elements in ParSet(n) for all nodes n in the task graph are constants, then the average time complexity of our algorithm is $O(N(E+N))$.

6 Performance Analysis

In this section, we study the performance of our partitioning heuristic by considering some regular DAGs.

6.1 Partitioning Fork and Join DAGs

Since a DAG is composed of fork and join components [2, 7, 12, 13], we study the performance of our partitioning algorithm using these primitive structures to further understand its behavior.

6.1.1 Fork DAGs

Consider the fork DAG shown in figure 12. Each c_i is the communication cost of edge (r, n_i), and each e_i is the execution cost of node n_i. Also, e is the execution cost of the root node r. Without loss of generality, assume that the leaf nodes are sorted such that $c_i + e_i \geq c_{i+1} + e_{i+1}$, $1 \leq i \leq m-1$.

Theorem: The optimal partition for the fork DAG is constituted of the following tasks: $\{r, n_1, n_2, \ldots, n_k\}$, $\{n_{k+1}\}$, $\{n_{k+2}\}$, ..., $\{n_m\}$, where $0 \leq k \leq m$. For $k = 0$ the optimal partition is the trivial partition, and for $k = m$ the optimal partition is the singleton partition.

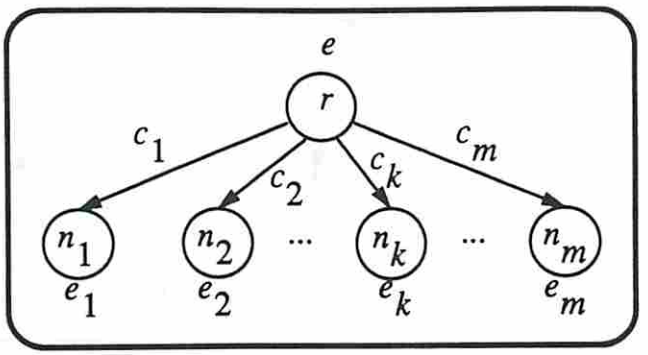

Figure 12: Fork DAG

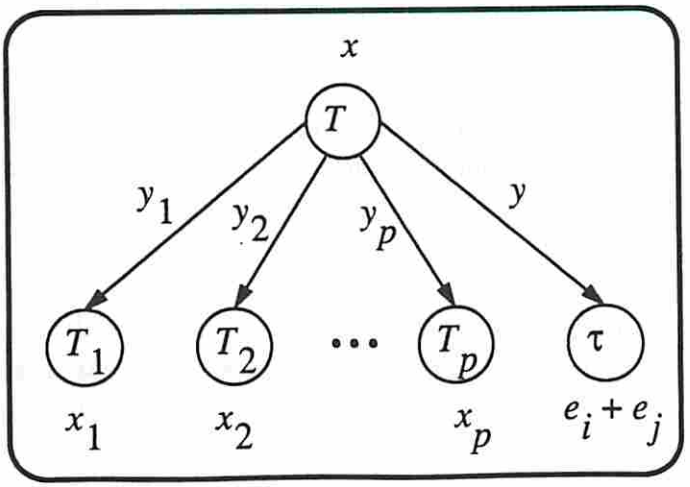

Figure 13: Proof: optimal partition for fork DAGs

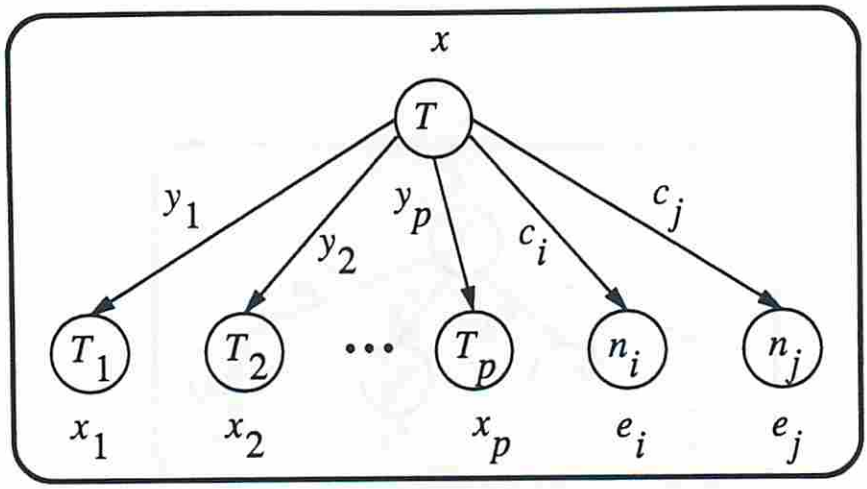

Figure 14: Proof: optimal partition for fork DAGs

Proof

First, let's prove that the optimal partition is constituted of the following tasks:
$\{r, N_1, N_2, \ldots, N_k\}$, $\{N_{k+1}\}$, $\{N_{k+2}\}$, ..., $\{N_m\}$, where $0 \leq k \leq m$ and N_i = some n_j (i not necessarily equal to j).
In other words, in the optimal partition, there is a task containing r and zero or more other n_i's, and the rest of the n_i's are in separate tasks (i.e. these tasks are constituted of a single n_i).
This is also equivalent to saying that in the optimal partition, any task that doesn't contain r cannot contain more than a single n_i.

Intuitively, since merging 2 independent tasks together doesn't reduce communication cost, it will never decrease the parallel execution time (CPL), and therefore, any task in the optimal partition should not consist entirely of n_i's. Hence, any task which does not contain r is constituted of a single n_i.

To prove this more formally, let's show that any partition Π that has a task τ not containing r and that contains more than one n_i has a CPL that is greater or equal than the one of the partition obtained from Π by putting each n_i that belongs to τ in a separate task.
Without any loss of generality, let $\tau = \{n_i, n_j\}$ $(i < j)$. The task graph of Π is shown in figure 13. Clearly, task T contains r[13]. x_k is the execution cost of task T_k. y_k is the communication cost of edge (T, T_k). x is the execution cost of task T. The communication cost of edge (T, τ) is $y = c_i + delay(data(T, \tau))$.
Also without any loss of generality, assume that $y_k + x_k \geq y_{k+1} + x_{k+1}$, $1 \leq k \leq p-1$.

[13] T could be constituted entirely of r.

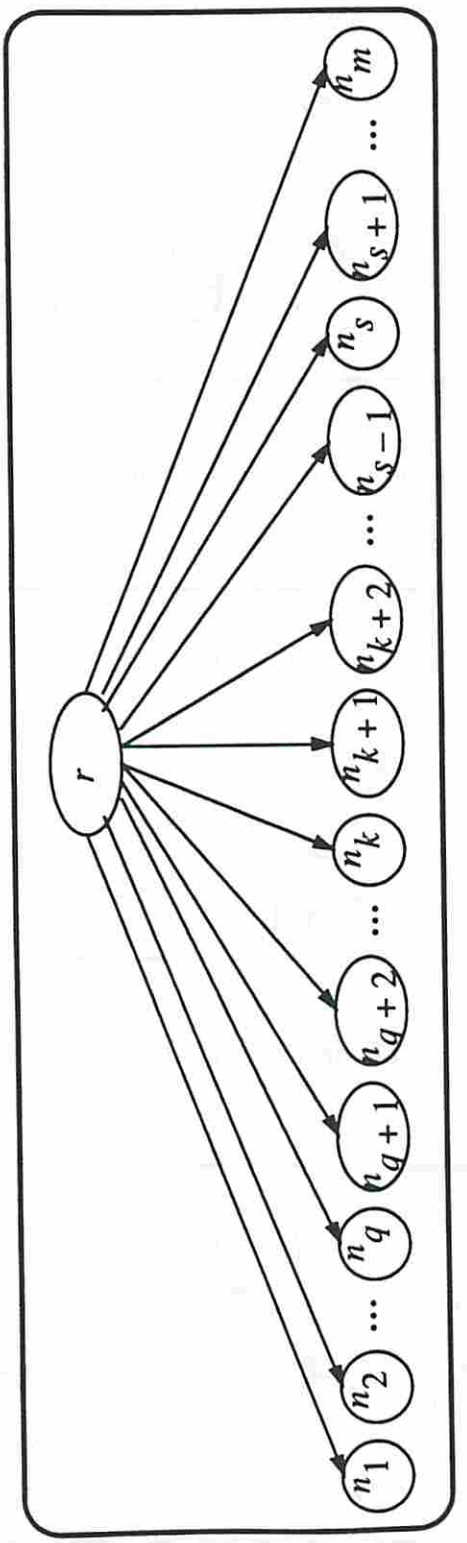

Figure 15: Proof: optimal partition for fork DAGs

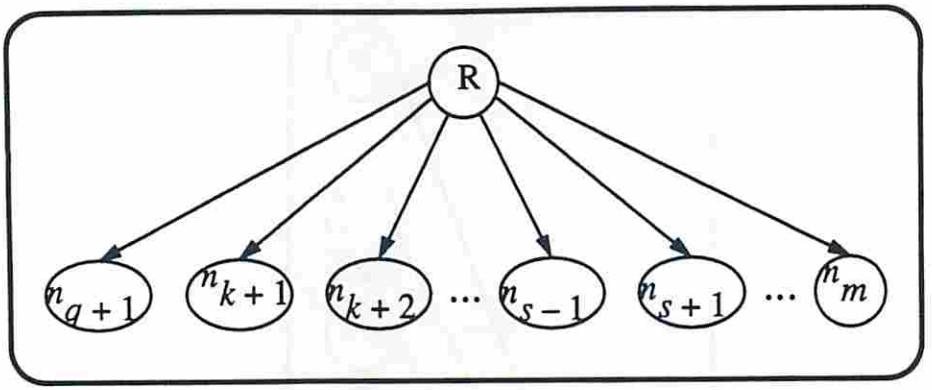

Figure 16: Proof: optimal partition for fork DAGs

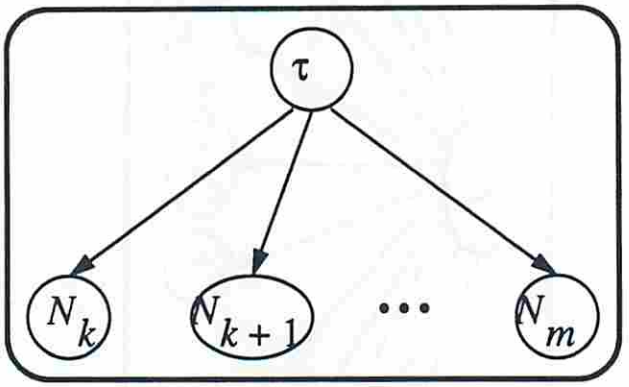

Figure 17: Proof: optimal partition for fork DAGs

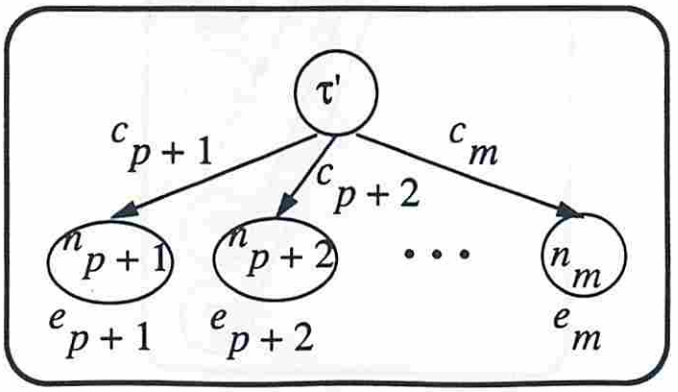

Figure 18: Proof: optimal partition for fork DAGs

The CPL of Π is
$l = x + max(y_1 + x_1, y + e_i + e_j)$.
Now consider the partition Π' obtained from Π by putting n_i and n_j in separate tasks. The task graph corresponding to Π' is shown in figure 14.
The CPL of Π' is
$l' = x + max(y_1 + x_1, c_i + e_i)$.
Since $y + e_i + e_j > c_i + e_i$, then $l \geq l'$.

Now let's prove the claim in the theorem.
First we use an intuitive reasoning.
Let's assume that the claim is not correct.
Hence there exists at least one n_s, $s > k$, such that the optimal partition is constituted of the following tasks:
$R = \{r, n_1, n_2, \ldots, n_q, n_s, n_{q+2}, n_{q+3}, \ldots, n_k\}$, $\{n_{k+1}\}$, $\{n_{k+2}\}$, $\ldots$, $\{n_{s-1}\}$, $\{n_{q+1}\}$, $\{n_{s+1}\}$, $\{n_{s+2}\}$, $\ldots$, $\{n_m\}$.
Refer to figure 15.
The task graph corresponding to this partition is shown in figure 16.
The CPL of this optimal partition is
$l_{opt} = e + e_1 + e_2 + \cdots + e_q + e_{q+2} + e_{q+3} + \cdots + e_k + e_s + c_{q+1} + e_{q+1}$.
Consider the partition constituted of the following tasks:
$\{r, n_1, n_2, \ldots, n_q\}$, $\{n_{q+1}\}$, $\{n_{q+2}\}$, $\ldots$, $\{n_m\}$.
Its CPL is
$l = e + e_1 + e_2 + \cdots + e_q + c_{q+1} + e_{q+1}$.
Note that $l < l_{opt}$, which should not be. Hence we have a contradiction, and therefore our assumption is not possible.
Then the claim of the theorem is correct.

Now let's prove the claim in the theorem more formally.
Assume that the claim is not correct.
The optimal partition Π_{opt} is constituted of the following tasks:
$\tau = \{r, N_1, N_2, \ldots, N_k\}$, $\{N_{k+1}\}$, $\{N_{k+2}\}$, $\ldots$, $\{N_m\}$, where $0 \leq k \leq m$ and N_i = some n_j (i not necessarily equal to j).
Without any loss of generality, assume that $k \geq 1$. When $k = 0$, the optimal partition is the trivial partition, and therefore the claim is true.
Also, without any loss of generality, assume that the N_i's ($1 \leq i \leq k$) in τ are ordered such that if $N_j = n_{j1}$ and $N_{j+1} = n_{j2}$ then $j2 > j1$, $1 \leq j \leq k-1$.
Let $N_0 = r$ and $n_0 = r$.
Let p be the smallest integer such that $N_p = n_p$ and $N_{p+1} \neq n_{p+1}$, $0 \leq p \leq k-1$.
For instance, if $\tau = \{r, n_1, n_2, n_3, n_5, \ldots\}$ then $p = 3$. If $\tau = \{r, n_1, n_5, \ldots\}$ then $p = 1$. If $\tau = \{r, n_2, \ldots\}$ then $p = 0$.
We have $\tau = \{r, n_1, n_2, \ldots, n_p, N_{p+1}, N_{p+2}, \ldots, N_k\}$, $0 \leq p \leq k-1$.
Clearly, $N_i \neq n_{p+1}$, $p+1 \leq i \leq k$. Therefore, n_{p+1} is in a task by itself. The task graph

corresponding to Π_{opt} is shown in figure 17.
Since one of the leaf nodes is n_{p+1} and all n_i's such that $1 \leq i \leq p$ are in task τ, then critical path of this task graph is (τ, n_{p+1}). Hence the CPL of Π_{opt} is
$l_{opt} = comp(\tau) + comm(\tau, n_{p+1}) + comp(n_{p+1})$.
Let the execution time of N_i be E_i, $1 \leq i \leq m$.
$comp(\tau) = e + e_1 + e_2 + \cdots + e_p + E_{p+1} + E_{p+2} + \cdots + E_k$.
$comm(\tau, n_{p+1}) = c_{p+1}$.
$comp(n_{p+1}) = e_{p+1}$.
Hence
$l_{opt} = e + e_1 + e_2 + \cdots + e_p + E_{p+1} + E_{p+2} + \cdots + E_k + c_{p+1} + e_{p+1}$.
Consider the partition Π constituted of the following tasks:
$\tau' = \{r, n_1, n_2, \ldots, n_p\}, \{n_{p+1}\}, \{n_{p+2}\}, \ldots, \{n_m\}$.
The task graph corresponding to Π is shown in figure 18.
Its critical path is (τ', n_{p+1}) and its CPL is
$l = e + e_1 + e_2 + \cdots + e_p + c_{p+1} + e_{p+1}$.
Note that $l < l_{opt}$, which means that we have a contradiction. Therefore, our assumption is not correct, and the claim of the theorem is correct.

Using our Algorithm

It is quite straight forward to show that our partitioning algorithm always leads to the optimal partition.

Using Sarkar's Partitioning Method

Sarkar's algorithm leads to the optimal solution only in some cases, as is stated in the following 2 theorems.

Theorem: If there exists an integer q, $1 \leq q \leq m-2$, such that
$\forall\ k,\ 1 \leq k \leq q,\ c_k \geq e_{k+1} + c_{k+1}$
AND
$\forall\ k,\ q+1 \leq k \leq m-1,\ c_k \leq e_{k+1} + c_{k+1}$
AND
$c_{q+1} \leq e_{q+2} + e_{q+3} + \cdots + e_m$.

then Sarkar's method finds the optimal partition Π_q.

Proof

Clearly, in this case we have
$\forall\ k,\ 1 \leq k \leq q,\ c_k > c_{k+1}$.
Also, since $c_q \geq e_{q+1} + c_{q+1}$, and we know that $c_i + e_i \geq c_{i+1} + e_{i+1}$, $1 \leq i \leq m-1$, then $c_q \geq c_k + e_k$, $q+1 \leq k \leq m$.
Therefore, $c_q > c_k$, $q+1 \leq k \leq m$.

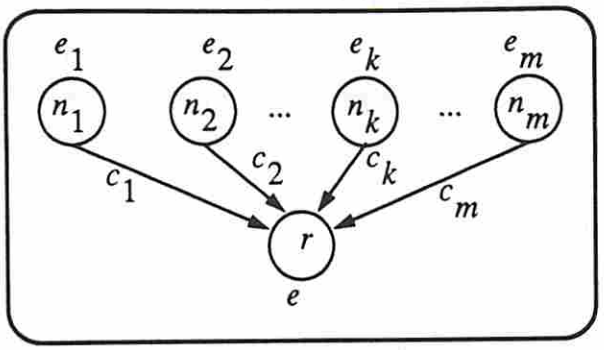

Figure 19: Join DAG

Therefore, the edges are merged in the following order:
(r, n_1), (r, n_2), ..., (r, n_q), ...
The merging of (r, n_1) leads to partition Π_1. Since $l_1 \leq l_0$, this merger is accepted. Next, (r, n_2) is merged and we get partition Π_2. Since $l_2 \leq l_1$, this merger is accepted. Since $l_k \leq l_{k-1}$, $1 \leq k \leq q$, then this process goes on until we reach partition Π_q, which is the optimal partition.

Theorem: Assume that the optimal partition of the fork DAG is Π_p, $1 \leq p \leq m$, and that Π_i, $0 \leq i \leq p-1$, is not an optimal partition.
If there exists an s, $1 \leq s \leq p$, such that merging edge (r, n_s) is not accepted (i.e. we have an increase in the CPL), then Sarkar's method does not find the optimal partition.

Proof

We know that if Π_i is an optimal partition, then $i \geq p$. If edge (r, n_s) is not merged, then we can never obtain any partition Π_i, $s \leq i \leq m$. Hence, the optimal partition can never be obtained.

6.1.2 Join DAGs

Consider the join DAG shown in figure 19. Each c_i is the communication cost of edge (n_i, r), and each e_i is the execution cost of node n_i. Also, e is the execution cost of the leaf node r. Without loss of generality, assume that the root nodes are sorted such that $c_i + e_i \geq c_{i+1} + e_{i+1}$, $1 \leq i \leq m-1$.

The case of join DAGs is the same as the one for fork DAGs, and the analysis here is the same as for fork DAGs. We just have to reverse the direction of the edges in the graph.

6.2 Partitioning Complete Binary Trees

In this section, we assume that the program graph to be partitioned is a complete binary tree. We also assume that all actors in the graph have the same weight, and all edges in the graph have the same weight.

6.2.1 Optimal Partition

Definitions Related to Trees

Consider a directed graph constituted of a tree. We assume that all leaf nodes are at the first level (top most, level 1) of the tree and the root node is at the last level (bottom level, level N where N is the number of levels in the tree) of the tree. The tree is upside down and therefore the leaf nodes are at the top and the root node is at the bottom. The direction of the arcs are from smaller to larger levels.

G-Trees

A G-tree T is a task graph that satisfies the following properties: The number of input arcs of any node is a power of 2, all nodes (tasks) at the same level have the same number of actors, the number of actors in any task (node in the task graph) is equal to the number of input edges of the node corresponding to the task minus 1 (clearly, this is not the case for nodes at level 1), and the sum of the number of actors in all nodes (i.e. tasks corresponding to the nodes) in T except the nodes at the top most level is $2^a - 1$, where 2^a is the number of nodes at the top most level of the tree.

Optimal Task Graph: Given a program graph g that is a complete binary tree, such that all actors in the graph have the same weight and all edges in the graph have the same weight. The optimal task graph is a G-tree.
For the proof of this and all the details concerning this section, please refer to [1].

CPL of G-trees

Assume that our program graph is a complete binary tree with N levels. Consider all G-trees T with m levels ($1 \leq m \leq N$). We have the following: $(2^{a_1} - 1)$ is the number of actors in each node at level 1, $(2^{a_2} - 1)$ is the number of actors in each node at level 2, ..., $(2^{a_m} - 1)$ is the number of actors in each node at level m, where $a_1, a_2, \ldots, a_m$ are positive integers (not zero) such that $a_1 + a_2 + \cdots + a_m = N$.

Let e be the execution time of each actor in the original program graph. Let c be the communication time of each edge in the graph[14]. Then the possible values for the CPL of T are $[(2^{a_1} - 1) + (2^{a_2} - 1) + \cdots + (2^{a_m} - 1)] * e + (m - 1) * c$.

[14]Note that the weight of the edges in any task graph is the same as the weight of the edges in the original program graph.

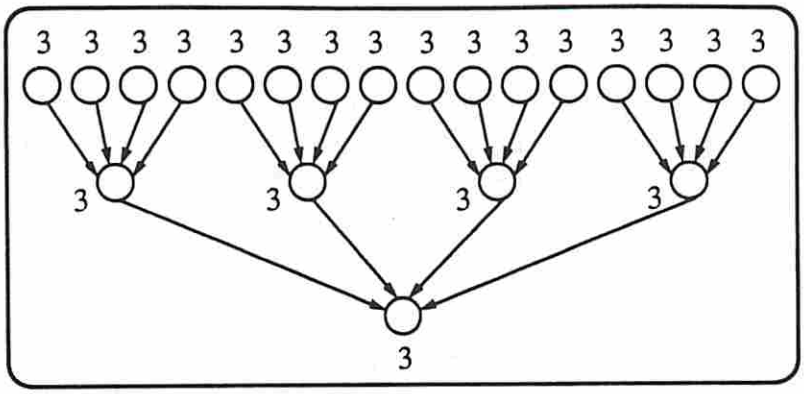

Figure 20: Optimal G-tree with 3 levels

Example

Let $N = 4$ and $m = 3$. Let's find all possible values of (a_1, a_2, a_3). The solution of $a_1 + a_2 + a_3 = 4$ is: $\{(1,1,2),(1,2,1),(2,1,1)\}$. Hence there are 3 possible G-trees with 3 levels.

Minimal CPL

Assume that our program graph is a complete binary tree with N levels. The G-tree with m levels which has the minimal CPL among all G-trees with m levels is one for which the following condition is satisfied:
$A_m = (2^{a_1} - 1) + (2^{a_2} - 1) + \cdots + (2^{a_m} - 1)$ is minimal given that $a_1, a_2, \ldots, a_m$ are positive integers (not zero) such that $a_1 + a_2 + \cdots + a_m = N$.
A_m is minimal when the a_i's are chosen in the following manner:
We divide N as evenly as possible on $a_1, a_2, \ldots, a_m$. In other words, if N is a multiple of m, then each a_i will take the value N/m (using integer division). Otherwise, (N MOD m) a_i's will take the value $(N/m + 1)$ and the rest take the value N/m (using integer division). Hence when N is not a multiple of m, we have more than one solution for $(a_1, a_2, \ldots, a_m)$.

Example

Let $N = 6$ and $m = 3$. The G-tree with 3 levels and $(a_1, a_2, a_3) = (2, 2, 2)$ has minimal CPL among all G-trees with 3 levels. This task graph is shown in figure 20. The number next to each node is the number of actors in the task corresponding to the node.

Optimal G-Tree

Given a complete binary tree with N levels as a program graph, We can show that (refer to [1] for the details) the CPL of the optimal partition can be expressed as

$CPL_{opt} =$

$$
\begin{aligned}
MIN \ \{ \ & (2^N - 1) * e, \\
& MIN\{[(2^{a_1} - 1) + (2^{a_2} - 1)] * e + c \ / \ a_1 + a_2 = N\}, \\
& MIN\{[(2^{a_1} - 1) + (2^{a_2} - 1) + (2^{a_3} - 1)] * e + 2 * c \ / \ a_1 + a_2 + a_3 = N\}, \\
& MIN\{[(2^{a_1} - 1) + (2^{a_2} - 1) + (2^{a_3} - 1) + (2^{a_4} - 1)] * e + 3 * c \ / \ a_1 + a_2 + a_3 + a_4 = N\}, \\
& \ldots\ldots \\
& MIN\{[(2^{a_1} - 1) + (2^{a_2} - 1) + \cdots + (2^{a_N} - 1)] * e + (N-1) * c \ / \ a_1 + a_2 + \cdots + a_N = N\} \\
\} &
\end{aligned}
$$

Note that $\{[(2^{a_1} - 1) + (2^{a_2} - 1) + \cdots + (2^{a_N} - 1)] * e + (N-1) * c \ / \ a_1 + a_2 + \cdots + a_N = N\}$ = $\{[(2^{a_1} - 1) + (2^{a_2} - 1) + \cdots + (2^{a_N} - 1)] * e + (N-1) * c \ / \ a_1 = a_2 = \cdots = a_N = 1\} = \{N * e + (N-1) * c\}$.

Remark: In general, there could be more than one optimal solution.

6.2.2 Best Partition Using our Heuristic

Assume that the execution time of actors is e and the communication cost of edges is c. We find the smallest integer x such that $x \geq 1$ and $2^x \geq \frac{c}{e} + 1$. The best partition using our heuristic is the one for which the task graph is a complete binary tree with $[N - (x-1)]$ levels such that all top most nodes have $(2^x - 1)$ actors and all other nodes have 1 actor. Hence, the corresponding CPL is

$$
\begin{aligned}
CPL &= [(2^x - 1) + ((N - (x-1)) - 1)] * e + [(N - (x-1)) - 1] * c \\
&= (2^x - 1 + N - x) * e + (N - x) * c
\end{aligned}
$$

Example

Assume that our program graph is a complete binary tree with 100 levels. Assume that the execution time of the actors is 1 and the communication cost of edges is 10. The smallest integer x such that $x \geq 1$ and $2^x \geq 11$ is $x = 4$. Hence, the best partition using our heuristic is the one for which the task graph is a complete binary tree with $100 - 3 = 97$ levels such that all top most nodes have $2^4 - 1 = 15$ actors and all other nodes have 1 actor. The corresponding CPL is $CPL = (2^4 - 1 + 100 - 4) * 1 + (100 - 4) * 10 = 1071$.

6.2.3 Performance of our Heuristic

Assume that we have a complete binary tree with N levels. Let the communication cost of edges be $c = 10$ and the execution cost of actors be $e = 1$. We determine the performance of our heuristic by comparing the partition obtained using this heuristic with the optimal partition for various values of N. We assume that the performance of a partitioning algorithm is the inverse of the CPL of the task graph corresponding to the partition obtained using the algorithm. Let l be the CPL of the task graph corresponding to the partition obtained

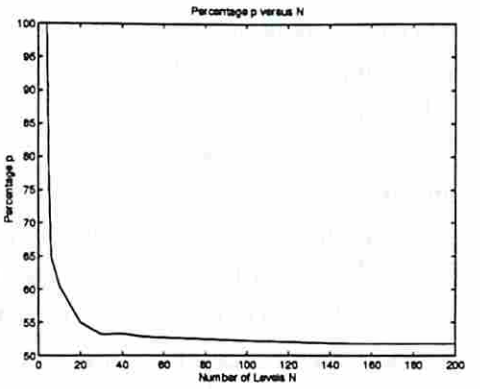

Figure 21: Performance of Heuristic 1

using our Heuristic and l_{opt} be the CPL of the task graph corresponding to the optimal partition. Let p be the percentage of the performance of our heuristic relative to the optimal partitioning algorithm.
$\frac{1}{l_{opt}} * \frac{p}{100} = \frac{1}{l} \Rightarrow p = \frac{l_{opt}}{l} * 100.$
The following table summarizes the results obtained.

N	4	5	6	8	10	20	30	40	50	100	150	200
l_{opt}	15	20	24	37	49	105	160	219	275	559	840	1125
l	15	26	37	59	81	191	301	411	521	1071	1621	2171
p	100	76.9	64.9	62.7	60.5	55.0	53.2	53.3	52.8	52.2	51.8	51.8

Figure 21 shows the plot of p as a function of N. We see from the curve that as N exceeds 25, p starts to decrease very slowly. When N reaches 50, the decrease in p becomes almost negligible. We can safely conclude that the performance of our heuristic is above 50% of the performance of the optimal partitioning algorithm, for any value of N.

6.2.4 Using Sarkar's Partitioning Method

Sarkar's algorithm sorts all edges in the graph by their weight (heaviest first) and merges them in this order. Since all edges in the graph have the same weight, Sarkar's method chooses edges to be merged randomly. Clearly, this could result in very poor performance.

7 Conclusions and Future Research

In this paper, we presented a heuristic for automatic program code partitioning for DMMs. Like most partitioning methods, our approach is compile-time. Given a weighted non-hierarchical (flat) Directed Acyclic Graph (DAG) representation of the program, we proposed a data-flow based partitioning method where all levels of parallelism available in the DAG

are exploited. The output of our algorithm is passed on as input to the scheduling phase. Due to the high cost of graph algorithms, it is not possible to come up with a heuristic that has very low cost and that gives good performance. Hence, we proposed a heuristic that gives reasonably good performance and that has relatively low cost. Our algorithm has a worst case time complexity of $O(E.N^3)$. However as was explained earlier in this paper, for real applications, the average time complexity is expected to be $O(N(E+N))$. For fork and join DAGs, our heuristic gives optimal performance. For complete binary trees, the performance is more than 50% of the optimal one. We need to do more experiments using other kinds of regular DAGs and real life benchmarks to further test our heuristic and improve it.

In addition, we need to find ways to reduce the time complexity of our partitioning algorithm. Following are some methods that could be used to achieve this goal and optimize our proposed algorithm:

- Use *incremental* methods to determine CPL, ParSet(n), DepSet(n), perfect edges, safe edges, instead of recomputing them for each iteration of the algorithm. For instance, we could try to express the CPL at step n as a function of the CPL at step $n-1$ and the way the merger is done at step n.
  Tao Yang [2, 12, 13] uses an incremental way to determine the CPL of a task graph. He defines the *tlevel* and *blevel* of a node n in a DAG to be the length of the longest path from an entry node to n, excluding the weight of n, and the length of the longest path from n to an exit node respectively. Then he defines the priority PRIO(n) of the node n to be $tlevel(n) + blevel(n)$. It is easy to see that the node with the highest priority belongs to the critical path. Using these definitions, Tao Yang was able to make the choice of the edge to be zeroed (i.e. merged) without having to compute the critical path of the task graph, and in an incremental manner. It would be interesting to see if we can use a method similar to his, to determine the edge to be merged in our partitioning algorithm.

- Find optimal solution for *restricted classes* of DAGs.
  We expect that the time complexity of our algorithm becomes much lower when we restrict ourselves to special classes of DAGs. For instance for DAGs for which the nodes have at most one output edge, there are some properties that could enable us to find the edge to be merged in a much cheaper way. Also, we can study DAGs for which each node has at most one input edge (trees).

- In our partitioning algorithm, we keep merging tasks until we reach the coarsest partition (i.e. the partition consisting of a single task). We accept the merger even if it results in an increase in PARTIME. If we can find a way to stop the merging process much earlier without sacrificing performance, then we will have a reduction in the time complexity of the algorithm without losing any performance.

Furthermore, an interesting question to answer would be: if heuristic H_1 gives larger improvements between successive merging iterations than heuristic H_2, does that mean that H_1 performs better than H_2 ?

Finally, an interesting future research would be to investigate merging more than 2 nodes in the task graph (i.e. tasks) at a time.

This will be the topic of future research.

References

[1] Moez Ayed. *Automatic Code Partitioning for Distributed Memory Multiprocessors.* PhD thesis, University of Southern California, EE Systems Dept., Computer Engineering Division, Dec 1996.

[2] Apostolos Gerasoulis and Tao Yang. On the granularity and clustering of directed acyclic task graphs. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):686–701, June 1993.

[3] Matt Haines and Wim Bohm. Towards a distributed memory implementation of sisal. Technical Report CS-91-123, Colorado State University, Computer Science Department, Colorado State University, Fort Collins, CO 80523, Nov 1991.

[4] Matthew Haines and Wim Bohm. A comparison of explicit and implicit programming styles for distributed memory multiprocessors. Technical Report CS-93-104, Colorado State University, March 1993.

[5] Matthew Dennis Haines. Distributed runtime support for task and data management. PHD Dissertation CS-93-110, Colorado State University, August 1993.

[6] S. Hiranandani, K. Kennedy, and C. W. Tseng. Compiling fortran d for mimid distributed-memory machines. *CACM*, 35(8):66–80, Aug 1992.

[7] Cheolwhan Lee, Tao Yang, and Yuan-Fang Wang. Partitioning and scheduling for parallel image processing operations. In *Proceedings of the IEEE Symposium on Parallel and Distributed Processing*, pages 86–90, Texas, October 1995.

[8] V. Sarkar and J. Hennessey. Compile-time partitioning and scheduling of parallel programs. In *Proceedings of the SIGPLAN 1986 Symposium on Compiler Construction*, pages 17–26, Palo Alto, CA, Jun 1986. ACM.

[9] V. Sarkar and J. Hennessey. Partioning parallel programs for macro-dataflow. In *Proceedings of the 1986 ACM Conference on Lisp and functional programming,*, pages 202–211, Aug 1986.

[10] Vivek Sarkar. Partitioning and scheduling parallel programs for execution on multiprocessors. PHD Thesis CSL-TR-87-328, Stanford University, Stanford, CA 94305-2192, Apr 1987.

[11] Vivek Sarkar, Stephen Skedzielewski, and Patrick Miller. An automatically partitioning compiler for sisal. Technical Report UCRL-98289, Lawrence Livermore National Laboratory, Livermore, CA 94550, Dec 1988.

[12] Tao Yang. *Scheduling and Code Generation for Parallel Architectures.* PhD thesis, Rutgers University, New Brunswick, Jew Jersey, May 1993.

[13] Tao Yang and Apostolos Gerasoulis. Dsc: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, September 1994.