Scheduling Tasks and Communications on a Two Levels Virtual Hierarchical Distributed System with Message Contention

Jean-Yves Colin
Laboratoire Litis, Université du Havre, 76610 Le Havre, France
e-mail: jean-yves.colin@univ-lehavre.fr

Moustafa Nakechbandi
Laboratoire Litis, Université du Havre, 76610 Le Havre, France
e-mail: moustafa.nakechbandi@univ-lehavre.fr

Ahmed Salem Ould Cheikh
Université de Nouakchott etc.

June 14, 2011

Abstract

A Directed Acyclic Graph (DAG) of tasks with small communication delays have to be scheduled on the identical parallel processors of clusters connected by a hierarchical network. The number of processors and of clusters is not limited. The communications have to be scheduled too to avoid contention. Task duplication is allowed. In this paper, we present and prove a new polynomial algorithm that, for a DAG with small communication delays and an architecture with communication channels that can transmit at most one message at any time, computes the earliest start dates of all tasks, avoids message contention, and always deliver messages on time. We also extend this result to architectures that include processors shared by two clusters or more.

Keywords: Scheduling, DAG, Hierarchical Communications, Message contention, Task Duplication, CPM/PERT.
1 Introduction

The efficient use of distributed memory multiprocessors and grids is a very difficult problem. An application is made of different parts, with their specific processing times and communication delays. In order to achieve good performances, one needs to schedule these different parts carefully.

In the classical scheduling problem with communication delays, a positive processing time is associated to each task of a Directed Acyclic Graph (DAG) and a positive communication delay is associated to each precedence constraint between the tasks of this DAG. The tasks then have to be scheduled on the processors of the distributed memory multiprocessor or grid. This problem is known to be NP-hard in the general case even if the number of available processors is not limited [9]. Many studies (including [3], [4], [5], [10], [18] and [19]) are currently available on several aspects of this classical scheduling problem with communication delays and a survey can be found in [14].

Task duplication, for example, is used in several studies. Task duplication tries to lower the communication overheads by executing identical copies of some of the tasks on different processors [1], [4], [5] and [13].

Hierarchical communications are taken into account in some studies too. In these problems, the processors are typically grouped into clusters, with communication between processors of the same cluster being faster than communications between processors of different clusters [1] [7] [8].

However it is now increasingly recognized that these classical scheduling problems with communication delays are not realistic, because the communication channels between the processors are not considered, although they are also limiting resources that can deeply affect the actual quality of computed schedules [2] [12] [15] [22]. In [16] for example, the authors show the NP-Completeness of the two processor scheduling problem with tasks of execution time 1 or 2 units, unit interprocessor communication latency and message contention. In [6], the authors offer and prove a CPM/PERT-like polynomial scheduling algorithm for DAG with small communication delays and task duplication, that is optimal and always avoid message contention, if resources are not limited. They consider neither clusters nor hierarchical communication networks, however. More recent studies propose heuristics to avoid contention for communication resources, and present extensive experimental evaluations to evaluate performance improvements [20] [21].

In this paper, we present and prove a new polynomial algorithm that, for a DAG with small communication delays, and an architecture made of clusters and a two level communication network with communication channels that can transmit at most one message at any time, computes, if resources are not
limited, the earliest start dates of all tasks, schedules the communications so that message contention is avoided, and always delivers messages on time. We also extend this result to architectures that include processors shared by two clusters or more.

The rest of this paper is organized as follows.

In the second part of this paper, we present our scheduling problem with hierarchical communication delays and finite capacity communication channels. We propose a new polynomial algorithm and prove that it solves this problem for graphs with small communication delays.

In the third part of this paper, we discuss the effect on the problem, on the solutions and on the algorithm, of additionally having shared processors between two clusters or more. We end with some concluding remarks.

2 The 2LVds Problem

We first define our scheduling problem.

2.1 The 2LVds Model

In our model, a 2-Levels Virtual Distributed System architecture (2LVds) is a distributed memory multi-processor architecture (or grid of processors) with a non limited number of homogenous processors. The processors are grouped into clusters, the number of clusters is not limited and the number of processors in each cluster is also non limited.

There is a complete communication network between these processors, so that each processor is directly connected to every other processors. Each direct connection between any two processors is made of two unidirectional channels, one in each direction (cf. Fig. 1). All communications channels between processors inside all clusters are identical and all communications channels between processors of different clusters are identical too, but slower than the intra-cluster channels.

Each unidirectional channel may carry at most one message at any time so they must be considered as limited resources, even if the number of available processors is not limited. Fig. 2 presents the Gantt chart of a simple problem where an apparently optimal schedule of the five tasks of a DAG on two processors actually needs to overlap on the same channel the transmission of two different messages between two processors.

Fig. 3 presents the Gantt chart of the same mapping of the tasks of Fig. 2 when message contention must be avoided, so the second message must be delayed until the end of the first one.
Figure 1: A 2lVdS architecture.

Figure 2: Example of the apparent schedule of a DAG of five communicating tasks on two processors when message contention is not considered. Arrows are messages between processors. Its makespan is 7.

Figure 3: Real schedule of the example of Fig. 2 when the second message has to be delayed to avoid contention. Its makespan is 8.
In the following, we suppose that each processor belongs to one and only one cluster. However we will consider at the end of this paper the case in which a subset of processors may belong to two clusters or more.

An application is represented by a DAG $G = (V, E)$ (or precedence graph) where $V$ designates the set of tasks, which will be executed on the processors, and where $E$ represents the set of precedence constraints.

Formally, a 2LVDs scheduling problem may then be specified by the four parameters $V, E, p, c$:

$V = \{1, 2, \ldots n\}$: set of $n$ tasks,

$E$: set of arcs $(i, j)$, with $(i, j) \in E$ representing a precedence constraint from task $i \in V$ to task $j \in V$.

$p$: set of processing times, with $p_i \in p$ being the processing time of task $i \in V$ on any processor $\pi$ of the 2LVDs architecture.

$c$: set of communications delays. To each arc $(i, j) \in E$ are associated two positive communication delays $c_{i,j}(1) \in c$ and $c_{i,j}(2) \in c$.

The values $c_{i,j}(1)$ and $c_{i,j}(2)$ in $c$ are defined by:

$c_{i,j}(1)$: positive communication delay of a message from task $i$ to task $j$, if $i$ and $j$ are executed on different processors inside the same cluster (intra-cluster communication delay).

$c_{i,j}(2)$: positive communication delay of a message from task $i$ to task $j$, if $i$ and $j$ are executed in different clusters (inter-cluster communication delay).

We will denote $\text{PRED}(i)$ (respectively $\text{SUC}\,(i)$) the set of immediate predecessors (resp. successors) of task $i$ in $G$.

A task is indivisible, starts when all the data it needs from its predecessors are available, and sends all the data needed by its successors at the end of its execution.

All the immediate successors of a task use the same result from this task, i.e. a task does not give one result (say, a part of a matrix) to one successor and another one (say, the other part of the computed matrix) to another successor. This assumption implies that a task needs to send one message only to a given processor, even if several of its successors are to be processed on it, because one message is enough for all (if this assumption does not hold, the task may usually be divided into sub-tasks such that the assumption is satisfied).
If two communicating tasks $i$ and $j$ are executed on the same processor, there is no need for any communication or its duration is considered negligible, so the communication delay is then 0.

Fig. 4 presents an example of such a DAG. The value above each node is its processing time, and the two values above each arc are its two communication delays.

Figure 4: Example of a DAG with two communication delays.

In most real hierarchical distributed architectures, we have $c_{i,j}(2) = c_{i,j}(1) \times \alpha + \beta$, with $\alpha \geq 1$ and $\beta \geq 0$, for any arc $(i,j)$. This model also covers virtual Local Area Networks of computers in which each computer is powered by one multi-core processor. Each core is then a processor in our model, and each such computer is a cluster. In these cases, $c_{i,j}(1)$ is usually negligible enough to be considered equal to 0, and $c_{i,j}(2)$ is some strictly positive value [1].

In the rest of this paper however, we merely need to have the more general rule $c_{i,j}(2) \geq c_{i,j}(1) \geq 0$.

Task duplication is allowed. That is, several instances (or copies) of the same task may be executed on different processors. We will denote $i_k$ the $k^{th}$ copy of task $i$. Because we must take into account the messages in a schedule, we will denote $m(i_k, j_l)$ a message sent from a copy $i_k$ of a task $i$ to a copy $j_l$ of a task $j$.

A schedule $S$ of a $2LVDS$ scheduling problem is then a 5-tuple $(F, t^c, \pi, M, t^m)$, where
\( F(i) \), is the positive number of copies of task \( i \in V \),
\( t^c(i_k) \), is the starting time of copy \( i_k \) of task \( i \), \( 0 < k < F(i) \),
\( \pi(i_k) \), is the processor assigned to copy \( i_k \) of task \( i \), \( 0 < k < F(i) \),
\( M(i, j) \), is the set of all messages sent by copies of task \( i \) to copies of task \( j \), \((i, j) \in E \),
\( t^m(m(i_k, j_l)) \), is the starting time of message \( m(i_k, j_l) \) sent from the copy \( i_k \) of task \( i \) to the copy \( j_l \) of task \( i \), \( m(i_k, j_l) \in M(i, j) \).

First, to be feasible, a schedule \( S \) must satisfy the following conditions:

- at least one copy of each task is processed, i.e. \( \forall i \in V, F(i) > 0 \),
- at any time, a processor executes at most one copy,
- if \((i, j)\) is an arc of \( E \), then for any copy \( j_l \) of task \( j \), there must exist at least one copy \( i_k \) of task \( i \) that either is on the same processor or sends its message on time to \( j_l \), i.e.
  \[
  \begin{align*}
  \text{if } \pi(j_l) = \pi(i_k) & \text{ then} \\
  \{ \text{they are on the same processor} \} & \\
  t^c(j_l) & \geq t^c(i_k) + p_i \\
  \text{else if } \pi(j_l) \text{ and } \pi(i_k) \text{ are in the same cluster} & \text{ then} \\
  \{ \text{they are not on the same processor} \} & \\
  t^c(j_l) & \geq t^c(i_k) + p_i + c_{i,j}(1) \\
  \text{else} & \\
  \{ \text{they are not in the same cluster} \} & \\
  t^c(j_l) & \geq t^c(i_k) + p_i + c_{i,j}(2) \\
  \end{align*}
  \]

If, in a schedule \( S \), \( i_k \) and \( j_l \) satisfy the above condition, we will say that the Generalized Precedence Constraint is true for the two copies (in short, that \( GPC(i_k, j_l) \) is true). This Generalized Precedence Constraint means that a task needs its data from one copy only of each one of its predecessors.

Second, a feasible schedule \( S \) must additionally satisfy the condition that no channel carries two messages or more simultaneously, i.e. in all channels used to transmit at least two messages \( m(i_k, j_l) \) and \( m(r_t, s_q) \) from a processor \( \pi(i_k) \) to a processor \( \pi(j_l) \), with message \( m(i_k, j_l) \) finishing before message \( m(r_t, s_q) \):

\[
\begin{align*}
  \text{if } \pi(j_l) \text{ and } \pi(i_k) \text{ are in the same cluster} & \text{ then} \\
  t^m(m(r_t, s_q)) & \geq t^m(m(i_k, j_l)) + c_{i,j}(1)
  \end{align*}
\]
else
   \( t^m(m(r_t, s_q)) \geq t^m(m(i_k, j_l)) + c_{i,j}(2) \)
end if

Now, let \( C(i_k) \) be the completion time of a copy \( i_k \) of a task \( i \), i.e. \( C(i_k) = t^c(i_k) + p_i \). The maximum completion time, or makespan, \( C_{\text{max}} \) of a solution \( S \) is the largest completion time of all copies of all tasks in this solution:

\[
C_{\text{max}} = \max_{i \in V, k \leq F(i)} \{ t^c(i_k) + p_i \}
\]

As usual for this kind of problem, we want to minimize \( C_{\text{max}} \), that is, find a feasible solution \( S^* \) with the smallest makespan \( C_{\text{max}}^* \).

One can note that, if \( c_{i,j}(1) = c_{i,j}(2) \), this scheduling problem is actually equivalent to the classical DAG scheduling problem with communication delays, which, in the general case, is a NP-hard problem, even if the number of processors is not limited [17]. So our problem is a NP-hard problem too in the general case. For this reason, we will only consider a DAG satisfying the conditions in the following equations (H1) and (H2), which guarantee that the DAG has small communication delays, even when considering the slower inter-clusters communications.

\[
\forall i \in V, \min_{g \in \text{PRED}(i)} p_g \geq \max_{h \in \text{PRED}(i) - \{g\}} c_{h,i}(1) \quad (\text{H1})
\]

The condition in equation (H1) means that processing times are locally superior or equal to the communication delays inside the clusters. This condition ensures that the earliest start date of any copy of each task may be computed in polynomial time.

\[
\forall i \in V, \min_{k \in \text{SUCC}(i)} p_k \geq \max_{j \in \text{SUCC}(i) - \{k\}} c_{i,j}(2) \quad (\text{H2})
\]

The condition in equation (H2) is very similar to the first. It means too that the processing times are locally superior or equal to the communication delays between the clusters, except that the first one deals with the predecessors of a task, the second one with its successors. Finally, because in our problem these communication delays are superior to the communication delays inside the clusters, the condition in equation (H2) implies that the condition in equation (H1) is true anyway in most cases.

### 2.2 The 2LVdsOpt Algorithm

Due to the way this new scheduling problem is defined, there is already a trivial solution: use one cluster only, and schedule all tasks on the processors of
this cluster. Indeed, there already exists a polynomial algorithm that solves
the problem of scheduling tasks and communications of a DAG satisfying
condition in equation (H1) on a homogeneous Virtual Distributed System,
and that builds a feasible solution with minimal makespan without any mes-
sage contention [6]. Obviously, this algorithm can be used to build a solution
that only uses one cluster, and there is no need for the condition in equation
(H2) in this case because there will not be any inter-cluster communication
in this trivial solution.

This trivial solution, however, is not helpful at all, because real architec-
tures have a limited number of processors in each cluster.

For this reason, we propose the new algorithm 2LVdSOPT. This algo-
rithm schedules the tasks and communications in a 2LVdS problem in poly-
nomial time and spreads the tasks on as many clusters as possible to use less
processors per cluster.

This algorithm has four steps.

The first step 2LVdSLWB() computes the earliest start date of all copies
of each task of the DAG.

The second step 2LVdSCS() computes the critical sequences of the DAG
according to the earliest start dates calculated during the first step.

The third step 2LVdSCC() computes the graph of the critical sequences
of the DAG, and its connected components according to the communication
delays $c_{i,j}(1)$.

The last step 2LVdSBUILD() computes the solution, scheduling the tasks
and communication on the 2LVdS architecture.

2.2.1 Computing the Earliest Start Dates

The first step of 2LVdSOPT computes the earliest start date $b_i$ of all copies
of each task $i$ of the DAG. This is done in procedure 2LVdSLWB() (cf. Algo-

rithm 1).

Table 1 presents the earliest start dates of each task of the DAG of Fig. 4
computed by procedure 2LVdSLWB().

<table>
<thead>
<tr>
<th>task $i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b_i$</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>4</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>0</td>
<td>9</td>
<td>11</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>18</td>
</tr>
</tbody>
</table>

Table 1: Earliest start dates $b_i$ of the tasks of the DAG of Fig. 4 computed
by procedure 2LVdSLWB() (cf. Algorithm 1).
Algorithm 1 procedure $\text{2LVdsLWB}(V, E, p, c)$

for all tasks $i \in V$ such that $\text{PRED}(i) = \emptyset$ do
    let $b_i = 0$ \{assign 0 to $i$ as its earliest start date $b_i$\}
end for

while there is a task $i$ which has not been assigned an earliest starting date $b_i$ and whose predecessors $h \in \text{PRED}(i)$ all have an earliest starting date $b_h$ assigned to them do
    let $c = \max_{h \in \text{PRED}(i)} b_h + p_h + c_{h,i}(1)$
    find $g \in \text{PRED}(i)$ such that $b_h + p_h + c_{h,i}(1) = c$
    let $b_i = \max(b_g + p_g, \max_{h \in \text{PRED}(i) - \{g\}} b_h + p_h + c_{h,i}(1))$
end while

2.2.2 Computing the Critical Sequences

The second step of $\text{2LVdsOPT}$ computes the critical sequences resulting from the earliest start dates calculated during step 1.

Let $B$ be the set of the earliest start dates $b_i$ of all tasks of $V$. Let $G_C$ be the critical subgraph of $G$ according to the earliest start dates in $B$. A critical subgraph is a subgraph of the DAG, with $(i, j)$ being an arc of $G_C$ if $(i, j) \in E$ and $b_j < b_i + p_i + c_{i,j}(1)$. That is, an arc $(i, j)$ in $G_C$ means that these two tasks must have copies on the same processor, because there is not enough delay between the end at date $b_i + p_i$ of any copy $i_k$ of task $i$ and the beginning at date $b_j$ of a copy $j_l$ of task $j$ to transmit the result of $i_k$ to $j_l$ from one processor to another one of the same cluster. The critical subgraph $G_C$ is always a forest [5]. A critical sequence $sc$ of the DAG is simply a proper path of $G_C$.

The computation is done in procedure $\text{2LVdsCSs}()$ (cf. Algorithm 2). Fig. 5 shows the six critical sequences $sc_1$ to $sc_6$ found for the DAG of Fig. 4 using the computed earliest start dates in Table 1.

2.2.3 Building the Graph of the Critical Sequences and Computing its Connected Components

The third step of $\text{2LVdsOPT}$ builds the undirected graph $G_{SC}$ of the critical sequences $sc_s$ and computes its connected components [11].

$G_{SC}$ has one node $s_s$ for each critical sequence $sc_s$ computed during the previous step. Also, there is one edge between two nodes $s_s$ and $s_t$ of $G_{SC}$ if there is at least one arc $(i, j)$ of $E$, with $i \in sc_s$, and $i \notin sc_t$, and $j \in sc_t$, such that $b_j < b_i + p_i + c_{i,j}(2)$. That is, an edge in $G_{SC}$ between two nodes $s_s$ and $s_t$ means that the two corresponding critical sequences $sc_s$ and $sc_t$ must be processed in the same cluster, because there is not enough time to
Algorithm 2 procedure 2t.VdSCs(V, E, p, c, B)

\[ GC = \emptyset \]

for all arcs \((i, j) \in E\) do
  if \(b_j < b_i + p_i + c_{i,j}(1)\) then
    \[ GC = GC \cup \{(i, j)\} \]
  end if
end for

s = 0

for all tasks \(i \in V\) do
  if task \(i\) is a leaf of the critical subgraph \(GC\) then
    let critical sequence \(sc_s\) be the path from the root of the tree in \(GC\) that includes task \(i\), to task \(i\)
    \[ s = s + 1 \]
  end if
end for

---

Figure 5: The six critical sequences \(sc_1\) to \(sc_6\) in the critical graph \(GC\) of the DAG in Fig. 4.
transmit at least one message between at least one task of the first critical sequence to another task of the second critical sequence from one cluster to another different cluster.

Let \( CC \) be the set of all computed critical sequences \( sc_s \).

The computation is done in procedure \( 2lVdSCc() \) (cf. Algorithm 3).

\[ \text{Algorithm 3 procedure } 2lVdSCc(V, E, p, c, B, CC) \]

\[ GSC = \emptyset \]

\[ \text{for all critical sequences } sc_s \in CC \text{ do} \]

\[ \text{let } s_s \text{ be the new node related to } sc_s \]

\[ \text{end for} \]

\[ \text{for all nodes } s_s \text{ do} \]

\[ GSC = GSC \cup \{ s_s \} \]

\[ \text{for all nodes } s_t \in GSC - \{ s_s \} \text{ do} \]

\[ \text{if there is no edge between } s_s \text{ and } s_t \text{ in } GSC \text{ and there is at least one arc } (i, j) \text{ of } E \text{ with } i \in sc_s \text{ and } i \notin sc_t \text{ and } j \in sc_t, \text{ such that } b_i < b_i + p_i + c_{i,j}(2) \text{ then} \]

\[ \text{add one edge between } s_s \text{ and } s_t \text{ to } GSC \]

\[ \text{end if} \]

\[ \text{end for} \]

\[ \text{end for} \]

\[ \text{compute the connected components } g_s \text{ of } GSC \]

Fig. 6 shows the graph of the critical sequences of the DAG of Fig. 4 and its two connected components.

### 2.2.4 Computing the Solution

The last step of \( 2lVdSOPT \) builds a solution with minimal makespan using all the data computed in the preceding phases.

One cluster is allocated to each connected component, and one processor of this cluster is allocated to each critical sequence of this connected component. One copy of each task of each critical sequence is executed at its earliest start date. All messages are sent as soon as the sending copy of the task finishes its execution.

The computation is done in procedure \( 2lVdSBUILD() \) (cf. Algorithm 4).

Fig. 7 shows the Gantt chart of the final schedule found for the DAG of Fig. 4. Two clusters, each with three processors, are used. Tasks 1, 2, 9 and 10 have two copies each in this schedule.
Figure 6: The graph GSC of the six critical sequences of the DAG in Fig. 4 and its two resulting connected components.

**Algorithm 4** procedure 2LVDSBuild($V$, $E$, $p$, $c$, $B$, $CC$, $GSC$)

```plaintext
for all connected components $g_c \in GSC$ do
    allocate a new cluster $\Pi_c$ to $g_c$
    for all node $s_s \in g_c$ do
        let $sc_s$ be the critical sequence related to node $s_s$
        allocate a new processor $\pi_s$ in cluster $\Pi_c$ to this critical sequence $sc_s$
        for all task $i \in sc_s$ do
            \{execute copy $i_{F(i)}$ of $i$ on processor $\pi_s$ at date $b_i$\}
            $F(i) = F(i) + 1$, $t^c(i_{F(i)}) = b_i$, $\pi(i_{F(i)}) = \pi_s$
        end for
    end for
end for
for all copy $j_l$ of task $j$ do
    let $\pi(j_l)$ be the processor that executes $j_l$
    for all task $i \in PRED(j)$ do
        if there is no copy of task $i$ on $\pi(j_l)$ and $\pi(j_l)$ does not already receive one message from any copy of $i$ on time for copy $j_l$ then
            find one copy $i_k$ of task $i$ that can send its message on time to $j_l$
            send one message $m$ from copy $i_k$ at date $b_i + p_i$ to processor $\pi(j_l)$
            if processor $\pi(j_l)$ already receives in the solution another message $m'$ from any copy of $i$ then
                remove message $m'$
            end if
        end if
    end for
end for
```

13
2.3 Complexity of the Algorithm

Let \( n \) be the number of tasks and \( m \) be the number of arcs.

The complexity of procedure \( 2LVDsLWB() \) is \( O(\max(m, n)) \), the complexity of procedure \( 2LVDsCs() \) is \( O(m) \), and building the graph of the critical sequences in \( 2LVDsCc() \) has a complexity of \( O(n) \) [5]. The computation of the connected components of this graph is \( O(n) \). Thus the complexity of procedure \( 2LVDsCc() \) is \( O(n) \) too.

Using a graph-level approach, one can show that the complexity of the first part of \( 2LVDsBuild() \) is \( O(n^2) \). Because the second part of \( 2LVDsBuild() \) tries, in the worst case, to find one suitable copy of each predecessor for each copy of each task, it is possible to establish that the complexity of this second part is \( O(m^2 n^2) \). The complexity of procedure \( 2LVDsBuild() \) is then \( O(m^2 n^2) \).

So the complexity of the overall algorithm is \( O(m^2 n^2) \).

2.4 Analysis of the Algorithm

**Theorem 1** The solution built by algorithm \( 2LVDsOPT \) has a minimal makespan.

**Proof:** Let suppose that there is a \( 2LVDs \) scheduling problem \( P \) satisfying conditions (H1) and (H2) such that there is a possible solution \( S \) with a smaller makespan than the one computed by \( 2LVDsOPT \).

If it exists, such a solution \( S \) can trivially be transformed to be executed on the processors of just one cluster, without any inter-clusters communications and with exactly the same makespan. This transformed solution \( S' \) is
then a possible solution of the equivalent VDS scheduling problem of [5] with the same DAG, same processing times, and $c_{i,j} = c_{i,j}(1)$. The makespan of this solution $S'$ is also lower than the makespan of the solution computed by 2LVdsOPT.

Because 2LVdsOPT computes a solution that has the same makespan than the solution calculated by VDSOPT for the equivalent VDS scheduling problem of [5], the makespan of solution $S'$ is lower than the makespan of the solution calculated by VDSOPT for the equivalent VDS scheduling problem.

However, for this equivalent VDS scheduling problem, the condition (H) of [5] is verified, because its condition (H) is the same than the condition (H1) of our 2LVds scheduling problem. So the VDSOPT algorithm of [5] always computes a solution with minimal makespan in this case and there cannot be any solution $S'$ with a lower makespan than the makespan of the solution calculated by VDSOPT for this VDS scheduling problem.

So we both have a solution $S'$ with a lower makespan than the makespan of the solution computed by VDSOPT in the equivalent VDS scheduling problem, and the fact that there cannot exist any solution with a lower makespan than the makespan of the solution calculated by VDSOPT in the equivalent VDS scheduling problem.

This is not possible, so this implies that such a problem $P$ does not exist. □

**Theorem 2** At least one copy of each task is executed.

**Proof:** In its first part, procedure 2LVdsBUILD() executes one copy of each task of each critical sequence $sc_s$ of each connected component on the processor allocated this critical sequence $sc_s$. Each critical sequence belongs to one connected component. And from [5], we know that each task belongs to at least one critical sequence. So each task has at least one copy executed in the solution built. □

**Theorem 3** The GPC are true for all copies of all tasks.

**Proof:** In the solution built by procedure 2LVdsBUILD(), each copy of each task $i$ is executed at the same earliest start date $b_i$. Let task $j$ be a successor of a task $i$ (i.e. $j \in SUCC(i)$). We always have $b_j \leq b_i + p_i$.

If we have $b_j < b_i + p_i + c_{i,j}(1)$, then any critical sequence that includes task $j$ includes task $i$ too, by definition of a critical sequence. In procedure 2LVdsBUILD(), one copy of each task of a critical sequence is executed on the processor allocated to this critical sequence. Thus, in this first case, for each copy of task $j$, there is one copy of task $i$ executed on the same processor that may directly give it its results.
If we have $b_i + p_i + c_{i,j}(1) \leq b_j < b_i + p_i + c_{i,j}(2)$, then no critical sequence includes both tasks $i$ and $j$. However, their critical sequences are in the same connected component, due at the very least to the arc $(i, j)$. So their critical sequences are all processed in the same cluster. Thus, in this second case, for each copy of task $j$, there is at least one copy of task $i$ executed on a processor in the same cluster that may send its results on time on the intra-cluster network.

Finally, if we have $b_i + p_i + c_{i,j}(2) \leq b_j$, then no critical sequence includes both tasks $i$ and $j$. Their critical sequences may, or may not, be in the same connected component, due to other arcs. In all cases, there is time enough for one copy of task $i$ anywhere to send its results to task $j$ on the inter-clusters network. Thus, in this last case too, for each copy of task $j$, there is a copy of task $i$ executed on a processor somewhere that may send its results on time.

**Theorem 4** In the solution computed, each copy of each task receives at least one message on time from at least one copy of each of its predecessor, if a message is needed.

**Proof:** Procedure 2LVdsBuild() studies each possible couple $(i_k, j_l)$ of each arc $(i, j)$ of the DAG.

Let $(i, j)$ be an arc of $E$. Let $j_l$ be a copy of task $j$ considered during the second part of the building of the solution by procedure 2LVdsBuild().

If there already is one copy $i_k$ of task $i$ on the processor $\pi(j_l)$ executing copy $j_l$ in the solution, then no message is needed by $j_l$ from any copy of task $i$, because the data needed by $j_l$ from one copy of $i$ are already on $\pi(j_l)$.

In all other cases, there is no copy at all of task $i$ on processor $\pi(j_l)$ in the solution. So an on time message is actually needed.

If there already is a message from any copy of $i$ to processor $\pi(j_l)$ on time for copy $j_l$, then there is no need for another message.

Else, if processor $\pi(j_l)$ does not already receive one message from any copy $i_k$ of task $i$ on time for copy $j_l$, it is either because there is no message yet from any copy of $i$ to processor $\pi(j_l)$, or because there already is one message $m_1$ from a copy $i_k$ of $i$ to processor $\pi(j_l)$ but this message $m_1$ is received too late for copy $j_l$.

However, because the GPC are true for all copies of all tasks, there is at least one copy $i_k$ somewhere that can send its message on time to copy $j_l$.

The algorithm will then find it and then have it send a message $m_2$ to copy $j_l$. This message $m_2$ will be on time for copy $j_l$.

Finally, if this new message $m_2$ is added to the solution by procedure 2LVdsBuild() and there already is one message $m_1$ from a copy $i_k$ of task $i$
to processor $\pi(j_l)$, the old message $m_1$ arrives later than the new one, because it is not on time for copy $j_l$, while message $m_2$ is. So the new message $m_2$ is also on time for all copies the old message $m_1$ is on time for. So the old message $m_1$ may always be suppressed harmlessly from the final solution. □

**Theorem 5** There is no collision between messages on any unidirectional channel.

**Proof:** Let us suppose that there is a collision between two messages.

If there is a collision between two messages, it must occur on a unidirectional communication channel from one processor to another. Because these messages must then start from the same processor, the sending tasks must belong to the same critical sequence.

Furthermore, on this unidirectional communication channel, two messages cannot start at the same date, because each message in the solution starts as soon as its sender ends, and because only one copy of any task is executed on any processor at any time, and finally because we have the hypothesis that each copy only needs to send one message at most toward any other processor.

Let $m_1$ be the first message involved in the collision that starts on the given channel, and $m_2$ be the second message, with $t^m(m_1) < t^m(m_2)$. Let $c_1$ be the duration of message $m_1$. If there is a collision between $m_1$ and $m_2$, then we must have $t^m(m_1) < t^m(m_2) < t^m(m_1) + c_1$.

However, the sender of $m_1$ is not the last task of its critical sequence, else there would be no message $m_2$.

Let task $k$ be the direct successor of task $i$ in the critical sequence executed on this processor. Because of condition (H2), we then have $p_k \geq c_1$, even if the channel connects two processors in different clusters. Thus the earliest date any message $m_2$ can start is $t^m(m_1) + p_k$. So we have $t^m(m_2) \geq t^m(m_1) + p_k$. This then implies that $t^m(m_2) \geq t^m(m_1) + c_1$.

So we both have $t^m(m_2) < t^m(m_1) + c_1$ and $t^m(m_2) \geq t^m(m_1) + c_1$. This is not possible so there cannot be a collision between $m_1$ and $m_2$. □

3 Some Remarks

3.1 Processors Shared by Two Clusters

Although we supposed in the model above that each processor belongs to one and only one cluster, one close problem is one in which two clusters share one or more processors. The communications between these shared processors and the processors from either cluster use then the faster value
for each arc $(i,j)$. This may be the case in the real world when two Local Area Networks (each one being considered a cluster) are connected by a computer with two network interface cards. This computer is often used as a router between the clusters, but has some processing power available to execute some of the tasks. This kind of architecture is not strictly hierarchical, but the 2LVDs model can be extended to specify that between any couple of clusters, there also is a non limited number of available processors that belongs to both clusters. Because these shared computers are usually unique between two clusters in the real world, the goal is to use as few of them as possible in a schedule.

Any large connected component, however, may usually be divided into three sub-components. Each of the first two sub-components (the "main" ones) is made of critical sequences that are connected to other critical sequences of the same sub-component, and maybe to other critical sequences of the third subcomponent. There is no connection in the graph of critical sequences between critical sequences of the two "main" sub-components. The third sub-component is then the "glue" between the first two, containing only critical sequences that are simultaneously connected to critical sequences in the two "main" sub-components (cf. Fig. 8).

![Diagram of connected component](image)

Figure 8: Example of a connected component that may be divided into two "main" sub-components and one "glue" sub-component.

The idea then is to process the two "main" sub-components in two separate clusters, while the third one (the "glue") is processed by the processors shared by these clusters. One can note that, if possible, any suitable "cut" of the initial connected component will do. However, if we manage to build
“main” sub-components that are more or less of the same size, the number of processors needed in each cluster is divided more or less by two compared to the number of processors needed in a cluster used for the whole connected component. Of course, the "glue" should be as small as possible too (ideally one critical sequence only) to lower the number of required shared processors. This seems to be a form of the Minimal Cut problem, but further studies are needed.

However, because communication channels are so important in real scheduling problems and because large clusters are not always available, this idea leads us to believe that it may be interesting to build large "virtual" clusters by linking small ones with a few shared computers. Again further studies are needed.

Finally, one can note that processor sharing does not have to be limited to two clusters, but may be extended to more, with some computers belonging to \( N \) clusters simultaneously (there is some practical limit to the number of clusters a real computer may belong to, of course). Then we just divide a connected component into \( N \) "main" sub-components and one "glue" one.

### 3.2 Optimality Conditions

The necessary conditions (H1) and (H2) are very similar, one dealing with the predecessors of a task, the other with its successors. In fact, if condition (H2) is satisfied, (H1) is usually satisfied too. This category of problem is sometimes called a 'medium-grain size' or 'large-grain size' problem (where communications are approximately equal, or are clearly inferior, to processing times), as opposed to 'small-grain size' problems (where communications are clearly superior to processing times), which are much more difficult in general. We believe that these results more precisely define the limits of the fuzzy notion of 'grain-size' in parallel processing (if all successors of any task use the same result from this task):

- if conditions (H1) and (H2) are satisfied, there is a polynomial algorithm able to optimally solve the \( \text{ge2LVDS} \) scheduling problem;
- else the general \( \text{2LVDS} \) scheduling problem is very difficult if condition (H1) is not satisfied, or there are strong risks of conflicts on the communication channels if condition (H2) is not satisfied, or both if neither condition holds.

In the first case, we will say that the grain size of the problem is large. The classical 'unit execution time-unit communication time (UET-UCT)' scheduling problem [18] belongs to this category.
In all other cases, we will say that the grain size of the problem is small. These cases are NP-hard and all heuristics built to deal with it should take into account the communication channels too, because their effects is even more crucial. Note also that the still fuzzier 'medium grain-size' notion disappears, all problems being classified as belonging to the first category or to the other one.

One important result is that this research gives a better hindsight on the importance, when building a parallel program for a cluster of multiprocessor systems, of following the rule 'processing times must be superior or equal to communication times', if possible. Actually, it appears that the performances of the slow inter-cluster network are usually as important or even more important than the performances of the fast intra-cluster networks on a real architecture. Although the slow $c_{i,j}(2)$ values do not appear when computing the makespan of a DAG on a Virtual Distributed System, the larger the communication delays are, the smaller the number of used clusters will need to be, and the larger they will be. Up to the point of only having one cluster used if the inter-cluster communication delays are much larger than the intra-cluster communication delays. Still, there usually is a trade-off between the processing times/communication times’ ratio and the expressed parallelism in a given application. Hence, further work needs to be done, particularly when the number of available clusters or processors is limited.

Also, another facet of this point emphasized by these results is the importance of the communication network as a possible major bottleneck, as important as the number of available processors itself. Having many channels seems usually much better than having only a few faster channels to carry the communications between the processors, and between the clusters too, unless theses channels are much faster. It is possible to have examples of graphs that simultaneously need most or all available channels between one processor and the rest of the processors, even in a distributed architecture with a complete communication network. Real hierarchical architectures, however, usually have only one or, at most, a few communication links between any two clusters, due to the cost of establishing a long distance, complete, dedicated communication network between the clusters. Collisions between messages will occur, or many messages will be delayed, and the real makespan of the solution will be much worse than expected by models that do not consider message contention. This confirms that the global performances on these architectures will probably be much slower than expected. One solution in these cases could be to ensure, one way or another, that communications delays are much smaller than processing times, and some techniques of task clustering may be used to diminish the number of needed communications too.
This is a NP-hard problem however, and additional researches are needed here too to find a good balance between too many communications for the communication network and too few parallel tasks for the processors.

One can note too that the number of used clusters, and the number of used processors, are not minimized in our problem. Instead of allocating one processor to each critical sequence, a processor used by an early finishing critical sequence may sometimes be reused to process a later starting critical sequence. Similarly, a cluster used to process a connected component mainly made of early finishing critical sequences can be reused and given to a connected component mainly made of late starting critical sequences. Incidentally, limiting the number of processors available per cluster seems to give harder problems than just limiting the number of clusters.

Of course, the results of 2LVdSOPT are only results similar to the results computed by the CPM/PERT algorithms for DAG without any communication delays. They can used for example in heuristics dealing with scheduling tasks and communications on real architectures with a limited number of clusters, and a limited number of processors per cluster.

Finally, as the logical following step of these results, we intend, in our future researches, to extend them to hierarchical n-levels distributed architectures.

4 Conclusion

A Directed Acyclic Graph of tasks with small communication delays had to be scheduled on the identical parallel processors of several clusters connected by a hierarchical network. The number of processors and of clusters was not limited. The communications had to be scheduled too to avoid message contention. Task duplication was allowed. We presented and proved a polynomial algorithm that, for a DAG with small communication delays, computes the earliest start dates of tasks, spread them on as many clusters as possible and schedule the communications so that there is no message contention and messages are always delivered on time. We also discussed the effect on the problem, on the solutions and on the algorithm, of additionally having processors shared by two clusters or more.

References


