
ELSEVIER

Parallel Computing 26 (2000) 1109–1128

PARALLEL COMPUTING

www.elsevier.com/locate/parco

List scheduling of general task graphs under LogP

Tomasz Kalinowski [a], Iskander Kort [b,*], Denis Trystram [b]

[a] *Institute of Computer Science, Polish Academy of Sciences, ul Ordona 21 01-237 Warsaw, Poland*
[b] *Laboratoire de Modélisation et de Calcul, BP53, Domaine Universitaire, 38041, Grenoble Cedex 9, France*

Received 1 October 1998; received in revised form 1 June 1999; accepted 1 November 1999

Abstract

List scheduling is the most frequently used scheduling technique. In this context worst case analysis as well as many experimental studies were performed for various computational models. However, many new models have been proposed during the last decade with the aim to provide a realistic but still simple and general model of parallel computation. LogP is one of the most popular models so far suggested. It takes into account the time a computation processor spends to manage a communication. Many experimental studies on current parallel architectures have shown that such a parameter cannot be neglected. The aim of this paper is to assess the applicability of the list scheduling approach to the LogP model. More precisely, we present two adaptations of the earliest task first (ETF) heuristic. Then, we establish an upper bound on list schedules under LogP. Finally, we present an extensive experimental study for different graph classes and model instances. © 2000 Elsevier Science B.V. All rights reserved.

Keywords: List scheduling; LogP model; Communication

1. Introduction

Efficient execution of parallel programs requires constant developing of scheduling strategies. This goal requires the construction of computational models which

* Corresponding author.
E-mail addresses: tkal@ipipan.waw.pl (T. Kalinowski), kort@imag.fr (I. Kort), trystram@imag.fr (D. Trystram).

0167-8191/00/$ - see front matter © 2000 Elsevier Science B.V. All rights reserved.
PII: S0167-8191(00)00031-4

describe the architecture components as well as the way parallel computation is performed. Such models should be precise enough to well reflect the behaviour of contemporary multi-computers being simultaneously general to cover a broad class of architectures. Many computational models have been suggested during the last years. These models can be classified into three categories: shared memory models, distributed memory models and bridging models.

In a shared memory model processors exchange data by issuing read and write requests to a common memory. The first model in this category was the PRAM model [7]. A PRAM machine consists of a set of processors sharing a global memory. A PRAM computation is organised as a sequence of steps. At each step, every processor reads some data from the global memory, performs some operation on it and then writes the results back in the global memory. However, PRAM is too simple to be realistic. Efficient PRAM algorithms often perform poorly on actual parallel architectures. This is because the model neglects communication and synchronisation costs.

In a distributed memory model, processors communicate by exchanging messages through an interconnection network. A representative model in this class is the standard delay model (SDM) [19]. In this model, a machine consists of a set of processors that communicate using an interconnection network. A parallel program is described by a directed acyclic graph (DAG) called a task graph. In such a graph nodes correspond to tasks, whereas arcs correspond to communications. A task cannot start unless it has received data from all its remote predecessors. Upon completion, a task sends its results to all its remote successors. The communication cost is neglected when tasks are assigned to the same processor, and depends on message size otherwise. SDM model takes into account communication latency, that is the time spent by the message to go through the interconnection network. However, the model ignores some other features relevant to current parallel architectures namely *communication overheads*. Communication overhead is the time a computation processor spends managing a data transfer. Many studies have shown that this is not the case for many parallel architectures especially when a high level communication software is used [6]. A second drawback of this model is that the per-processor bandwidth is unbounded. In other words, a processor can send or receive as many messages as needed at a time.

A bridging model describes a parallel architecture using a set of pertinent parameters. The aim is to bridge the gap between the world of parallel programming and the world of parallel architectures. This class includes BSP [21], QSM [8] and LogP [5] models. In this paper, we will focus on the LogP model since it is more realistic than many other models and can be easily extended to cope with various features related to parallel architectures. LogP addresses two aspects of inter-processor communication: communication overheads on computation processors as well as communication latency.

In this work, we present two list scheduling heuristics for the LogP model. List scheduling is the most frequently used technique when the number of processors is bounded. It has been implemented in many programming environments [15,20,23] which confirms its usefulness in practice.

The proposed algorithms are based on the ETF strategy. We also give a worst-case performance bound. The next important problem under consideration is performance comparison of the proposed algorithms. We performed an extensive simulation study for different parallel program attributes, and parallel architecture characteristics. The aim of the conducted simulation was to indicate the area of application of these algorithms.

The structure of this paper is as follows. Section 2 presents some results on scheduling under both SDM and LogP models. Section 3 contains a brief description of the LogP model and presents some basic notations used throughout the paper. In Section 4, the scheduling problem under the LogP model is discussed, the proposed list scheduling algorithms are described and their performance bounds are given. In Section 6, simulation results for the implemented algorithms are reported. Finally, we provide some conclusions on the applicablility of list scheduling to the LogP model as well as some guidelines for future work.

2. Related work

2.1. Scheduling under SDM-like models

Many papers were concerned with the problem of scheduling task graphs in some more or less realistic computational models. In [19], Rayward-Smith showed that the problem of finding a minimum-length schedule for a UECT graph [1] is NP-complete when the number of processors is a parameter of the problem. In [18], the authors showed that the problem is NP-complete for task graphs with a constant communication delay $C, C > 1$, unit execution times and when the number of processors is unbounded.

However, some instances of the scheduling problem remain tractable. In [3], Chrétienne described an optimal polynomial-time scheduling algorithm for trees. This algorithm is based on task replication. The same author showed that the problem is tractable when the number of processors is unbounded and when either the tree is flat or the communication times are not greater than the execution times [2]. A survey of some recent results on scheduling with communication delays can be found in [4].

Many heuristics have been suggested to deal with general task graphs under SDM-like computational models. These heuristics are twofold : *placement* heuristics and *clustering* heuristics. Placement heuristics assume a bounded number of processors. On the other hand, clustering heuristics assign tasks to an unbounded number of virtual processors, so that communication delays are minimized. We focus in this paper on placement heuristics. List scheduling is probably the most frequently used placement algorithm. It has been studied for almost forty years since the pioneer works by Hu [10] and Graham [9]. This approach has been successfully applied for various computational models. Also, a performance bound (Graham's

[1] Unit Execution and Communication Times.

bound) has been computed when no communication delays are considered [9]. Graham's algorithm was first extended by Rayward-Smith to the UECT case [19]. A performance guarantee has been established for this case, as well.

In [11], Hwang et al. presented a list scheduling algorithm called Earliest Task First (ETF). This algorithm deals with arbitrary execution and communication delays. The authors established that the schedules produced by ETF have a length not greater than

$$\left(2-\frac{1}{P}\right)w_{\text{opt}}+C_{\max},$$

where P is the number of processors, w_{opt} the length of an optimal schedule when communication delays are ignored, $C_{\max}$ is the length of a maximum-length path in the graph when execution delays are ignored. Notice that the term $(2-(1/P))w_{\text{opt}}$ is the bound established by Graham. Since the heuristics we suggest in this paper are based on ETF, we present a brief description of this algorithm.

In ETF, the earliest starting time (denoted by e_s) of each *available* task [2] is computed. To do so, the heuristic finds out the set of available tasks (denoted by A) and the set of free processors (denoted by I) at the current moment (denoted by CM). Then, every task T_i in A is tentatively scheduled on each free processor p and the time when all messages incoming from T_i's remote predecessors arrive is computed. Such time is denoted by $r(T_i,p)$ and computed according to the expression:

$$\begin{cases} r(T_i,p)=0 \text{ if } T_i \text{ has no predecessors,} \\ r(T_i,p)=\max\{\sigma(T_j)+w(T_j)+t_c(l(T_j,T_i))|T_j\in \mathit{Pred}(T_i) \text{ and } \pi(T_j)\neq p\}, \end{cases}$$

where $t_c(l(T_j,T_i))$ is the communication time of an $l(T_j,T_i)$ words message, w and π are respectively the weight and allocation function, and σ is the starting time. Thereafter, a task $\hat{T}$ which can be started first on a processor $\hat{p}$ is chosen. If $e_s(\hat{T})$ is greater than the completion time of some currently executed task then the scheduling decision is postponed to take into account tasks which become available at the next decision moment (denoted by NM) and to consider new possible task placements. The complexity of the ETF algorithm is in $O(Pn^2)$ where n is the number of tasks and P is the number of processors. An outline of ETF is given in Algorithm 1.

2.2. Scheduling under LogP

Scheduling task graphs is much more difficult under the LogP model. The problem is NP-complete even for some simple graphs such as fork graphs [22] and coarse-grained inverse trees [16].

Some polynomial-time optimal algorithms have been proposed. In [16], the authors described an algorithm that computes optimal linear schedules for inverse trees. The same authors proposed an algorithm to determine optimal *k-linear* schedules for trees [26]. A schedule is said to be k-linear if each processor is allowed

[2] A task is said to be available at time t if all its predecessors terminate before t.

to execute at most k parallel paths. In [14], optimal polynomial-time algorithms are described for some special cases of the fork graph.

Some heuristics have been presented for the general case. The work in [25] addressed some issues related to scheduling under LogP. In [1], the author proposed a scheduling heuristic based on task replication under a LogP-like computational model.

Algorithm 1. ETF

Begin
{Initially: $I = \{0, \ldots, P-1\}, A = \{T \in V | Pred(T) = \emptyset\}, CM = 0, NM = \infty, Q = \emptyset$}
while $(Q \neq V)$ **do**
  while $(A \neq \emptyset$ **and** $I \neq \emptyset)$ **do**
    {Compute for every task $T \in A$ and for every processor $i \in I$ value $r(T,i)$}
    {Select a task $\hat{T}$ and a processor $\hat{\imath}$ s.t. $\hat{T} \in A, \hat{\imath} \in I$ and
    $r(\hat{T},\hat{\imath}) = \min_{T \in A} \min_{i \in I} r(T,i)$. Let $\hat{e}_s = \max(CM, r(\hat{T},\hat{\imath}))$}
    if $(\hat{e}_s \leqslant NM)$ **then**
      Assign $\hat{T}$ to $\hat{\imath}$
      Update sets A, I, Q.
      if $(\sigma(\hat{T}) + w(\hat{T}) < NM)$ **then**
        $NM := \sigma(\hat{T}) + w(\hat{T})$;
      end if
    else $\{\hat{e}_s > NM\}$
      exit the innermost **while** loop.
    end if
  end while
  {Update CM, NM, A, I}
end while
End

3. Basic concepts and notations

3.1. LogP model

As mentioned in Section 1, the LogP model was defined in [5] with the aim to bridge the gap between the world of parallel programming and the world of parallel architectures. In other words, the model is intended to allow writing parallel algorithms and programs that would sustain good performance when run on a wide range of parallel architectures. In order to achieve such a goal, the model describes a parallel architecture using four parameters:

L (latency) is the time it takes a message to go through the interconnection network.

o (overhead) is the time needed by a computation processor to manage a communication. Managing a communication may correspond to dividing the message into packets at the sender side and to error checking at the receiver side.

g (gap) is the minimum delay required between two consecutive communication events of the same type (i.e. two sends or two receives). In other words, a processor can send (or receive) at most one message every g time units. Such a gap occurs for example when a processor must wait for a communication buffer to be available before sending the next message.

P is the number of processors.

Besides these parameters, the model makes some assumptions on the parallel architecture. First, it is assumed that the interconnection network has a completely connected topology. Second, it is assumed that the network has a bounded capacity. More precisely, at most $\lceil (L/g) \rceil$ messages coming from or destined to a processor can be in transit at any time. Finally, the model assumes that processors run asynchronously.

In LogP, only *elementary messages* can be exchanged between processors. However, it is well known today that communicating long messages leads to better performance on most of the current parallel architectures. In order to overcome this drawback, many researchers defined some LogP extensions [12,17]. These extensions allow processors to exchange messages of arbitrary sizes. Therefore, the model parameters (except P) become functions of message size.

3.2. Definitions

A parallel program is described by a *directed acyclic graph* (DAG) denoted by $G = (V, E, w, l)$. Each node T_i in V represents a task. A weight $w(T_i)$ is associated to T_i. This weight is the execution time of task T_i. Each arc (T_i, T_j) in E represents a communication from T_i to T_j. A weight $l(T_i, T_j)$ is associated to (T_i, T_j). This weight is the length of the message sent by T_i to T_j. Let T_i be a task. We denote by $Pred_G(T_i)$ (resp. $Succ_G(T_i)$) the set of immediate predecessors (resp. successors) of T_i in G.

Let $M = (\boldsymbol{L}, \boldsymbol{o}, \boldsymbol{g}, \boldsymbol{P})$ be a LogP instance such that $\boldsymbol{g} = \boldsymbol{o}$. [3] Moreover, we will restrict ourselves to LogP instances with unbounded capacity. A schedule of G under M is a map S that assigns to every task T_i a pair $(\sigma_S(T_i), \pi_S(T_i))$. $\pi_S(T_i)$ is the processor that will execute T_i. $\sigma_S(T_i)$ is the starting time of T_i on $\pi_S(T_i)$.The overhead parameter can be taken into account by scheduling *communication tasks*. Such tasks are defined as follows. Let (T_i, T_j) be an arc in E. Assume that T_i and T_j are assigned to distinct processors in S. A send task, $send(T_i, d_{ij}, \pi_S(T_j))$ and a receive task $receive(\pi_S(T_i), T_i, d_{ij})$ have to be scheduled on $\pi_S(T_i)$ and $\pi_S(T_j)$, respectively. Task $send(T_i, d_{ij}, \pi_S(T_j))$ sends message d_{ij} produced by T_i to $\pi_S(T_j)$. Similarly, task $receive(\pi_S(T_i), T_i, d_{ij})$ receives message d_{ij} produced by T_i from $\pi_S(T_i)$.

Schedule S is said to be *feasible* under M if and only if the conditions below are fulfilled:

- A processor does not execute more than one task at a time.
- A computation task does not start unless it has received all its data.
- A computation task does not send any of its results before it completes.

[3] Such an assumption holds for some parallel computers [12,13].

- A message cannot be received, unless it has been sent. Moreover, a delay of at least L is required between a send task and its corresponding receive task.
- All tasks in V are scheduled.

Let S be a schedule of G. We denote by $PT(S)$ the *completion time* (also called the makespan) of S. Let (T_i, T_j) be an arc in E. We denote by $L(l(T_i, T_j))$ (resp. $\boldsymbol{o}(l(T_i, T_j))$ the latency (resp. overhead) of (T_i, T_j) communication. The duration of this communication depends on the order according to which T_i sends data to its successors and on the order according to which T_j receives data from its predecessors. The worst case is reached when (T_i, T_j) message is the last message sent by T_i and is the first message received by T_j. Let $L_{\max}(T_i, T_j)$ denote the maximum duration of (T_i, T_j) communication. Then $L_{\max}(T_i, T_j)$ is given by the following expression [17]:

$$L_{\max}(T_i, T_j) = 2\boldsymbol{o}(l(T_i, T_j)) + \boldsymbol{L}(l(T_i, T_j)) + \sum_{T_k \in Succ_G(T_i)\setminus\{T_j\}} \boldsymbol{o}(l(T_i, T_k)) + \sum_{T_k \in Pred_G(T_j)\setminus\{T_i\}} \boldsymbol{o}(l(T_k, T_j)).$$

The *granularity* of task T_i is defined as follows:

$$\begin{cases} \gamma(T_i) = \infty & \text{if } T_i \text{ has no predecessors,} \\ \gamma(T_i) = \dfrac{\min\{w(T_j) | T_j \in Pred_G(T_i)\}}{\max\{L_{\max}(T_j, T_i) | T_j \in Pred_G(T_i)\}} & \text{otherwise.} \end{cases}$$

The granularity of task graph G is

$$\gamma(G) = \min\{\gamma(T_i) | T_i \in V\}.$$

The graph is called *coarse grained* when $\gamma(G) \geqslant 1$, and is called *fine grained*, otherwise.

4. List scheduling under LogP

As mentioned earlier, our algorithms are based on the ETF heuristic. Originally, the ETF was proposed for an SDM-like computational model in which send and receive tasks are not taken into account (i.e., $\boldsymbol{g} = \boldsymbol{o} = 0$). Taking into account these tasks implies further modifications of the original ETF algorithm since the scheduled communications influence the starting times of other data transfers. More specifically, the earliest starting time of each available task has to be re-computed whenever a task is assigned to a processor. This is not the case in the original ETF algorithm. In the proposed extensions to the ETF, we assume that the execution of each task is preceeded by a sequence of receive tasks concerned with incoming communications. After task completion, all outgoing communications are initialized before the execution of any other task on this processor. This simple communication scheme reduces the number of possible assignments of send and receive tasks and so the algorithm complexity. Another problem has been encountered in the implementation. Consider a computation task T. The number of the remote successors of T

cannot be computed unless all the successors of T are scheduled. Thus, the time during which processor $\pi(T)$ is involved in sending the results of T cannot be computed in advance. Two approaches can be applied to cope with this problem. If task starting times can vary during the construction of a schedule, then task executions can be delayed due to allow the insertion of send tasks. Otherwise, if task starting times are fixed then idle slots have to be inserted in the schedule in advance to accommodate prospective send tasks. In our opinion, the first approach can lead to too complex algorithms because the delay in the starting time of one task can propagate along a chain of tasks. We propose two opposite methods of assigning time slots to potential send tasks:

- Two-pass ETF algorithm (2ETF). In the first pass, an ETF schedule is computed assuming that send and receive overheads are a part of communication time but they do not occupy processors. In the second pass, for the given task allocation the schedule is adjusted according to LogP requirements.
- A strategy with reservation (ETFR). It this algorithm, it is assumed that all task successors are to be allocated on other processors. Thus, an idle slot is inserted which is sufficiently large for all potential send tasks.

4.1. Two-pass ETF heuristic

During the first pass, the original ETF algorithm is applied under an SDM model instance in which the cost of a communication (T_i, T_j) is set to be $t_c = 2\boldsymbol{o}(l(T_i, T_j)) + \boldsymbol{L}(l(T_i, T_j))$. During the second pass, communication tasks are inserted using algorithm LogP_FEASIBLE_SCHEDULE (whose code is detailed in Algorithm 2). This algorithm accepts as an input a precedence graph G, a LogP instance M and the schedule S of G under SDM computed by the first pass. It produces a schedule S' of G under LogP. This schedule is feasible under M provided that $\boldsymbol{g} = \boldsymbol{o}$. Schedule S' is computed as follows. Tasks are visited according to increasing values of their starting times in S. At each time step t, the algorithm computes the set C of all tasks that start at t in S. This is done by statement $S.CurrentTasks(C)$ in Algorithm 2. Then, for each task T_j in C, the algorithm schedules the tasks that receive T_j input data from its remote predecessors. Afterward, task T_j is scheduled. Finally, the tasks that send T_j output data to its remote successors are scheduled. Let us calculate the complexity of the 2ETF heuristic. The complexity of the first pass is in $O(n^2P)$ [11]. Consider now the second pass. Collecting tasks that start at time t can be done in $O(P)$. Scheduling a task T_j in C requires visiting all the immediate predecessors and successors of T_j. So this can be done in $O(d_{max})$ where d_{max} is the maximum degree of graph G. On the other hand, the cardinal of C is not greater than P. So, the complexity of the processing performed at each time step is in $O(Pd_{max})$. Moreover, at most n time steps have to be considered since at most n tasks are scheduled on a single processor in S. So, LogP_FEASIBLE_SCHEDULE has a complexity in $O(nPd_{max})$. Since $d_{max} \leqslant n$, the complexity of 2ETF is in $O(n^2P)$ as the original ETF.

Algorithm 2. LogP_FEASIBLE_SCHEDULE

Input

G, w, l: a precedence graph.

$M = (\boldsymbol{L}, \boldsymbol{o}, \boldsymbol{g}, \boldsymbol{P})$: A LogP instance s.t. $\boldsymbol{g} = \boldsymbol{o}$.

S: a schedule of G under SDM.

Output

S': a schedule of G under M.

Begin

{Associate to every processor $i, (0 \leqslant i < P)$ variable $ct(i)$ representing the completion time of i in S'. Initially: $ct(i) = 0,\ \forall i,\ 0 \leqslant i < P$}

$S.CurrentTasks(C)$;

repeat

for each task T_j in C **do**

$\pi_{S'}(T_j) := \pi_S(T_j)$;

for each task $T_k \in Pred(T_j)$ **do**

if $\pi_S(T_k) \neq \pi_S(T_j)$ **then**

$\pi_{S'}(receive(\pi_{S'}(T_k), T_k, d_{kj})) := \pi_{S'}(T_j)$;

$\sigma_{S'}(receive(\pi_{S'}(T_k), T_k, d_{kj})) := \max(ct(\pi_{S'}(T_j)), \sigma_{S'}(send(T_k, d_{kj}, \pi_{S'}(T_j)))$
$+ \boldsymbol{o}(l(T_k, T_j)) + \boldsymbol{L}(l(T_k, T_j))$;

$ct(\pi_{S'}(T_j)) := \sigma_{S'}(receive(\pi_{S'}(T_k), T_k, d_{kj})) + \boldsymbol{o}(l((T_k, T_j))$;

end if

end for

$\sigma_{S'}(T_j) := ct(\pi_{S'}(T_j))$;

$ct(\pi_{S'}(T_j)) := \sigma_{S'}(T_j) + w(T_j)$;

for each task $T_k \in Succ(T_j)$ **do**

if $\pi_S(T_k) \neq \pi_S(T_j)$ **then**

$\pi_{S'}(send(T_j, d_{jk}, \pi_S(T_k))) := \pi_{S'}(T_j)$;

$\sigma_{S'}(send(T_j, d_{jk}, \pi_S(T_k))) := ct(\pi_{S'}(T_j))$;

$ct(\pi_{S'}(T_j)) := \sigma_{S'}(send(T_j, d_{jk}, \pi_S(T_k))) + \boldsymbol{o}(l(T_j, T_k))$;

end if

end for

end for

$S.CurrentTasks(C)$;

until $C = \emptyset$

End

4.2. ETF with reservation heuristic

Like ETF, ETFR assigns highest priorities to tasks that can start the earliest. Nevertheless, in order to prevent a computation task from delaying some send tasks, ETFR reserves enough space to accomodate all send tasks related to a given computation task. More specifically, ETFR proceeds as follows (the code is detailed in Algorithm 3). The set of ready tasks and the set of available processors at the current moment (CM) are determined. These sets are denoted by A and I, respectively. Then,

the heuristic computes for every task T in A and for every processor i in I the time $e_s(T,i)$ at which i may start T. This time is computed as follows. Let T' be an immediate predecessor of T. Let $NSM(T')$ be the next send moment of task T', that is the time at which processor $\pi(T')$ can send the next message of T'. The immediate predecessors of T are then sorted according to non-decreasing values of $NSM(T') + \boldsymbol{o}(l(T',T)) + \boldsymbol{L}(l(T',T))$. This expression is the time at which the message sent by T' to T reaches a remote processor. Let *sPred* be the array composed of the sorted elements of $Pred(T)$. The following procedure is used to compute $e_s(T,i)$.

Begin
$e_s(T,i) := ct(i)$; {$ct(i)$: current completion time of processor i}
for ($j = 1$ **to** $|sPred|$) **do**
  $e_s(T,i) := \max(e_s(T,i), NSM(sPred[j]) + \boldsymbol{o}(l(sPred[j],T)) + \boldsymbol{L}(l(sPred[j],T))$
  $+ \boldsymbol{o}(l(sPred[j],T))$;
end for
End

Afterward, ETFR selects the task $\hat{T}$ and the processor $\hat{\imath}$ that minimize e_s. Tasks that receive the input data of $\hat{T}$ from its remote predecessors are scheduled. Once $\hat{T}$ finishes, ETFR reserves enough space for the tasks that send the results of $\hat{T}$ to its remote successors.

Let $\hat{T}'$ be an immediate successor of $\hat{T}$. Assume that $\hat{T}$ and $\hat{T}'$ are assigned to the same processor. Then, the space reserved for communication $(\hat{T},\hat{T}')$ becomes useless. However, this space is not used by ETFR to schedule other tasks. This leads to a simpler and a more efficient heuristic but may result in unnecessary idle periods. In order to assess the effect of these idle periods on the final schedule length we considered another heuristic called ETFRGC (ETFR with garbage collection). This heuristic proceeds in two phases. In the first phase ETFR is applied. In the second phase, useless idle periods are removed from the ETFR produced schedule. The garbage collector proceeds this way. Tasks are visited according to decreasing values of their starting times (i.e. the Gantt chart is scanned from right to left). At each time step t, the algorithm computes the set C of all tasks that start at time t. This is done analogously r to 2ETF. Let T be the current task. Let f be 0 if T is the first task scheduled on $\pi(T)$ On the contrary, let f be the finish time of the task scheduled just before T on $\pi(T)$. Two cases are distinguished. If T is a computation or a send task, then $\sigma(T)$ is changed to f (thus, the potential idle period $[f,\sigma(T)]$ is removed). Now, assume that T is a receive task. Let fs be the finish time of the sending task associated to T(cf. Fig. 1). Then $\sigma(T)$ is changed to $\max(f, fs + L)$. This ensures that the schedule remains feasible once all the useless idle periods are collected.

Now, let us calculate the complexity of ETFR. The outermost **while** loop repeats at most n times. The innermost **while** loop repeats at most $2n$ times. Remember that computing $e_s(T,i),\ T \in A, i \in I$ requires sorting T incoming communications. So the complexity of this step is in $O(nPd^-_{\max} \log d^-_{\max})$ when all tasks in A and all processors in I are considered. $d^-_{\max}$ denotes the maximum in-degree of G. Scheduling task $\hat{T}$ and reserving space for send tasks have a complexity in $O(d^-_{\max})$ and $O(d^+_{\max})$, respectively.

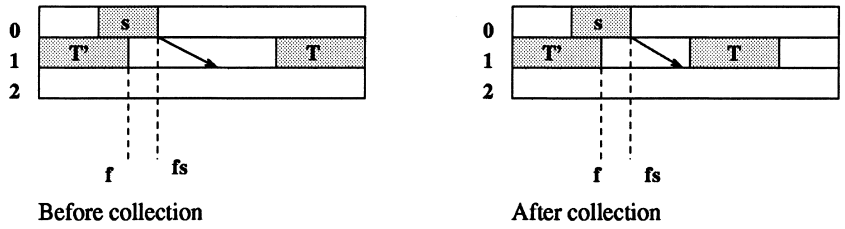

Fig. 1. Collecting useless periods in the case of a receive task.

$d^+_{\max}$ denotes the maximum out-degree of G. So, an iteration of the innermost **while** loop has a cost in $O(nPd^-_{\max}\log d^-_{\max})$.

Therefore, the complexity of ETFR is in $O(n^2Pd^-_{\max}\log d^-_{\max})$. On the other hand, the garbage collection phase of ETFRGC has a complexity in $O(nP)$. So, ETFRGC also has a complexity in $O(n^2Pd^-_{\max}\log d^-_{\max})$.

Algorithm 3. ETFR

Begin
{Initially: $I=\{0,\ldots,P-1\}, A=\{T\in V|Pred(T)=\emptyset\}, CM=0, NM=\infty$,
$Q=\emptyset, ct(i)=0\ \forall i\ i\in\{0,\ldots,P-1\}$}
while $(Q\neq V)$ **do**
  while $(A\neq\emptyset$ **and** $I\neq\emptyset)$ **do**
    {Compute for every task $T\in A$ and for every processor $i\in I$ value $e_s(T,i)$}
    {Select a task $\hat{T}$ and a processor $\hat{\imath}$ s.t. $\hat{T}\in A, \hat{\imath}\in I$ and $e_s(\hat{T},\hat{\imath})=\min_{T\in A}\min_{i\in I} e_s(T,i)$. Let $\hat{e}_s=e_s(\hat{T},\hat{\imath})$}
    if $(\hat{e}_s\leqslant NM)$ **then**
      for all $(T'\in sPred(T)$ **s.t.** $\pi(T')\neq\hat{\imath})$ **do**
        {Schedule necessary communication tasks.}
        $\pi(send(T',d',\hat{\imath})):=\pi(T')$; $\sigma(send(T',d',\hat{\imath})):=NSM(T')$;
        $\pi(receive(\pi(T'),T',d')):=\hat{\imath}$;
        $\sigma(receive(\pi(T'),T',d')):=\max(\sigma(send(T',d',\hat{\imath}))+\boldsymbol{o}(l(T',T))$
        $+\boldsymbol{L}(l(T',T)),ct(\hat{\imath}))$;
        $NSM(T'):=NSM(T')+\boldsymbol{o}(l(T',T))$;
        $ct(\hat{\imath}):=\sigma(receive(\pi(T'),T',d'))+\boldsymbol{o}(l(T',T))$;
      end for
      $\pi(\hat{T}):=\hat{\imath}$; $\sigma(\hat{T}):=\hat{e}_s$;
      $NSM(\hat{T}):=\sigma(\hat{T})+w(\hat{T})$;
      {Let $v=\sum_{T'\in Succ(T)}\boldsymbol{o}(l(\hat{T},T'))$}
      $ct(\hat{\imath}):=\sigma(\hat{T})+w(\hat{T})+v$;{Reserve v time units for send tasks}
      $A:=A\setminus\{\hat{T}\}$;$I:=I\setminus\{\hat{\imath}\}$;$Q:=Q\cup\{\hat{T}\}$;
      if$(ct(\hat{\imath})<NM)$ **then**
        $NM:=ct(\hat{\imath})$;
      end if
    else $\{\hat{e}_s>NM\}$

```
        exit the innermost while loop.
      end if
    end while
    {Update CM, NM, A, I}
  end while
  End
```

5. An upper bound on LogP schedules

Let $G=(V,E,w,l)$ be a task graph. Let $M=(\boldsymbol{L},\boldsymbol{o},\boldsymbol{g},\boldsymbol{P})$ be a LogP instance such that $\boldsymbol{g}=\boldsymbol{o}$. Let S be a schedule of G produced by a heuristic H under a given computational model. This latter could be SDM or even a model where communication costs are neglected. The only claim is that S respects the precedence relation defined by E. Let S' be a LogP schedule of G under M which is derived from S by inserting necessary communication tasks. Assume that a performance guarantee B is already known for heuristic H. An upper bound on the length of S' is given by the theorem below.

Theorem 1. *The length of S' is not greater than $(1+(1/\gamma(G)))B$ where $\gamma(G)$ is the granularity of G.*

Proof. The proof uses an intermediate graph G'. The set of G' nodes is V. We denote by E' the set of arcs in G'. This set is constructed from E as follows:

- Initially E' is equal to E.
- Add to E' *arcs* (T_i,T_j) such that
  - $(T_i,T_j)\notin E$,
  - T_i and T_j are scheduled on the same processor in S and $\sigma_S(T_i)<\sigma_S(T_j)$,
  - there is no task which is scheduled on $\pi_S(T_i)$ between T_i and T_j in S.
- Let (T_i,T_j) be an arc in E'. This arc is deleted from E' iff
  - T_i and T_j are scheduled on the same processor in S,
  - there exists a task T_k such that $\pi_S(T_k)=\pi_S(T_i)$ and $\sigma_S(T_i)<\sigma_S(T_k)<\sigma_S(T_j)$.

  The way to construct E' is shown in Fig. 2.

Each node T_i in G' is associated to a cost $w(T_i)$. Each arc (T_i,T_j) in E' is associated to a cost $c(T_i,T_j)$. This cost is defined as follows:

- if $\pi_S(T_i)\neq\pi_S(T_j)$ then $c(T_i,T_j)=L_{\max}(T_i,T_j)$.
- if $\pi_S(T_i)=\pi_S(T_j)$ then $c(T_i,T_j)=\sum_{T_k\in Succ_G(T_i)}\boldsymbol{o}(l(T_i,T_k))$

We call a cluster in G' the set of tasks assigned to the same processor in S. Remark that each cluster in G' is a path. Moreover, for each path C in G' we have:

$$\sum_{T_i\in C} w(T_i)\leqslant PT(S).$$

Now, let (T_i,T_j) be an arc in E'. If $\pi_S(T_i)=\pi_S(T_j)$ then T_j cannot start in M unless T_i has sent data to all its remote successors. If $\pi_S(T_i)\neq\pi_S(T_j)$ then T_j cannot start unless the data sent by T_i to T_j is available on $\pi_S(T_j)$.

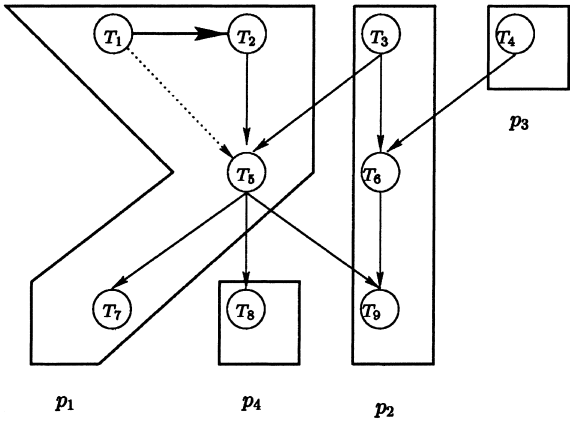

Fig. 2. Assume that T_2 is scheduled just after T_1 on the same processor in S. Then arc (T_1, T_2) is added to E'. After adding this arc (T_1, T_5) becomes redundant, so it is deleted from E'.

Once S' is built, each arc $(T_i, T_j) \in E'$ will have a real cost denoted by $rc(T_i, T_j)$. This cost corresponds to the delay undergone by T_j due to T_i's communications. We have $rc(T_i, T_j) \leqslant c(T_i, T_j)$.

On the other hand, the completion time of S' is the length of a longest path in G' assuming that each arc (T_i, T_j) has a cost of $rc(T_i, T_j)$. Let $CP = (T_{i_1}, \ldots, T_{i_q})$ be a path in G' of maximum length assuming that each arc (T_i, T_j) in E' has a cost of $c(T_i, T_j)$. According to the previous remarks, we deduce that

$$PT(S') \leqslant \sum_{j=1}^{q} w(T_{i_j}) + \sum_{j=1}^{q-1} c(T_{i_j}, T_{i_{j+1}}).$$

Since $c(T_i, T_j) \leqslant L_{\max}(T_i, T_j)\ \forall (T_i, T_j), (T_i, T_j) \in E'$ and using the granularity definition, we deduce that

$$\begin{aligned} PT(S') &\leqslant \sum_{j=1}^{q} w(T_{i_j}) + \sum_{j=1}^{q} \frac{w(T_{i_j})}{\gamma(G)} \leqslant \left(1 + \frac{1}{\gamma(G)}\right) \sum_{j=1}^{q} w(T_{i_j}) \\ &\leqslant \left(1 + \frac{1}{\gamma(G)}\right) PT(S) \leqslant \left(1 + \frac{1}{\gamma(G)}\right) B. \qquad \square \end{aligned}$$

Notice that when the graph is coarse grained the makespan of S' is at most twice B.

The usual technique for deriving upper bounds is to decompose the Gantt chart of a schedule S produced by a given heuristic **H** into computation periods and idle periods. This leads to the expression

$$PT(S) = \frac{1}{P}\left(\sum_{T\in V} w(T) + \sum_{T\in Idle(S)} w(T)\right),$$

where $Idle(S)$ is the set of idle periods in S. Such periods are seen to be virtual tasks having a cost of w. Then, the technique attempts to cover the idle periods by a path in G. Such a path usually exists because the main purpose of a list scheduling algorithm is to efficiently use the system processors by minimizing their idle times.

Things are not so similar under the LogP model. Indeed, the Gantt chart of a schedule S decomposes as follows

$$\begin{aligned} PT(S) &= \frac{1}{P}\left(\sum_{T\in V} w(T) + \sum_{T\in Idle(S)} w(T) + \sum_{T\in Send(S)} w(T) + \sum_{T\in Recv(S)} w(T)\right) \\ &= \frac{1}{P}\left(\sum_{T\in V} w(T) + \sum_{T\in Idle(S)} w(T) + 2\sum_{T\in Send(S)} w(T)\right), \end{aligned}$$

where $Idle(S)$ is defined as above. $Send(S)$ and $Recv(S)$ denote the set of send and receive tasks in schedule S, respectively.

Therefore, a good heuristic should minimize the expression $\sum_{T\in Idle(S)} w(T) + 2\sum_{T\in Send(S)} w(T)$. Unfortunately, minimizing idle periods may involve more send tasks and vice versa. The 2ETF heuristic minimizes idle periods under an SDM model instance. However, inserting communication tasks may lead to a high number of idle periods and send tasks. On the other hand, ETFRGC attempts to minimize idle periods assuming a maximum number of send tasks, then it removes the useless idle periods. So, both heuristics make a special emphasis on minimizing idle periods and do not control the number of the generated send tasks.

6. Experimental results

We have analysed the worst case behaviour of any list scheduling algorithm (including ETF) under LogP. We deal in this section with comparing the two complementary heuristics 2ETF and ETFR. In the first heuristic, computation tasks are assigned to processors without taking into account communication overheads. Then, send and receive tasks are inserted into the schedule. Notice that this approach is followed by mostly all of the current scheduling systems such as PYRROS [24] and HYPERTOOL [23]. On the other hand, the second heuristic assigns computation tasks to processors assuming that all send tasks are scheduled. We will also evaluate the effect of collecting useless idle periods induced by ETFR on the makespan.

Twenty randomly-generated graphs were used in the experiments. Each graph consists of 20 layers and on average eight tasks per layer. These graphs belong to two classes. In the first class (graphs 1–10) the average number of successors of a task (denoted by s) is two. In the second class (graphs 11–20), s is equal to eight. Average execution time is $s * \boldsymbol{o} + \boldsymbol{L}$. Assuming that $\boldsymbol{L} = \boldsymbol{o} = 10$, average execution time is 30 and 90 for the two proposed classes of test graphs. Both the execution times and the

Table 1
Graphs granularities

Graph	$\boldsymbol{L} = 10, \boldsymbol{o} = 10$	$\boldsymbol{L} = 1, \boldsymbol{o} = 10$	$\boldsymbol{L} = 10, \boldsymbol{o} = 1$
1	0.029	0.033	0.125
2	0.029	0.033	0.125
3	0.011	0.012	0.056
4	0.010	0.011	0.053
5	0.013	0.014	0.059
6	0.011	0.012	0.056
7	0.010	0.011	0.053
8	0.010	0.011	0.053
9	0.008	0.009	0.048
10	0.010	0.011	0.053
11	0.005	0.005	0.032
12	0.006	0.006	0.037
13	0.013	0.013	0.080
14	0.028	0.029	0.185
15	0.008	0.008	0.045
16	0.010	0.011	0.053
17	0.040	0.043	0.250
18	0.005	0.006	0.036
19	0.018	0.020	0.100
20	0.021	0.023	0.130

number of successors are uniformly distributed. We mainly aim at answering the following questions. Where could current scheduling approaches like 2ETF be applied? Do ETFR and ETFRGC produce better schedules than 2ETF.

The granularities of the test graphs are shown in Table 1. [4] Notice that graph granularities do not increase significantly when only latency time is reduced (cf. columns 2 and 3 in Table 1). However, the granularity increase is much more important when the overhead is reduced (cf. columns 2 and 4 in Table 1).

The experiments were organised as follows. In a first step, we applied the original ETF algorithm under the SDM model assuming a communication cost of $t_c = 2 * \boldsymbol{o} + \boldsymbol{L}$. Three values of the pair $(\boldsymbol{L}, \boldsymbol{o})$ were considered: $(10, 10), (1, 10)$ and $(10, 1)$. Table 2 shows the results obtained for $P = 8$. Table 3 shows the results obtained for $P = 4$.

In a second step, we conducted experiments for three LogP instances with different communication parameters and such that $P = 8$. The obtained results are shown in Table 4. In a third step we conducted experiments with the same communication parameters as in the previous step, but with only four processors. The obtained results are shown in Table 5.

Let us analyse the results in Tables 2 and 3 first. These tables allow to establish the following. Reducing the latency time from 10 to 1 (cf. columns 3 and 4) involves an average makespan improvement of 3.9% when $P = 4$ and 5.8% when $P = 8$. On the

[4] The granularity recalled in Section 3 is used.

Table 2
Results of the original ETF for a 8-processors machine

Graph	t_{seq}	$\boldsymbol{L} = 10, \boldsymbol{o} = 10$	$\boldsymbol{L} = 1, \boldsymbol{o} = 10$	$\boldsymbol{L} = 10, \boldsymbol{o} = 1$
1	2316	1188	1114	1024
2	3667	1175	1087	1012
3	3875	1201	1084	955
4	4604	1251	1154	1044
5	3124	1021	918	832
6	4588	1245	1143	1066
7	4827	1173	1084	1011
8	4991	1272	1165	1079
9	4481	1291	1193	1091
10	3497	1141	1021	937
11	14 365	3562	3416	3312
12	14 783	3514	3367	3262
13	16 458	3810	3644	3559
14	12 455	3512	3422	3254
15	14 802	3513	3352	3244
16	13 111	3469	3361	3255
17	12 658	3364	3232	3093
18	15 420	3521	3394	3275
19	12 652	3497	3365	3233
20	17 775	3791	3749	3548

Table 3
Results of the original ETF for a 4-processors machine

Graph	t_{seq}	$\boldsymbol{L} = 10, \boldsymbol{o} = 10$	$\boldsymbol{L} = 1, \boldsymbol{o} = 10$	$\boldsymbol{L} = 10, \boldsymbol{o} = 1$
1	2316	1239	1137	1062
2	3667	1342	1271	1234
3	3875	1351	1284	1244
4	4604	1483	1403	1344
5	3124	1171	1086	1045
6	4588	1468	1407	1369
7	4827	1464	1425	1400
8	4991	1480	1435	1374
9	4481	1495	1425	1346
10	3497	1311	1200	1149
11	14 365	4466	4326	4271
12	14 783	4465	4331	4306
13	16 458	5102	5032	4969
14	12 455	4374	4303	4245
15	14 802	4464	4356	4308
16	13 111	4513	4408	4323
17	12 658	4172	4202	3998
18	15 420	4628	4547	4453
19	12 652	4358	4292	4209
20	17 775	5042	4981	4902

other hand, reducing the overhead from 10 to 1 (cf. columns 3 and 5) involves an average makespan improvement of 8.3% when $P = 4$ and 11.6% when $P = 8$. The improvements are greater in the case of 8-processors machines. This is not surprising since the interconnection network becomes the bottleneck when the processors number is high.

Now, let us compare the results of the original ETF algorithm with those of the 2ETF algorithm. The purpose of this comparison is to assess the effect of inserting communication tasks into an SDM schedule. We noticed that inserting tasks drastically increases the makespan by 43% in the case when $\boldsymbol{L} = \boldsymbol{o} = 10$ and by 48% in the case when $\boldsymbol{L} = 1$ and $\boldsymbol{o} = 10$ (This is an average over the twenty graphs and assuming that $P = 8$). However, the makespan increases only by 4% when $\boldsymbol{L} = 10$ and $\boldsymbol{o} = 1$. So, as expected, inserting communication tasks increases drastically the makespan in machines where the overhead is a bottleneck.

Tables 4 and 5 show that algorithms 2ETF, ETFR and ETFRGC produce schedules of nearly the same length under the third LogP instance (i.e. $\boldsymbol{L} = 10$ and $\boldsymbol{o} = 1$). In such instance, the overhead is not an important issue. So, whatever algorithm is applied (original ETF, 2ETF, ETFR or ETFRGC) the obtained schedules do not differ too much. Hence, we will not consider the two LogP instances ($\boldsymbol{L} = 10, \boldsymbol{o} = 1, \boldsymbol{P} = 4$) and ($\boldsymbol{L} = 10, \boldsymbol{o} = 1, \boldsymbol{P} = 8$) in the rest of our discussion.

Now let us compare 2ETF and ETFR. Table 4 shows that taking overheads into account while scheduling (ETFR case) leads to better makespans than inserting

Table 4
Schedules lengths when $P = 8$

Graph	t_{seq}	$\boldsymbol{L} = \boldsymbol{o} = 10$			$\boldsymbol{L} = 1, \boldsymbol{o} = 10$			$\boldsymbol{L} = 10, \boldsymbol{o} = 1$		
		2ETF	**ETFR**	**ETFRGC**	**2ETF**	**ETFR**	**ETFRGC**	**2ETF**	**ETFR**	**ETFRGC**
1	2316	1522	1516	1440	1380	1441	1358	1046	1058	1045
2	3667	1618	1609	1511	1561	1507	1416	1041	1074	1054
3	3875	1710	1558	1459	1657	1470	1370	991	991	977
4	4604	1815	1720	1654	1628	1623	1575	1087	1133	1120
5	3124	1428	1415	1297	1374	1375	1289	855	869	855
6	4588	1817	1788	1683	1777	1720	1651	1114	1118	1106
7	4827	1662	1633	1504	1659	1583	1459	1055	1045	1032
8	4991	1745	1682	1597	1841	1646	1566	1111	1130	1119
9	4481	1883	1714	1654	1812	1650	1582	1136	1127	1117
10	3497	1632	1475	1376	1475	1428	1304	970	975	957
11	14 365	5275	4875	4845	5217	4639	4552	3484	3455	3447
12	14 783	5149	4641	4580	5184	4667	4611	3392	3319	3305
13	16 458	5991	5345	5275	5694	5275	5195	3758	3825	3808
14	12 455	4742	4349	4293	4548	4343	4290	3339	3358	3344
15	14 802	5154	4718	4630	5067	4571	4481	3363	3415	3401
16	13 111	4936	4595	4514	4960	4525	4459	3388	3399	3383
17	12 658	4470	4196	4149	4335	4025	3985	3189	3225	3218
18	15 420	5088	4746	4676	5160	4552	4455	3397	3474	3460
19	12 652	4953	4548	4477	4922	4458	4350	3344	3284	3270
20	17 775	5869	5193	5116	5885	5115	5036	3723	3713	3703

Table 5
Schedules lengths when $P = 4$

Graph	t_{seq}	$L = o = 10$			$L = 1, o = 10$			$L = 10, o = 1$		
		2ETF	**ETFR**	**ETFRGC**	**2ETF**	**ETFR**	**ETFRGC**	**2ETF**	**ETFR**	**ETFRGC**
1	2316	1586	1622	1446	1483	1602	1467	1092	1091	1074
2	3667	1919	2117	1857	1920	2118	1858	1295	1327	1301
3	3875	1963	2191	1900	2058	2160	1900	1314	1297	1269
4	4604	2381	2445	2166	2292	2442	2149	1435	1455	1422
5	3124	1704	1931	1663	1610	1901	1654	1103	1119	1094
6	4588	2286	2491	2201	2439	2504	2212	1471	1470	1444
7	4827	2236	2519	2171	2295	2492	2087	1491	1493	1455
8	4991	2516	2549	2154	2396	2560	2227	1475	1503	1468
9	4481	2314	2382	2126	2331	2357	2041	1430	1456	1430
10	3497	1801	1989	1735	1917	2017	1753	1206	1227	1197
11	14 365	6694	6862	6421	6621	6801	6298	4497	4579	4547
12	14 783	7181	6938	6554	6974	6992	6579	4539	4563	4526
13	16 458	8103	8145	7645	8076	7877	7432	5232	5229	5165
14	12 455	6153	6105	5770	6084	6004	5688	4397	4370	4331
15	14 802	6871	6945	6580	6656	6812	6358	4543	4551	4505
16	13 111	6388	6443	6095	6220	6469	6199	4499	4536	4494
17	12 658	5795	5862	5602	5810	5738	5457	4131	4292	4254
18	15 420	7395	7126	6676	7257	7077	6623	4703	4705	4664
19	12 652	6724	6458	6197	6477	6266	5946	4396	4434	4381
20	17 775	8095	8121	7523	8164	8031	7524	5199	5226	5176

overheads once the schedule is built (2ETF case). The improvement of ETFR over 2ETF is more important for second class graphs ($s = 8$) than for first class graphs ($s = 2$). For instance, when $\boldsymbol{L} = 1, \boldsymbol{o} = 10$ and $\boldsymbol{P} = 8$, the improvement is of 4% for first class graphs and of 9% for second class graphs. However, ETFR leads to worse makespans when $\boldsymbol{P} = 4$ especially for first class graphs. In this case, on average two task successors are executed on the same processor as their predecessor and so slots reserved by ETFR are too large. Thus, useless time slots should be removed using ETFRGC algorithm. In general, the application of ETFRGC leads to significantly better results (compared to ETFR) if the number of processors is lower than the average graph width or if the number of task successors is low (cf. graphs 1–10).

7. Concluding remarks

In this paper, we have discussed the scheduling problem under the constraints of LogP model. We have proposed two new heuristics, based on the ETF algorithm, we gave performance bounds for them and also we compared the schedules computed with these algorithms for program graphs with different characteristics. The main difference between LogP and earlier distributed memory models like SDM is that communication incurs overhead in computation processor use. Thus, the problem encountered during construction of program schedules for LogP model is the proper

insertion of receive and send tasks. In the proposed algorithms, we made an assumption that receive tasks directly precede computation. Similarly, all send tasks are executed sequentially right when computation completes. This approach reduces the number of possible task assignments and simplifies the algorithms but possibly eliminates some good solutions. Moreover, we assumed that task starting times are fixed, i.e., they cannot be altered after tasks are scheduled. Thus, we were obliged to leave idle slots in which send tasks were inserted.

The proposed algorithms compute effective schedules if the communication time is not greater than the average computation time. The best results were obtained for ETFRGC algorithm. It is strongly recommended to use this strategy if the number of processors is smaller than the graph parallelism and the number of task successors is low.

Future research should be aimed at finding low complexity algorithms which allow to change task starting times due to insertion of send tasks. Moreover, it would be desirable to compare schedules obtained for program graphs against program execution times on a contemporary machine which preserves properties of the LogP model, like the IBM-SP [13].

References

[1] C. Boeres, Versatile communication cost modelling for multicomputer task scheduling heuristics, Ph.D. Thesis, University of Edinburgh, 1996.
[2] P. Chrétienne, Task scheduling over distributed memory machines, Technical Report 253, MASI, Pierre and Marie Curie University, Paris, 1988.
[3] P. Chrétienne, Complexity of tree-scheduling with interprocessor communication delays, Technical Report 90.5, MASI, Pierre and Marie Curie University, Paris, 1990.
[4] P. Chrétienne, E.G. Coffman Jr., J.K. Lenstra, Z. Liu (Eds.), Scheduling Theory and Its Applications, Wiley, New York, 1995.
[5] D.E. Culler, et al., LogP: a practical model of parallel computation, Communications of the ACM 39 (11), November 1996, 78–85.
[6] D.E. Culler, L.T. Liu, R.P. Martin, C. Yoshikawa, LogP performance assessment of fast network interfaces, IEEE Micro, February 1996.
[7] S. Fortune, J. Wyllie, Parallelism in random access machines, in: Proceedings of the 10th ACM Symposium on Theory of Computing, 1978, pp. 114–118.
[8] P.B. Gibbons, Y. Matias, V. Ramachandran, Can a shared-memory model serve as a bridging model for parallel computation? in: Proceedings of the Nineth Annual ACM Symposium on Parallel Algorithms and Architectures, Newport, Rhode Island, June 22–25, 1997, pp. 72–83, SIGACT/SIGARCH and EATCS.
[9] R.L. Graham, Bounds on multiprocessing timing anomalies, SIAM J. Applied Mathematic 17 (1969) 416–429.
[10] T.C. Hu, Parallel sequencing and assembly line problems, Operational Research 9 (1961) 841–848.
[11] J.J. Hwang, Y.C. Chow, F.D. Anger, C.Y. Lee, Scheduling precedence graphs in systems with interprocessor communication times, SIAM Journal of Computing 18 (2), (1989) 244–257.
[12] K.K. Keeton, T.E. Anderson, D.A. Patterson, Logp quantified: the case for low-overhead local area networks, in: Hot Interconnects III: A Symposium on High Performance Interconnects, Stanford University, Stanford, CA, August 1995, pp. 10–12.
[13] I. Kort, D. Trystram, Assessing LogP model parameters for the IBM-SP, in: EUROPAR'98, Southampton, UK, September 1998.

[14] I. Kort, D. Trystram, Some results on scheduling flat trees in LogP model, Journal of Information Systems and Operational Research (INFOR) (1998), to appear.
[15] T. Lewis, H. El-Rewini, Parallax: A tool for parallel program scheduling, IEEE Parallel and Distributed Technology 1 (2) (1993) 62–72.
[16] W. Löwe, M. Middendorf, W. Zimmermann, Scheduling inverse trees under the communication model of the LogP-Machine, Theoretical Computer Science (1997), to appear.
[17] W. Löwe, W. Zimmermann, J. Eisenbiegler, On linear schedules of task graphs on generalized LogP-Machines, in: Europar'97: Parallel Processing, LNCS 1300, 1997, pp. 895–904.
[18] C.H. Papadimitriou, M. Yannakakis, Towards an architecture-independent analysis of parallel algorithms, SIAM Journal on Computing 19 (2) (1990) 322–328.
[19] V.J. Rayward-Smith, UET scheduling with unit interprocessor communication delays, Discrete Applied Mathematics 18 (1997) 55–71.
[20] B. Shirazi, H.B. Chen, K. Kavi, J. Marquis, A.R. Hurson, PARSA: A parallel program software development tool, in: IEEE CS Press (Ed.), Symposium on Assesment of Quality Software Develompemnt Tools, 1994, pp. 96–111.
[21] L. Valiant, A bridging model for parallel computation, Communication of the ACM 33 (1990) 103–111.
[22] J. Verriet, Scheduling tree-structured programs in the LogP model, Technical Report UU-CS-1997-18, Department of Computer Science, Utrecht University, 1997.
[23] M.Y. Wu, D.D. Gajski, Hypertool: a programming aid for message-passing systems, IEEE Transactions on Parallel and Distributed Systems 1 (3) (1990) 330–343.
[24] T. Yang, A. Gerasoulis, PYRROS: static task scheduling and code generation for message-passing multiprocessors, in: Proceedings of the Sixth ACM International Conference on Supercomputing, Washington, DC, July 1992, pp. 428–437.
[25] W. Zimmermann, W. Loewe, An approach to machine-independent parallel programming, Lecture notes in Computer Science (1994) 854.
[26] W. Zimmermann, M. Middendoff, W. Loewe, On optimal *k*-linear scheduling of tree-like task graphs for Log-Machines, Lecture notes in Computer Science (1998) 1470.