Scheduling Arbitrary Task Graphs on LogP Machines

Cristina Boeres, Aline Nascimento[⋆], and Vinod E.F. Rebello

Instituto de Computação, Universidade Federal Fluminense (UFF)
Niterói, RJ, Brazil
{boeres,aline,vefr}@pgcc.uff.br

Abstract. While the problem of scheduling weighted arbitrary *DAG*s under the *delay model* has been studied extensively, comparatively little work exists for this problem under a more realistic model such as *LogP*. This paper investigates the similarities and differences between task clustering algorithms for the delay and *LogP* models. The principles behind three new algorithms for tackling the scheduling problem under the *LogP* model are described. The quality of the schedules produced by the algorithms are compared with good delay model-based algorithms and a previously existing *LogP* strategy.

1 Introduction

Since tackling the scheduling problem in an efficient manner is imperative for achieving high performance from message-passing parallel computer systems, it continues to be a focus of great attention in the research community. Until recently, the standard communication model in the scheduling community has been the *delay model*, where the sole architectural parameter for the communication system is the *latency* or *delay*, i.e. the transit time for each message word [12]. However, the dominant cost of communication in today's architectures is that of crossing the network boundary. This is a cost which cannot be modelled as a latency and therefore new classes of scheduling heuristics are required to generate efficient schedules for realistic abstractions of today's parallel computers [3]. This has motivated the design of new parallel programming models, e.g. *LogP* [4]. The *LogP model* is an MIMD message-passing model with four architectural parameters: the latency, L; the overhead, o, the CPU penalty incurred by a communication action; the gap, g, a lower bound on the interval between two successive communications; and the number of processors, P.

In this paper, the term *clustering algorithm* refers specifically to the class of algorithms which initially consider each task as a cluster and then merge clusters (tasks) if the completion time can be reduced. A later stage allocates the clusters to processors to generate the final schedule. A number of clustering algorithms

[⋆] The author was supported by a postgraduate scholarship from Fundação de Amparo à Pesquisa do Estado do Rio de Janeiro (FAPERJ) and CAPES, Ministério de Educação e Cultura (MEC), Brazil.

© Springer-Verlag Berlin Heidelberg 1999

exist for scheduling tasks under the delay model, well known examples include: *DSC* [13]; *PY* [12]; and an algorithm by Palis et al. [11] which we denote by *PLW*. It has been proved that these algorithms generate schedules within at least a factor of two of the optimal for arbitrary *DAG*s.

In comparison, identifying scheduling strategies that consider *LogP*-type characteristics is difficult. Only recently have such algorithms appeared in the literature. To the best of our knowledge, all of these algorithms are limited to specific types of *DAG*s or restricted *LogP* models. Kort and Trystram [8] presented an optimal polynomial algorithm for scheduling *fork* graphs on an unbounded number of processors considering $g = o$ and equal sized messages. Zimmerman et al. [14] extended the work of linear clustering [7, 13] to propose a clustering algorithm which produces optimal k-linear schedules for tree-like graphs when $o \geq g$. Boeres and Rebello [2, 3] proposed a task replication strategy for scheduling arbitrary UET-UDC (unit execution tasks, unit data communication) *DAG*s on machines with a bounded number of processors under both the *LogP* (also when $o \geq g$) and *BSP* models.

The purpose of this work is to study, from first principles, the problem of scheduling arbitrary task graphs under the *LogP* model using task replication-based clustering algorithms, by generalising the principles and assumptions used by existing delay model-based algorithms. The following section introduces the terminology and definitions used throughout the paper and outlines the influence of the *LogP* parameters on the scheduling problem. The issues that need to be addressed in the design of a clustering-based scheduling algorithm for the *LogP* model are discussed in Sect. 3. In Sect. 4, three replication-based task clustering algorithms are briefly presented for scheduling general arbitrary weighted *DAG*s on *LogP* machines (although an unbounded number of processors is assumed). Section 5 compares the makespans produced by the new algorithms against other well known clustering algorithms under both delay and *LogP* model conditions. Section 6 draws some conclusions and outlines our future work.

2 Scheduling under the Delay and *LogP* Models

A parallel application is often represented by a *directed acyclic graph* or *DAG* $G = (V, E, \varepsilon, \omega)$, where: the set of vertices, V, represent *tasks*; E the precedence relation among them; $\varepsilon(v_i)$ is the execution cost of task $v_i \in V$; and $\omega(v_i, v_j)$ is the weight associated to the edge $(v_i, v_j) \in E$ representing the amount of data units transmitted from task v_i to v_j. Two dependent tasks (i.e. tasks v_i and v_j for which $\exists(v_i, v_j) \in E$) are often grouped together to avoid incurring the cost of communicating. If a task has a successor in more than one cluster it may be advantageous to *duplicate* the ancestor task and place a copy of it in each of the clusters containing the successor. In any case, with or without task duplication, the task clustering problem is NP-hard [12].

This paper associates independent overhead parameters with the *sending* and *receiving* of a message, denoted by λ_s and λ_r, respectively (i.e. $o = \frac{\lambda_s + \lambda_r}{2}$). For the duration of each overhead the processor is effectively *blocked* unable to execute other tasks in V or even to send or receive other messages. Consequently, any scheduling algorithm must, in some sense, view these sending and receiving overheads also as "tasks" to be executed by processors. The rate of executing communication events may be further slowed due to the network (or the network interface) capacity modeled by the gap g. The execution order of the overhead tasks is important since two or more messages can no longer be sent simultaneously as in the delay model. Throughout this paper, the term *delay model conditions* is used to refer to situation where the parameters λ_s, λ_r and g are zero and the term *LogP conditions* used when otherwise.

In [12], the *earliest start time* of a task v_i, $e(v_i)$, is defined as a *lower bound* on the optimal or earliest possible time that task v_i can start executing. As such it is not always possible to schedule task v_i at this time [12]. In order to avoid confusion, we introduce the term $e_s(v_i)$, the *earliest schedule time*, to be the earliest time that task v_i can start executing on any processor. Thus, the completion time or *makespan* of the optimal schedule for G is $\max_{v_i \in V}\{e_s(v_i)+\varepsilon(v_i)\}$.

Many scheduling heuristics prioritise each task in the graph according to some function of the perceived computation and communication costs along all paths to that task. The costliest path to a task v_i is often referred to as the *critical path* of task v_i. Possible candidate functions include both $e(v_i)$ and $e_s(v_i)$. One must be aware that the critical path of a task v_i may differ depending on the cost function chosen. Scheduling heuristics which use some form of critical path analysis need to define how to interpret and incorporate the *LogP* parameters when calculating the costs of paths in the *DAG*. One of the difficulties being that these are communication costs paid for by computation time.

The *LogP* model is not an extension of the delay model but rather a generalisation, i.e. the delay model is a specific instance of the *LogP* one. Therefore, it is imperative to understand the underlying principles and identify the assumptions which can be adopted in *LogP* clustering algorithms.

3 Cluster-Based Scheduling for the *LogP* Model

Clustering algorithms typically employ two stages: the first determines which tasks should be included and their order within a cluster; the second, identifies the clusters required to implement the input *DAG* G mapping each to a processor. These algorithms aim to obtain an optimal schedule for a *DAG* G by attempting to construct a cluster $C(v_i)$ for each task v_i so that v_i starts executing at the earliest time possible (i.e. $e_s(v_i)$). In algorithms which exploit task duplication to achieve better makespans, $C(v_i)\ \forall v_i \in V$ will contain the *owner* task v_i and, determined by some cost function, *copies* of some of its ancestor tasks.

The iterative nature of creating a cluster can be generalised to the form shown in Algorithm 1. In lines 2 and 5, the list of candidate tasks (*lcands*) for inclusion into a cluster consists of those tasks which become immediate ancestors of the cluster, i.e. $iancs(C(v_i)) = \{u \mid (u,t) \in E \wedge u \notin C(v_i) \wedge t \in C(v_i)\}$.

Algorithm 1 : *mechanics-of-clustering* $(C(v_i))$;

```
1 $C(v_i) = \{v_i\}$;      /* Algorithm to create a cluster for task $v_i$ */
2 $lcands = iancs(C(v_i))$;
3 while $progress\ condition$ do
4    let $cand \in lcands$;  $C(v_i) = C(v_i) \cup \{cand\}$;
5    $lcands = iancs(C(v_i))$;
```

The goodness of a clustering algorithm (and the quality of the results) depend on four crucial design issues: (i) calculating the makespan; (ii) the ordering of tasks within the cluster; (iii) the *progress condition* (line 3); and (iv) the choice of candidate task for inclusion into the cluster (line 4).

The makespan of a cluster $C(v_i)$ can be determined by the costliest path to task v_i (critical path). Effectively, there are only two types of path: the path due to *computation* in the cluster; and the paths formed by the immediate ancestor tasks of the cluster. The *cluster path cost*, $m(C(v_i))$, is the earliest time that v_i can start due to cluster task computation, including receive overheads but ignoring any idle periods where a processor would have to wait for the data sent by tasks $\in iancs(C(v_i))$ to arrive at $C(v_i)$. The order in which the task are executed in the cluster has a significant influence on this cost (finding the best ordering is a NP-complete problem [6]). The *ancestor path cost*, $c(u, t, v_i)$ is the earliest time that v_i can start due solely to the path from task u, an immediate ancestor of task $t \in C(v_i)$, to v_i. The ancestor path with the largest cost is known as the *critical ancestor path*

The *progress condition* determines when the cluster is complete. Commonly this condition is often derived from a comparison of the makespan before and after the inclusion of the chosen candidate task. This can be simplified to compare the two critical paths if an appropriate cost function is used. A candidate task is added to the cluster only if its inclusion does not increase the makespan. Otherwise, the creation process stops since the algorithm assumes that the makespan can be reduced no further, i.e. that the makespan is a monotonically decreasing function, up to its global minima, with respect to the inserted tasks. This assumption is based on the fourth issue – every choice of candidate has to be the best one. The list *lcands* contains the best candidate since the inclusion of a non-immediate ancestor can only increase the makespan of the cluster. But which of the tasks is the best choice?

Under the delay model, the inclusion of a task has two effects: it will increase the cluster path cost at the expense of removing the critical ancestor path; and introduce a new ancestor path for each immediate ancestor of the inserted task not already in the cluster. Because the cluster path cost increases monotonically under this model, the progression condition can be simplified to a combination of the following two conditions, the algorithm is allowed to progress while: the cluster path is not the critical path (Non-Critical Path Detection (NCPD)); and inserting the chosen task does not increase the makespan (Worth-While Detection (WWD)). The makespan of a cluster can only be reduced if the cost of the critical path is reduced. If the critical ancestor path is the critical path of the cluster, the critical ancestor task u is the only candidate whose inclusion into the cluster has the potential to diminish the makespan. While selecting this task is on the whole a good choice, it is not always the best. The critical path cost could increase the makespan sufficiently to cause the algorithm to stop prematurely, at a local optimum for the makespan, because the inserted task u has a high communication cost with one of its ancestor (a cost not reflected by the chosen cost function value for u).

The effects of including a candidate task under *LogP* conditions are similar except that the cluster path cost may not necessarily increase. Communication overheads need to be treated like tasks since they are costs incurred on a processor even though they are associated with ancestor paths. This means that an immediate ancestor path can affect the cluster path cost and the cost of the other ancestor paths of the cluster. Not only can the inclusion of a non-critical ancestor path task reduce the makespan but improvements to the makespan may be achieved by progressing beyond the NCPD point through the removal of the receive overheads incurred in the cluster. In addition to dependencies between program tasks, the problem of ordering tasks within a cluster now has to address the dependencies of these overhead "tasks" to their program tasks and their network restrictions (i.e. the effects of parameter g).

All in all, the quality of makespans produced by a clustering algorithm depends on the quality of the implementation of each of the above design issues. These issue are not independent, but interrelated by the multiple role played by the cost function. How paths are costed is important since the same cost function is often used: to calculate the makespan and thus affecting the progress condition; and to order tasks within a cluster. Furthermore, the cost function may also be the basis for choosing a candidate task. If a design issue cannot be addressed perfectly, the implementation of the other issues could try to compensate for it. For example, if the best candidate is not always chosen, the makespan will not be a monotonically decreasing function. Therefore, progress condition must be designed in such way that will allow the algorithm to escape local minima but not to execute unnecessarily creating a poor cluster.

3.1 Adopted Properties for *LogP* Scheduling

In order to schedule tasks under a *LogP*-type model, we propose that clustering algorithms apply some *scheduling restrictions* with regard to the properties of clusters. In the first restriction, only the owner task of a cluster may send data to a task in another cluster. This restriction does not adversely affect the makespan since, by definition of $e_s(v_i)$, an owner task will not start execution later than any copy of itself. Also, if a non-owner task t in cluster $C(v_i)$ were to communicate with a task in another cluster, the processor to which $C(v_i)$ is allocated would incur a send overhead after the execution of t which in turn might force the owner v_i to start at a time later than its earliest possible.

The second restriction specifies that each cluster can have only one successor. If two or more clusters, or two or more tasks in the same cluster, share the same ancestor cluster then a unique copy of the ancestor cluster is assigned to each successor irrespective of whether or not the data being sent to each successor is identical. This removes the need to incur the multiple send gap costs and send overheads which may unnecessarily delay the execution of successor clusters.

The third restriction, currently a temporary one, assumes that receive overheads are executed immediately before the receiving task even though they have to be scheduled at intervals of atleast g. Future work will investigate the merits of treating these overheads just like program tasks and allowing them to be scheduled in any valid order within the cluster. The fourth restriction is related to the third and to the behaviour of the network interface. This restriction assumes that the receive overheads, scheduled before their receiving task, are ordered by the arrival time of their respective messages. It is assumed that the network interface implements a queue to receive messages.

These restrictions are new in the sense that they do not need to be applied by scheduling strategies which are used exclusively under delay model conditions. The purpose of these restrictions is to aid in the minimisation of the makespan, though they do incur the penalty of increasing the number of clusters required and thus the number of processors needed. Where the number of processors used is viewed as a reflection on the cost of implementing the schedule, post-pass optimisation can be applied in a following stage of the algorithm to relax the restrictions and remove clusters which become redundant (i.e. clusters whose removal will not increase the makespan).

4 Three New Scheduling Algorithms

This section discusses the design of three clustering algorithms, focusing on the cluster creation stage. These algorithms are fundamentally identical but differ in the manner they address the design issues discussed in Sect. 3. The second stages of these algorithms, which are identical, determine which of the generated clusters are needed to implement the schedule.

Unlike traditional delay model approaches which tend to use the earliest start time $e(v_i)$ as the basis for the cost function, the *LogP* algorithms presented here use the earliest schedule time $e_s(v_i)$. Using $e_s(v_i)$ should give better quality results since this cost represents what is achievable (remember that $e(v_i)$ is a lower bound and that cost is an important factor in the mechanics of cluster algorithms). On the other hand, since finding $e_s(v_i)$ is much more difficult, these algorithms in fact create the cluster $C(v_i)$ to find a good approximation, $s(v_i)$, to $e_s(v_i)$ which is then used to create the clusters of task v_i's successors. $s(v_i)$, therefore, is the *scheduled time* or *start time* of task v_i in its own cluster $C(v_i)$.

In the first algorithm, $BNR2$ [10], tasks are not ordered in non-decreasing order of their earliest start time (as in most delay model approaches) nor by their scheduled time but rather in an order determined by the sequence in which tasks were included in the cluster and the position of their immediate successor task at the time of their inclusion. The *ancestor path cost*, $c(u, t, v_i)$, for immediate ancestor task u of task t in $C(v_i)$ is the sum of: the execution cost of $C(u)$

$(s(u) + \varepsilon(u))$; one sending overhead (λ_s) (due to the second *LogP* scheduling restriction); the communication delay ($\tau \times \omega(u,t)$); the receiving overhead (λ_r); and the computation time of tasks t to v_i, which includes receive overheads of edges arriving after the edge (u,t) and takes into account g and restrictions 3 and 4. The critical ancestor task is the task $u \in iancs(C(v_i))$ for which $c(u,t,v_i)$ is the largest.

If critical ancestor path cost, $c(u,t,v_i)$, is greater than the cluster path cost then task u becomes the chosen candidate for inclusion into cluster $C(v_i)$ (the NCPD point of the progress condition). Before committing u to $C(v_i)$, $BNR2$ compares this critical ancestor path cost with the cluster path cost of $\{C(v_i) \cup \{u\}\}$ (the WWD point of the progress condition). Note this detection point does not actually check for an non-increasing makespan. This less strict condition allows the algorithm to proceed even if the makespan increases, permitting $BNR2$ to escape some local makespan minimas as long as the NCPD condition holds. One drawback is that the algorithm can end up generating a worse final makespan. A solution might be to keep a record of the best makespan found so far [10].

The second algorithm, $BNR2_s$, is a simplification of the previous algorithm. Only candidates from the set of *immediate ancestors* of task v_i are considered for inclusion into $C(v_i)$. But the inclusion of an ancestor task u implies that all tasks in $C(u)$ should be included in $C(v_i)$. Also, the immediate ancestor clusters $C(w)$ of $C(u)$ become immediate ancestors of cluster $C(v_i)$ unless owner task w is already in $C(v_i)$. The mechanics of the algorithm are almost the same as $BNR2$: the candidate chosen is still the task on the critical ancestor path; however, the progress condition compares the two makespans (before and after the candidate's inclusion); and the tasks in the cluster are ordered in nondecreasing order of their scheduled time.

The third algorithm, $BNR2_h$, is a hybrid of the first two. As in $BNR2_s$, the cluster tasks are ordered in nondecreasing order of their scheduled time and the critical ancestor path task is the chosen candidate. Initially the algorithm behaves like $BNR2_s$, inserting into $C(v_i)$ copies of all tasks in $C(u)$ (u being the first chosen immediate ancestor) and inheriting copies of the immediate ancestor clusters $C(w)$ of $C(u)$. After this cluster insertion, the algorithm's behaviour switches to one similar to $BNR2$ where the cluster is now grown a task at a time. Note that the initial inherited ancestor tasks w are never considered as candidates for inclusion. The main difference from $BNR2_h$ is the progress condition. This algorithm is allowed to continue while the *weight* of the cluster (this being the sum of the weights of the program tasks in the cluster) is less than both its makespan and best makespan found so far. This is a monotonically increasing lower bound on the makespan of $C(v_i)$. The progress condition thus allows the algorithm to speculatively consider possible worthwhile cluster schedules even though the makespan may increase. When the algorithm stops, the best makespan found will be the global minima (given the tasks inserted and ordering adopted). This is not necessarily the optimal makespan since this depends on addressing the design issues (outlined in Sect. 3) perfectly.

5 Results

The results produced by the three algorithms have been compared with the clustering heuristics *PLW* and *DSC* under delay model conditions, and *MSA* under *LogP* conditions using a benchmark suite of UET-UDC *DAG*s which includes out-trees (*Ot*), in-trees (*It*), diamond *DAG*s (*D*) and a set of randomly generated *DAG*s (*Ra*) as well as irregular weighted *DAG*s (*Ir*) taken from the literature. A sample of the results are presented in the following tables, where the number of tasks in the respective graphs is shown as a subscript. For fairness, the value of the makespan M is the execution time for the schedule obtained from a *parallel machine simulator* rather than the value calculated by the heuristics. The number of processors required by the schedule is P. C, the number of times the progression condition is checked, gives an idea of the cost to create the schedule.

Although the scheduling restrictions cause the *LogP* strategies to appear greedy with respect to the number of processors required, it is important to show that the makespans produced and processors used are comparable to those of delay model approaches under their model (Table 1). Scheduling approaches such as *DSC* and *PLW* have already been shown to generate good results (optimal or near-optimal) for various types of *DAG*s. Table 2 compares the makespans generated for UET-UDC *DAG*s with those of *MSA* (assuming $\lambda_r \geq g$).

Table 1. Results for a variety of graphs under the delay model.

		DSC		PLW		MSA		$BNR2$			$BNR2_s$			$BNR2_h$		
DAG	τ	P	M	P	M	P	M	P	M	C	P	M	C	P	M	C
Ir_{41} [7]	1	13	16	22	16	10	16	10	15	120	17	15	67	16	15	71
Ir_{41}	2	9	21	20	22	10	18	13	17	283	12	18	68	13	17	82
Ir_{41}	4	8	31	19	27	10	23	11	22	305	9	24	67	12	21	93
Ir_{41}	8	3	44	19	39	5	27	10	27	381	5	30	69	9	26	98
Ra_{80}	4	27	25	41	31	48	22	31	18	294	55	19	111	57	19	125
Ra_{186}	4	66	22	87	19	52	19	91	14	642	117	14	225	123	14	233
It_{511}	4	98	39	167	27	85	25	134	25	1357	48	37	510	85	28	797
Ot_{511}	4	146	27	264	13	256	9	256	9	4096	256	9	510	256	9	510
Di_{400}	4	27	146	165	140	37	116	31	98	4983	289	112	760	267	104	736

	DSC		PLW		$BNR2$		$BNR2_s$		$BNR2_h$	
DAG	P	M	P	M	P	M	P	M	P	M
Ir_{7a} [5]	2	9	3	8	3	8	3	8	3	8
Ir_{7b} [13]	3	8	5	12	3	8	3	8.5	3	8
Ir_{10} [11]	3	30	4	27	3	26	4	26	4	26
Ir_{13} [1]	7	301	8	275	8	246	9	246	9	246
Ir_{18} [9]	5	550†	8	480	9	370	8	400	9	380

†[9] reports a makespan of 460 on 6 processors.

The results produced by the three cluster algorithms $BNR2$, $BNR2_s$ and $BNR2_h$ are, in the majority of cases, better than those of DSC, PLW and MSA under their respective models. $BNR2_s$ on the whole produces results only slightly worse than the other two in spite of its simplicitic approach (which is reflected by its C value). Under the delay model, we also analysed the effect of task ordering on the makespans produced by $BNR2_s$ and $BNR2_h$. Only in two cases for $BNR2_h$, does optimising the order improve the makespan (becoming 25 for It_{511} and 370 for Ir_{18}). The ease by which $BNR2_s$ and $BNR2_h$ find

Table 2. Results considering various LogP parameters values.

				MSA		$BNR2$			$BNR2_s$			$BNR2_h$						MSA		$BNR2$			$BNR2_s$			$BNR2_h$		
DAG	λ_s	λ_r	τ	Prs	M	Prs	M	C	Prs	M	C	Prs	M	C	λ_s	λ_r	τ	Prs	M	Prs	M	C	Prs	M	C	Prs	M	C
Ir_{41}	1	1	1	8	30	34	24	225	14	23	68	15	23	114	2	1	8	6	37	11	32	396	4	32	69	4	32	110
Ir_{41}	1	1	2	10	32	18	24	313	14	25	67	14	25	122	2	2	1	6	35	21	32	267	11	30	67	11	30	142
Ir_{41}	1	1	4	3	33	12	27	332	9	29	68	11	28	116	2	2	2	6	36	15	32	300	11	31	67	9	31	132
Ir_{41}	1	1	8	5	36	11	31	394	4	32	69	5	31	112	4	0	1	6	27	14	23	354	12	25	68	15	24	86
Ir_{41}	2	0	1	10	25	31	19	276	14	21	69	14	21	86	4	0	2	6	28	16	24	344	9	27	69	15	25	97
Ir_{41}	2	0	2	6	25	26	22	305	14	24	67	13	22	93	4	0	4	6	30	14	27	381	5	30	69	7	29	98
Ir_{41}	2	0	4	6	26	16	24	344	9	27	68	15	25	97	4	0	8	6	34	10	31	374	3	33	69	4	33	108
Ir_{41}	2	0	8	6	30	10	29	368	4	31	69	5	31	104	4	1	1	6	34	12	27	332	9	29	68	11	28	116
Ir_{41}	2	1	1	10	33	18	24	313	14	25	67	14	25	122	4	1	2	6	34	13	29	335	8	31	69	6	28	119
Ir_{41}	2	1	2	10	34	15	25	326	11	27	67	14	27	135	4	1	4	5	35	12	30	384	4	32	69	5	30	110
Ir_{41}	2	1	4	6	34	13	29	335	8	31	69	6	28	119	4	4	1	1	41	15	49	316	2	40	69	2	40	113

				MSA		$BNR2$			$BNR2_s$			$BNR2_h$		
DAG	λ_s	λ_r	τ	P	M	P	M	C	P	M	C	P	M	C
Ra_{80}	4	2	2	25	56	146	46	238	150	60	121	113	60	329
Ra_{186}	4	2	2	78	47	158	36	660	134	38	232	102	38	416
It_{511}	4	2	2	85	45	85	45	1545	30	58	510	64	46	1106
Ot_{511}	4	2	2	256	9	256	9	4096	256	9	510	256	9	510

optimal schedules for out-trees is reflected by closeness of C to the number of tasks in the graph.

6 Conclusions and Future Work

We present three clustering algorithms for scheduling arbitrary task graphs with arbitrary costs, under the *LogP* model, onto an unbounded number of processors. To the best of our knowledge, no other general *LogP* scheduling approach exists in the literature. This work is not based on extending an existing delay model algorithm to a *LogP* one. Instead, these algorithms were design from first principles to gain a better understanding of the mechanics of *LogP* scheduling. Fundamental differences exist between these algorithms and their nearest delay model relatives (e.g. *PLW*), due in part to interaction of the *LogP* model parameters on the design issues and to the assumptions used to simplify the problem under the delay model.

Based on the results obtained so far, initial versions of these algorithms compare favourably against traditional clustering-based scheduling heuristics such as *DSC* and *PLW* which are dedicated exclusively to the delay model. For UET-UDC *DAG*s, all three algorithms perform surprising well compared to *MSA* which exploits the benefits of bundling messages to reduce the number of overheads incurred. Results of further experiments using graphs with a more varied range of granularities and connectivities are needed to complete the practical evaluation of these algorithms.

Future work will primarily focus on studying the effects of relaxing the scheduling restrictions and of various design issue decisions e.g. on what basis should program and overhead tasks be ordered within a cluster [10], are there alternative cost functions better able to choose candidate tasks?

References

[1] I. Ahmad and Y-K Kwok. A new approach to scheduling parallel programs using task duplication. In K.C. Tai, editor, *International Conference on Parallel Processing*, volume 2, pages 47–51, Aug 1994.

[2] C. Boeres and V. E. F. Rebello. A versatile cost modelling approach for multicomputer task scheduling. *Parallel Computing*, 25(1):63–86, 1999.

[3] C. Boeres, V.E.F. Rebello, and D. Skillicorn. Static scheduling using task replication for LogP and BSP models. In D. Pritchard and J. Reeve, editors, *The Proceedings of the 4th International Euro-Par Conference on Parallel Processing (Euro-Par'98)*, LNCS 1470, pages 337–346, Southampton, UK, September 1998. Springer-Verlag.

[4] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[5] A. Gerasoulis and T. Yang. A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors. *Journal of Parallel and Distributed Computing*, 16:276–291, 1992.

[6] A. Gerasoulis and T. Yang. List scheduling with and without communication. *Parallel Computing*, 19:1321–1344, 1993.

[7] S.J. Kim and J.C. Browne. A general approach to mapping of parallel computations upon multiprocessor architectures. In *Proceedings of the 3rd International Conference on Parallel Processing*, pages 1–8, 1988.

[8] I. Kort and D. Trystram. Scheduling fork graphs under LogP with an unbounded number of processors. In D. Pritchard and J. Reeve, editors, *The Proceedings of the 4th International Euro-Par Conference on Parallel Processing (Euro-Par'98)*, LNCS 1470, pages 940–943, Southampton, UK, September 1998. Springer-Verlag.

[9] Y. K. Kwok and I. Ahmad. Dynamic critical-path scheduling: An effective technique for allocating tasks graphs to multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(5):505–521, May 1996.

[10] A. Nascimento. Aglomeração de tarefas em arquiteturas paralelas com memória distribuída. Master's thesis, Instituto de Computação, Universidade Federal Fluminense, Brazil, 1999. (In Portuguese).

[11] M.A. Palis, J.-C Liou, and D.S.L. Wei. Task clustering and scheduling for distributed memory parallel architectures. *IEEE Transactions on Parallel and Distributed Systems*, 7(1):46–55, January 1996.

[12] C.H. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. *SIAM J. Comput.*, 19:322–328, 1990.

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

[14] W. Zimmermann, M. Middendorf, and W. Lowe. On optimal k-linear scheduling of tree-like task graph on LogP-machines. In D. Pritchard and J. Reeve, editors, *The Proceedings of the 4th International Euro-Par Conference on Parallel Processing (Euro-Par'98)*, LNCS 1470, pages 328–336, Southampton, UK, September 1998. Springer-Verlag.