On Minimising the Processor Requirements of LogP Schedules

Cristina Boeres*, Gerson N. da Cunha, and Vinod E. F. Rebello**

Instituto de Computação, Universidade Federal Fluminense (UFF)
Niterói, RJ, Brazil
{boeres,gerson,vinod}@ic.uff.br

Abstract. This paper briefly describes the mechanisms used to reduce the number of processors required by a class of task replication-based scheduling heuristics for the *LogP* model. These heuristics, which are based on the task clustering algorithm design methodology [3], are known to generate schedules with good makespans [3,5]. Results in this paper show that a significant reduction, on average, in the number of processors required can be obtained without degrading the makespan. These mechanisms can also be used to tradeoff *quality* in terms of an increase in the makespan for *quantity* in terms of a further reduction in the number of processors required.

1 Introduction

The principal objective of task scheduling is to find a schedule with the minimal makespan, for a given program representation and communication model on bounded or unbounded number of processors. A secondary objective, though just as important, is minimising the cost of implementing this makespan.

The standard communication model used in the task scheduling problem is the *delay model*, where the *latency* or *delay* for a message transmission is the only communication parameter considered relevant [12]. However, with the advances in technology, the evolution of parallel architectures, and the use of software-based communication libraries to support portability, other communication characteristics have been shown to influence performance significantly [10].

A well-known and now established abstract model of parallel computation, the *LogP* model, has attemped to address this fact [7]. The principal feature encompassed in this model is that inter-processor communication also requires processing time by the communicating processors. This model identified four architectural parameters: L is the *latency* incurred when transmitting a message; o, the processing *overhead* incurred when sending or receiving (during which time the processor cannot execute tasks nor send or receive other messages); g, the

* This work is funded by grants from the the Fundação de Amparo à Pesquisa do Estado do Rio de Janeiro (FAPERJ). The authors are partially supported by grants from Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq).

** On leave from the Universidade Católica de Petrópolis, Petrópolis, RJ, Brazil.

R. Sakellariou et al. (Eds.): Euro-Par 2001, LNCS 2150, pp. 156–166, 2001.

minimum *gap* between two consecutive message sends or receives by a processor; and P, the number of processors available. These parameters also capture the fact that networks have a limited capacity for message transmissions. While recent, more accurate models have also been proposed [1,13], these appear to be refinements of the $LogP$ model rather than radically new models. These $LogP$-type models succeed in modeling a number of parallel machines [1,7,9,10,13].

Since characteristics such as the processing cost to send and receive messages cannot be modeled as part of the message delay [7], poor program performances may result from a schedule based on an inaccurate representation of the target machine [4]. In comparison to the number of scheduling heuristics devised for the delay model, few scheduling strategies based on communication models that specify $LogP$-type characteristics have been proposed [2,4,6,8,9]. Furthermore, these algorithms tend to be limited to specific classes or types of graphs (with the exception of [8]) and restricted $LogP$ models (with the exception of [9]). One of the difficulties when handling the $LogP$ parameters is that the overheads are communication costs paid for by computation time.

Clustering or grouping tasks together on a processor reduces communication costs. If the successors of a task have been assigned to different processors, it may be advantageous to *replicate* this task on each of the processors containing a successor. Under the $LogP$ model, task replication may still be worthwhile even if the copy is not scheduled on the same processor as its successor. Replicating a task which needs to send more than one message avoids the delays (o or g) that would be incurred by the subsequent messages sent by this task.

In [5], a number of task replication-based clustering algorithms were proposed based on the task clustering algorithm design methodology proposed in [2,3]. These algorithms schedule arbitrary task graphs on an unbounded number of processors under an unrestricted $LogP$ model and generate schedules whose makespan compare favourably with those produced by existing scheduling algorithms [2,3,5]. The methodology, like most cluster algorithms [11,14], consists of two stages. The *first stage* tries to create a cluster for each task (the task becomes the *owner* of the cluster), made up of copies of ancestor tasks which allow the owner to be scheduled as early as possible. The *second stage* determines the number of copies of each cluster necessary to implement a schedule with the makespan found earlier.

In order to guide the construction process of each cluster, the methodology enforces four cluster design restrictions during the first stage [3]. *Restriction 1* (R1) prevents the owner from being delayed by overheads due to message sends to other clusters. *Restriction 2* (R2) removes the problem of ordering the send overheads by restricting each cluster to sending only one message. *Restriction 3* (R3) forces the receive overheads within the cluster to be scheduled as late as possible, while *Restriction 4* (R4) assumes that these overheads are ordered by the arrival time of their respective messages. While the first two restrictions aid in the process of minimizing the schedule time of each owner, the number of clusters (and thus processors) required to implement this *virtual schedule* (i.e. the schedule obtained by abiding to the restrictions) tends to be extremely

high. This problem could be alleviated if the second stage were to carefully relax R1 and/or R2 and remove the copies of clusters which become redundant.

Most scheduling heuristics try to find the smallest makespan by calculating the earliest time that tasks (in the case of clustering algorithms, the owner tasks) can be scheduled. In *LogP* clustering algorithms, the delaying of both tasks and the transmission of messages can be used to produce time intervals or windows of available processing time in which additional tasks could be included into a cluster, or where more than one send overhead could be scheduled, without compromising the makespan of the virtual schedule.

This work aims to re-design the second stage of the task clustering algorithm design methodology by proposing a series of mechanisms to decrease the number of clusters required by the final schedule without, in the first instance, increasing the makespan. These same mechanisms can be used to trade-off (i.e. decrease further the number of) processors for an increase in the makespan. In this paper, Restriction 2 (R2) is relaxed to permit an owner to send various messages if this eliminates at least some of the copies of the owner's cluster.

The following section presents some terminology used by design methodology. Section 3 defines the concept of processing windows and describes three mechanisms used to increase their size. How these windows are used to reduce the number of clusters required is also discussed. Section 4 summarises the results obtained while Sect. 5 presents some conclusions.

2 Terminology

In task scheduling, a parallel application is often represented by a *directed acyclic graph* or *DAG* $G = (V, E, \varepsilon, \omega)$, where: the set of vertices, V, represents the *tasks*; E, their precedence relationships; $\varepsilon(v_i)$ is the execution cost of task $v_i \in V$; and $\omega(v_i, v_j)$, the amount of data transmitted from task v_i to v_j.

The design methodology associates independent overhead parameters with the *sending* and *receiving* of a message, denoted by o_s and o_r, respectively and separate parameters for the gap between successive message sends (g_s) and receives (g_r) as proposed in [13]. Also, the latency parameter L is treated as the transmission time per unit data.

The emphasis of previous scheduling algorithms based on this two-stage design methodology has been on the design and implementation of the first stage [5]. This stage attempts to create a cluster $C(v)$ for each $v \in V$ (subject to the cluster design restrictions [3]) so that the *schedule time* of v, denoted by $s(v)$, is the earliest possible. Due to task replication, only a subset (the set of *useful clusters*, *uc*) of the $|V|$ clusters created will be utilized to form the set of *necessary clusters*, i.e. the number of copies of each cluster $\in$ *uc* (due to R1 and R2) necessary to implement the virtual schedule of G with makespan $\mathcal{M} = \max_{v \in V}\{s(v) + \varepsilon(v)\}$. The second stage starts with the *sink clusters* (clusters whose owners have no successors $\in V$) and visits useful immediate ancestor clusters in reverse topological order. Cluster $C(w)$ is an immediate ancestor of $C(v)$ if owner w is an immediate ancestor of a task in $C(v)$ and $w \notin C(v)$.

3 Relaxing Restriction 2

Even though, the first stage creates $|V|$ clusters, one for each task in G, only $|uc|$ of these clusters will be used. However, due to the replication of clusters, $|nc|$ *necessary clusters* (where $|nc| \geq |uc|$) are required to implement the virtual schedule with makespan $\mathcal{M}$. The mechanisms proposed in this paper study the effects of relaxing restriction R2 in order to reduce $|nc|$ and thus implement the final schedule with a *reduced set of clusters*, *rsc*.

These mechanisms basically attempt to define a large enough *processing window*, for the clusters $\in nc$, within which the send overheads of more than one message can be scheduled (subject to R1 but not R2) without increasing the makespan. This work extends the idea described in Example 1 and Fig. 1 (a).

Example 1: Assume cluster $C(w)$ has been constructed by the first stage according to the cluster design restrictions and that message (w, u_i) sent by the owner w in cluster $C'(w)$, a copy of cluster $C(w)$, to task u_i in $C(v)$ arrives t time units *earlier* than the *schedule time* of the associated receiving overhead. It would be possible therefore to delay the sending of that message from $C'(w)$ by t time units, creating a *processing window* of size t between the end of the execution of $w \in C'(w)$ and the sending overhead. If t is large enough (i.e. $t \geq \max\{o_s, g_s\}$) this message could also be sent by $C(w)$ since the o_s for (w, u_i) can be scheduled after the o_s of $C(w)$'s message without delaying $C(v)$, thus removing the need for cluster $C'(w)$. The removal of a cluster also eliminates the need for all of its ancestor clusters. These gains are made possible, in part, by R3 which schedules the receive overheads as late as possible within each cluster.

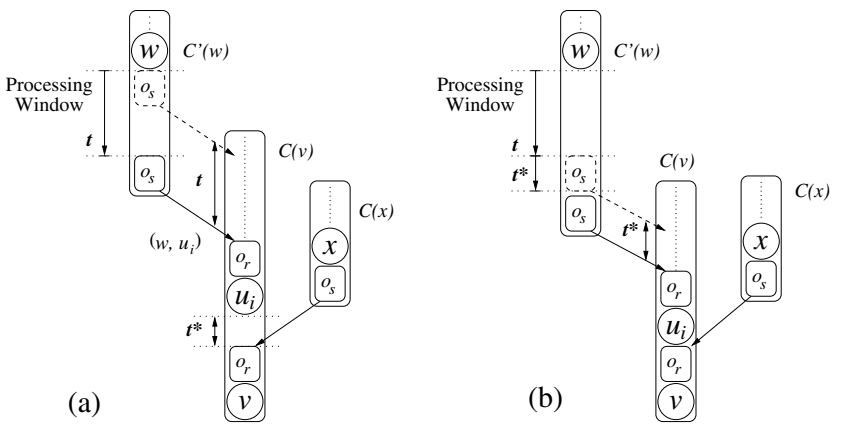

Fig. 1. Both, (a) Example 1: the delaying of message (w, u_i) and (b) Mechanism One: the delaying of task u_i, can be used to increase the processing window of (w, u_i)

Based on this idea, the number of clusters can be reduced by rescheduling communications. However, further gains could be attained by delaying clusters since not all of the owners v need to be scheduled at their $s(v)$ to achieve the makespan $\mathcal{M}$. This section discusses 3 mechanisms used to increase the process-

ing window of $C(w)$, through a combination of delaying tasks and rescheduling communications. Mechanism One delays the non-owner tasks within the cluster $C(v) \in uc$. Mechanism Two reorders message receives within the successors $C'(v) \in rsc$ of $C(w)$. Once the sizes of the processing windows for $C(w)$ have been identified, the messages sent by w are rescheduled and a sufficient number of copies $C'(w)$ of $C(w)$ is defined. Mechanism Three tries to delay the $C'(w)$s in rcs. This process is then repeated for the next reverse-topological cluster $\in uc$.

Defining a Processing Window: A message (w, u_i) may not necessarily be processed immediately on arrival at its destination cluster $C(v)$ due to the execution of other tasks or receive overheads in that cluster. The mechanisms aim to determine, for each cluster $\in nc$, the earliest and latest time that its message can be sent, thus defining a processing window for that message. The earliest message send time of (w, u_i), from the first stage, is $emst(w, u_i, v) = s(w) + \varepsilon(w)$. The latest message send time $lmst(w, u_i, v)$ depends on the latest time that the receive overhead of (w, u_i) can be processed in $C(v)$. The period of time $[emst(w, u_i, v), lmst(w, u_i, v)]$ is the interval during which message (w, u_i) can be sent to $C(v)$ without delaying task v and whose duration defines the processing window $pw(w, u_i, v)$.

The schedule time of non-owner task $u_i \in C(v)$, denoted by $st(u_i, v)$, is the earliest time that task u_i can be scheduled in cluster $C(v)$. Considering Example 1, where only the sending of messages are delayed and not the tasks, $lmst(w, u_i, v) = st(u_i, v) - (o_s + L \times \omega(w, u_i) + n \times o_r)$ when message (w, u_i) is the nth last of all the messages to arrive at $u_i \in C(v)$.

Mechanism One: Delaying Non-owner Tasks. This mechanism explores the possibility of delaying the execution of tasks within clusters $\in uc$ so that messages being sent to them can arrive even later. As shown in Fig. 1 (b), in order to increase the processing window of immediate ancestor clusters, only non-owner tasks will be delayed.

The *delayed schedule time* of a non-owner task u_i in a cluster $C(v)$, denoted by $dst(u_i, v)$, is the latest time that u_i can be scheduled without delaying v. For the last non-owner task in $C(v) = \langle u_1, u_2, \ldots, u_l, v\rangle$, $dst(u_l, v) = s(v) - nmsg(v) \times o_r - \varepsilon(u_l)$ where $nmsg(v)$ is the number of messages received from immediate ancestors of v which do not belong to $C(v)$. For the other tasks $u_i \in C(v)$ with $i = l - 1, \ldots, 1$, $dst(u_i, v) = dst(u_{i+1}) - nmsg(u_{i+1}) \times o_r - \varepsilon(u_i)$.

The latest message send time, $lmst_{M1}(w, u_i, v)$, of (w, u_i) under Mechanism One is now defined as a function of $dst(u_i, v)$. This provides the scope to increase both the processing window associated with $C(w)$ due to the message (w, u_i) even further, and the possibility for this copy of $C(w)$ to send other messages.

Let $\mathcal{S} = \{(w_1, u_i), \ldots, (w_j, u_i), \ldots (w_k, u_i)\}$ be the subset of messages sent to task $u_i \in C(v)$ which arrive before u_i's overheads are processed (i.e. $dst(u_i, v) - nmsg(u_i) \times o_r$). The latest message send time of messages (w_j, u_i) in $\mathcal{S}$ is $lmst_{M1}(w_j, u_i, v) = dst(u_i, v) - (o_s + L \times \omega(w_j, u_i) + nmsg(u_i) \times o_r)$. But not all the messages sent to u_i necessarily arrive before this time. Let $\mathcal{L} =$

$\langle (w_1, u_i), \ldots, (w_q, u_i), \ldots, (w_k, u_i) \rangle$ be the list of those messages in non-increasing order of their arrival time. The latest send time for each message in $\mathcal{L}$ is given by $lmst_{M1}(w_q, u_i, v) = dst(u_i, v) - (o_s + L \times \omega(w_q, u_i) + l \times o_r),\ q = 1, \ldots k.$

Mechanism Two: Ordering the Receive Overheads. In Mechanism One, messages in $\mathcal{S}$ are delayed sufficiently only to allow them to arrive at the same time, i.e., the time at which the first of the respective receive overheads for u_i is processed. The processing windows of respective senders can be increased if these messages were to arrive at $C'(v)$ just in time for the execution of their receive overheads. This mechanism refines the results of Mechanism One, by reordering the messages $(w, u_i) \in \mathcal{S}$ (and thus removing restriction R4) according to the number of messages that task w needs to send to its immediate successors $\in rsc$. Delaying the receiving of w's message as much as possible, again increases $pw(w, u_i, v')$.

Defining the Reduced Set of Clusters: Once the mechanisms have been applied to increase the processing window of $C(w) \in uc$, the number of copies of cluster $C(w)$ required to send all of the messages of w to its successors $\in rsc$ has to be found. The list of messages sent by owner w to immediate successor clusters $\in rsc$, denoted by list of tuples $mlist(w) = \langle [u_i, v] \mid u_i \in C'(v) \wedge C'(v) \in rsc \wedge (w, u_i) \in E \wedge w \notin C'(v) \rangle$, can easily be calculated. For each message in the list $mlist(w)$, the *sending order* $so(w, u_i, v) = \lfloor pw(w, u_i, v)/o_s \rfloor + 1$, determines the latest position of (w, u_i) in the sequence of messages sent by a copy of cluster $C(w)$. In order to establish the number of messages that a copy of $C(w)$ can send, the tuples in $mlist(w)$ are organised in non-decreasing order of their respective $so(w, u_i, v)$.

The number of copies of $C(w)$ necessary and the messages that each copy must send are specified as follows. For each tuple $[u_i, v] \in mlist(w)$ with sending order $so(w, u_i, v) = p$, a search is made for a copy of $C(w)$ which has been assigned $p-1$ messages. If no such copy is found, a search for a copy assigned to $p-2$ messages is tried, and so on until, if necessary, a new copy of $C(w)$ is allocated. The objective is to assign the maximum number of messages to a copy of $C(w)$ without increasing the makespan. When all of the tuples in $mlist(w)$ have been visited, those copies of $C(w)$ with assigned messages (denoted as $C'(w)$) are included in rsc.

Mechanism Three: Delaying the Schedule Time of the Owners. It is possible that the send overheads of $C'(w)$ are still executed earlier than necessary. Delaying the messages will allow owner w and the non-owner tasks $\in C'(w)$ to be delayed too, thus increasing the processing windows of messages sent to $C'(w)$. Also, all sink clusters can be delayed so that they complete their execution by makespan $\mathcal{M}$.

Reducing the Schedule's Processor Count: After the cluster copy elimination process, an attempt is made to minimise the number of processors required by mapping the clusters $\in rsc$ appropriately. This phase is common in most

clustering algorithms [5,14] and typically the mapping process is simple: multiple clusters can be mapped to the same processor if their execution intervals do not overlap, i.e. as long as the execution of a cluster and its send overheads do not delay the execution of another cluster.

It may be possible to reduce further the number of processors required by allowing the makespan to be increased. The mechanisms (and mapping) can be used to increase the makespan incrementally until a satisfactory number of processors is obtained, or to find the number of processors necessary for a given makespan greater than $\mathcal{M}$.

4 Results

This section summarises the results obtained from 5 experiments using the $LogP$ scheduling algorithm, named Configuration 2 (C2) in [5], and a version of C2 (RPM) in which the second stage has been modified to incorporate the mechanisms described earlier. Experiment 1 compares the reduction in the number of both clusters and processors using the same benchmark suite of DAGs [4] (which includes various sizes of in-trees, diamond DAGs and a set of randomly generated DAGs as well as irregular DAGs) and variety of $LogP$ communication parameter values used in [3]. Experiment 2 attempts to analyse the relative merits of each of the proposed mechanisms. Both this experiment and Experiment 3 focus on diamond DAGs since the schedules produced tend to require an exorbitant number of processors. Experiment 3 investigates how the reduction in the number of required clusters is affected by the granularity of the graphs and by the $LogP$ parameters. Experiments 4 and 5 compare the modified scheduling algorithm with another $LogP$ scheduling algorithm, MSA [4], proposed for the bounded processor scheduling problem. Earlier published work showed that, on the whole, while the schedules produced by MSA had poorer makespans, the number of processors required were significantly smaller compared to the schedules of C2 [2]. Experiment 4 identifies the differences in processor requirements for MSA and the modified algorithm. Experiment 5 compares the processor requirements of schedules produced by these two algorithms when the makespans are fixed by MSA, i.e. RPM increases the makespan in an attempt to reduce further the number of processors required.

Experiment 1: Compares the results of 77 schedules for a variety of graphs and $LogP$ conditions. The average reduction in the number of clusters and processors achieved without increasing the makespan was 60.2% and 63% respectively. The average degree of cluster replication ($adcr = |rsc|/|uc|$), which is related to the efficiency of the schedule, improved from 7 to 3.1. In 58 of the 77 cases (75.3%), the number of clusters required ($|rsc|$) was less than that of C2 ($|nc|$) and equal in the remaining 19. In 12 of these 19 cases, the schedules produced contained no replicated clusters ($|rsc| = |uc|$) so no gains were possible. However, a reduction in the number of processors (np) was obtained in 65 (84.4%) cases. This shows that in 7 (9.1%) cases where the mechanisms appeared unsuccessful, the rescheduling of the clusters improved the processor mapping.

Experiment 2: In order to compare the merits of the different mechanisms, four second stage algorithms were implemented: Alg1 delays the messages so that they arrive in time for the processing of their receive overhead (see Example 1 in Section 3); Alg2 is Alg1 together with Mechanism One; Alg3 is Alg2 with Mechanism Two; and Alg4 implements all three mechanisms.

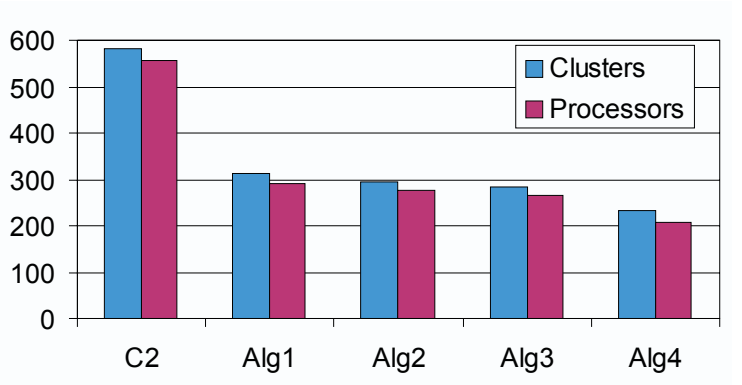

Fig. 2. The average number of clusters and processors required

Alg1: This technique is particularly useful when the communication costs are low, since the schedules contain many copies of small clusters and thus intensive communication, particularly for diamond and random graphs. As the communication costs increase (especially o_s) there is a degradation in the reductions acheived. The average reduction compared to C2 for the same 77 cases was 46.6% for rsc and 47.5% for np. The $adcr$ decreased to 4.1. In addition to the 19 cases cited for C2 in Experiment 1, Alg1 found the smallest rsc (i.e., the other mechanisms were unable to decrease this number) in a further 9 (11.7%). In 6 of these 9 cases, Alg1 eliminated all of the replicated clusters ($|rsc| = |uc|$). The smallest np was attained in 4 cases. In 52 (67.5%) of the cases, $|rc|$ and np were less than C2. These results show that the simple fact of having messages arrive just in time can provide a significant improvement in schedule efficiency.

Alg2: The results, in general, appear to show that there is little to be gained from delaying non-owner tasks within clusters. In comparison to C2, the reduction in rsc was 49.3% and 50.4% for np, up 3% from Alg1 and can be attributed mainly to the diamond DAGs. These results reflect the characteristics of the cluster design process, indicating that clusters contain few idle periods.

Alg3: The reduction in rsc and np now improved to 51.2% and 52.2%, respectively. Here, the gains were achieved principally by the random graphs. In 56 (70.1%) and 61 (79.2%) of the cases, $|rsc|$ and np were less than C2, respectively. To exploit the advantages of re-ordering message receptions, it is necessary to have clusters in which a task has various immediate ancestor clusters.

Alg4: In addition to the results presented earlier in Experiment 1, one further point re-emphasises the importance of Mechanism 3 to the mapping process. All of the mechanisms are necessary to attain the smallest rsc in only 19 (24.7%) cases, while they are necessary in 58 (75.3%) in order to find the smallest np.

Experiment 3: Diamond *DAG*s were used because of their regularity and high degree of cluster replication in the schedules produced by algorithms based on the design methodology. The results show, firstly, as the tasks in the graph increases so does the % reduction in the number of clusters achieved by RPM, independent of the communication parameters. As the number of tasks in G increases, so does $|nc|$ but at much faster rate due to replication. The % reduction also increases as the granularity of G increases, independent of graph size. Increases in the $LogP$ parameters causes C2 to generate fewer necessary clusters. While the % reduction is not affected by increasing latencies, increasing o_s limits the amount of reduction achieved since fewer messages can be sent by a cluster copy. However, the % reduction in the number of clusters is extremely high (more than 80%) when o_r is not too small. The only exceptions occur when $adcr$ is almost one.

Experiments 4 and 5: Although the new mechanisms cause a dramatic decrease in the number of processors in comparison to C2, with MSA [4] the results are not as marked. MSA requires on average 92.8% *fewer* processors [3] compared to C2, and 82.1% when compared to RPM. If diamond *DAG*s are excluded, 56.6% fewer processors than C2 becomes 24.8% with RPM. Bear in mind that this represents the fact that C2 employs, on average, almost 13.9 times as many processors as MSA. RPM uses 5.6 times as many, with the benefit of a better makespan in 95.3% of the 107 cases tested.

When RPM degrades the makespan to that found by MSA, np decreases further. MSA now only requires 33.8% (and 24.3%, when excluding diamond *DAG*s) fewer processors than RPM. This is equivalent to RPM using, on average, 1.5 times as many processors. MSA performs better since it does not replicate clusters but instead bundles messages sent to common processor destinations in order to minimise the number of send and receive overheads incurred.

5 Conclusions

This paper presented a series of mechanisms to be used in the second stage of a existing clustering algorithm design methodology [3] for task replication based $LogP$ scheduling heuristics. The first stage of heuristics based on this design methodology is responsible for the creation of clusters from which a (virtual) schedule can be constructed. While the makespans of these schedules are good, their implementation cost in terms of processor numbers can be extremely high [2,3,5]. The mechanisms proposed in this paper, reduce the number of processors required by schedules (generated by one of the best heuristics based on the methodology [5]), on average, by 63%. Using these same mechanisms, it is possible to further reduce the processor requirements at the expense of an increase in the makespan for the given schedule. This can be viewed as an alternative approach to the problem of scheduling on a fixed number of processors, but with the advantage of knowing the performance benefits of increasing the number of available processors.

Note that the schedules produced by $LogP$ heuristics which employ these mechanisms satisfy the $LogP$ network capacity constraint of there being at most

$\lceil L/g \rceil$ messages in transit to any processor or from any processor at any time[7]. The gains obtained by these mechanism are achieved by relaxing two of the four cluster design restrictions (R2 and R4) used by the first stage.

References

1. A. Alexandrov, M. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model - one step closer towards a realistic model for parallel computation. *7th Annual Symposium on Parallel Algorithms and Architectures (SPAA'95)*, 1995. 157
2. C. Boeres, A. Nascimento, and V. E. F. Rebello. Scheduling arbitrary task graphs on LogP machines. In P. Amestoy *et al.*, editors, *The Proceedings of the 5th International Euro-Par Conference on Parallel Processing (Euro-Par'99)*, LNCS 1685, pages 340–349, Toulouse, France, August 1999. Springer. 157, 162, 164
3. C. Boeres, A. P. Nascimento, and V. E. F. Rebello. Cluster-based task scheduling for LogP model. *International Journal of Foundations of Computer Science*, 10(4):405–424, 1999. 156, 157, 158, 162, 164
4. C. Boeres and V. E. F. Rebello. A versatile cost modelling approach for multicomputer task scheduling. *Parallel Computing*, 25(1):63–86, 1999. 157, 162, 164
5. C. Boeres and V. E. F. Rebello. On the design of clustering-based scheduling algorithms for realistic machine models. In *The Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS'01)*, San Francisco, USA, April 2001. IEEE Computer Society Press. 156, 157, 158, 162, 164
6. 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 Proc. of the 4th International Euro-Par Conference on Parallel Processing (Euro-Par'98)*, LNCS 1470, pages 337–346, Southampton, UK, September 1998. Springer. 157
7. D. Culler *et al.*. 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, USA, May 1993. 156, 157, 165
8. T. Kalinowski, I. Kort, and D. Trystram. List scheduling of general task graphs under LogP. *Parallel Computing*, 26(9):1109–1128, 2000. 157
9. W. Lowe and W. Zimmermann. Scheduling balanced task-graphs to LogP-machines. *Parallel Computing*, 26(9):1083–1108, 2000. 157
10. R. P. Martin, A. M. Vahdat, D. E. Culler, and T. E. Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. In *Proc. of the 24th International Symposium on Computer Architecture*, pages 85–97, June 1997. 156, 157
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. 157
12. C. H. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. *SIAM J. Comput.*, 19:322–328, 1990. 156
13. A. Tam and C. Wang. Realistic communication model for parallel computing on cluster. In *Proc. of the 1st IEEE International Workshop on Cluster Computing*, pages 92–101, Melbourne, Australia, December 1999. IEEE Computer Soc. Press. 157, 158

14. 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. 157, 162