Scheduling vs Communication in PELCR

Marco Pedicini[1] and Francesco Quaglia[2]

[1] IAC, Consiglio Nazionale delle Ricerche, Roma, Italy
[2] DIS, Università "La Sapienza", Roma, Italy

Abstract. PELCR is an environment for λ-terms reduction on parallel/distributed computing systems. The computation performed in this environment is a distributed graph rewriting and a major optimization to achieve efficient execution consists of a message aggregation technique exhibiting the potential for strong reduction of the communication overhead. In this paper we discuss the interaction between the effectiveness of aggregation and the schedule sequence of rewriting operations. Then we present a Priority Based (BP) scheduling algorithm well suited for the specific aggregation technique. Results on a classical benchmark λ-term demonstrate that PB allows PELCR to achieve up to 88% of the ideal speedup while executing on a shared memory parallel architecture.

1 Introduction

PELCR (Parallel Environment for Lambda-Calculus Reduction) is a recent software [7] for efficient optimal reduction of λ-terms on parallel/distributed computing systems. The development of this software is based on results in the field of functional programming, which have shown how the reduction of λ-terms can be mapped onto a particular graph rewriting technique known as Directed Virtual Reduction (DVR) [1,2,3,4,5]. In DVR, each computational step corresponds to a transition from a graph G to a graph G' obtained through the *composition* of two labeled edges, say e' and e'', insisting on a node v. Such a composition has the following effects: (i) a new node v' is created with two exiting labeled edges that point to the source nodes of e' and e'' respectively, (ii) the labels of e' and e'' are modified to reflect that the two edges must not be composed anymore.

PELCR allows edge compositions to be performed concurrently by supporting the graph distribution among multiple machines. As respect to this point, the nodes dynamically originated by DVR steps are distributed according to a load balancing mechanism able to prevent overload on any machine.

An additional optimization embedded by PELCR deals with communication, implemented in the form of message exchange based on the MPI layer. More precisely, PELCR adopts a message aggregation technique that collects application messages destined to the same machine (each of those messages notifies the existence of a new edge in the graph), and delivers them using a single MPI message. The advantage of aggregation is in the reduction of the network path setup time, due to the reduction in the amount of MPI messages. On the other hand, aggregation delays the delivery of application messages since they are not sent as

© Springer-Verlag Berlin Heidelberg 2002

soon as they are produced, but only after the construction of the aggregate, with possible performance loss anytime the destination machine for the aggregate is idle and is waiting for new load (i.e. for new edges to be composed) to come in. Therefore, the higher the arrival rate of application messages into a given aggregate, the more valuable the aggregate since adequately small send delay for the aggregate allows in any case strong reduction of the communication overhead due to network path setup. The frequency of application messages arrival into a given aggregate, namely the goodness of the aggregate, depends on the order of DVR steps occurring on the machine. Currently those steps are scheduled in FCFS (First Come First Served) order, which means that edge compositions are performed in the same order according to which the machine becomes aware of new edges in the graph. In this paper we extend PELCR functionalities by embedding a Priority Based (PB) scheduling mechanism which aims at increasing the amount of valuable aggregates of the execution. This is done by associating with each edge a scheduling priority computed on the basis of the amount of application messages destined to remote machines which are expected to be produced due to DVR steps involving that edge. Higher priority is assigned in case of higher expected number of application messages, which should increase the frequency of application messages insertion into the aggregates. We report an experimental evaluation whose results point out how the effectiveness of the aggregation process can be increased thanks to PB. In particular, using a classical benchmark λ-term, we show that PB allows PELCR to achieve up to 88% of the ideal speedup due to parallelization, with an increase of up to 10% as compared to the case of FCFS schedule.

The remainder of this paper is structured as follows. In Section 2 we report a short, technical description of PELCR's software engine, which will allow us to easily describe PB in Section 3. Performance data are reported in Section 4.

2 PELCR Engine

The PELCR engine running on the i-th machine involved in the λ-term reduction keeps track of information related to locally hosted nodes of the graph through a list, namely $nodes_i$. The element of the list associated with a node v is denoted as $nodes_i(v)$ and has a compound structure. A relevant field of the structure, called $nodes_i(v).composed$, is a list containing the edges incident on v that have already been composed with each other according to DVR. A buffer $incoming_i$ is used to store received application messages. Each of those messages carries information related to a new edge that must be added to the graph, i.e. it must insist on a locally hosted node, and must be composed with edges, if any, insisting on the same node. Note that $incoming_i$ contains also those application messages that the engine on the i-th machine sends to itself due to notification of new edges produced by locally executed DVR steps. Denoting with m an application message, with e_m the edge carried by the message m, and with $e.target$ (resp. $e.source$) the target (resp. source) node of an edge e, the main loop executed by the engine can be schematized as shown in Figure 1.

```
1  while not $end\_computation$ do
2     ⟨collect all incoming application messages and store them in $incoming_i$⟩;
3     while not $empty(incoming_i)$ do
4        ⟨extract a message $m$ from $incoming_i$⟩;
5        if $e_m.target \in nodes_i$      'node already in the local list'
6        then
7           for each edge $e \in nodes_i(e_m.target).composed$ do
8              ⟨compose $e_m$ with $e$⟩;
9              ⟨select the destination machine for hosting the node originated by the composition⟩;
10             ⟨send the edges produced by the composition to
                 the machines hosting $e_m.source$ and $e.source$, respectively⟩;
11          endfor
12       else ⟨add $e_m.target$ to $nodes_i$⟩;   'add new node to the local list'
13       ⟨add $e_m$ to $nodes_i(e_m.target).composed$⟩;
14    endwhile;
15    ⟨send out all the non-empty aggregates⟩;
16 endwhile
```

Fig. 1. PELCR Engine's Main Loop

It is interesting to note that PELCR adopts message exchange only for the notification of new edges, while it does not adopt message exchange for notification of new nodes in the graph. Any new node destined to the i-th machine is added to $nodes_i$ as soon as the i-th machine becomes aware of the existence of at least one edge that should insist on that node (see line 12). This technique allows to reduce the amount of notification messages at the application level thus tackling the communication overhead problem in a way orthogonal to what done by aggregation. Also, when a new edge e must be added to the graph, access to the structure $nodes_i(e.target)$ is implemented efficiently through a hashing mechanism that allows fast retrieve of the memory address of the structure itself.

To support aggregation, the communication module on the i-th machine collects application messages destined to the j-th machine into an aggregation buffer $out\_buff_{i,j}$ and periodically sends the aggregate content. To determine when an aggregate must be sent, the module keeps an age estimate for each aggregation buffer $out\_buff_{i,j}$ by periodically incrementing a local counter $c_{i,j}$. The value of $c_{i,j}$ is initialized to zero and is set to zero each time the application messages aggregated in the buffer are sent via an MPI message. At the end of the composition phase associated with the extraction of an edge from $incoming_i$, $c_{i,j}$ is increased by one if the corresponding buffer stores at least one application message to be sent. Therefore, one tick of the age counter is equal to the average time for DVR steps associated with an edge extraction from $incoming_i$. Also, the counter value represents the age of the oldest message stored in the aggregation buffer. Messages aggregated into $out\_buff_{i,j}$ are sent when $c_{i,j}$ reaches a well suited maximum value $max_{i,j}$ which is dynamically determined by PELCR in an application specific manner. Actually $max_{i,j}$ represents a kind of time-out for the send delay of application messages aggregated into $out\_buff_{i,j}$.

Note that any non-empty aggregate is sent either when the corresponding time-out expires, or after the completion of the internal cycle in the engine main loop (see line 15). In other words, the application forces the communication module to send out any non-empty aggregate as soon as the extraction phase of edges from $incoming_i$ (and the corresponding compositions) terminates. This is done since delaying the sent of those aggregates until the time-out expiration is expected to not achieve benefits since the application remains busy due to the receipt phase of application messages (see line 2), with no possibility to produce new messages to be inserted into the aggregates for a while. Once inserted the first message into an aggregation buffer, say $out\_buff_{i,j}$, the higher the frequency of message aggregation into that buffer within the expiration of the time-out $max_{i,j}$ (or before the send operation forced by the application), the more valuable the aggregate. Currently, the extraction of application messages in line 4 is performed in FCFS order, i.e. the messages are extracted according to their insertion order in the buffer $incoming_i$. As a consequence there is no attempt to determine a schedule sequence of DVR steps which tends to increase the frequency of application messages insertion into the aggregation buffers. This is exactly the objective of the PB solution we present in the next section.

3 Priority Based Scheduling

Let us consider a snapshot of the content of the buffer $incoming_i$ before the engine starts the execution of the internal **while** cycle in line 3 of its main loop. In other words, the snapshot refers to the buffer content just after the collection phase of application messages to be inserted into the buffer itself. We use the notation M to identify the set of application messages contained into the buffer at the snapshot time. M might be an empty set. We study the problem of determining a well suited schedule of message extractions from $incoming_i$, to be adopted while extracting messages during the execution of the **while** internal cycle. Actually, well suited means a schedule sequence that amplifies the benefits from aggregation performed by the communication module. We use the notation $S = (M, \overset{sc}{\rightarrow})$ to indicate such a schedule sequence, where M has the meaning defined above and $\overset{sc}{\rightarrow}$ is a total ordering relation determining the order of message extractions from $incoming_i$. In other words, given two messages m and m' in M, $m \overset{sc}{\rightarrow} m'$ means that m must be extracted before m' in the cycle.

Each time an edge e_m carried by the application message m is extracted from the buffer $incoming_i$ and composed with an edge e already in the graph and insisting on the node $e_m.target$, the composition typically produces a "non-null" result, which means that a new node and two new edges are originated. However, "null" result might arise in some circumstances depending on the labels of the edges to be composed. In this case, the only effect of composition is to modify the labels of the edges e_m and e in order to keep track that they must not be composed anymore. This is done in practice by adding e_m to the list of already composed edges associated with node $e_m.target$ (see line 13 of the engine main loop). Let us denote as $EDGES(e_m)$ the amount of new edges that would be added to the graph due to the extraction of m from the buffer $incoming_i$, and due to the composition of e_m with other edges already insisting on the node $e_m.target$, computed on the basis of the current state of the portion of the graph locally maintained at the time the schedule sequence must be determined. The value of $EDGES(e_m)$ depends on both (i) the amount of edges currently insisting on $e_m.target$ at the time the schedule sequence must be determined and (ii) the labels of those edges (since composition with e_m might produce null result, with no new edge to be added to the graph). Note that $EDGES(e_m)$ might be different from the real amount of new edges that will be really added to the graph due to the extraction and composition of e_m since the state of the graph at the time of the extraction might be different from the state of the graph at the time the schedule sequence is determined. In particular, changes in the state might be caused by message extractions preceding the extraction of m in the schedule sequence adopted while executing the internal **while** cycle. $EDGES(e_m)$ can be rewritten as $EDGES(e_m) = LOCAL(e_m) + REMOTE(e_m)$, where the two functions $LOCAL(e_m)$ and $REMOTE(e_m)$ denote the amount of new edges that will insist on locally hosted and remotely hosted nodes of the graph, respectively. Note that whether new edges will insist on local or on remote nodes depends on which machines host the source node of e_m and the source nodes of the edges insisting on $e_m.target$ at the time the schedule sequence is determined.

The idea underlying PB is to define a schedule sequence of message extractions from $incoming_i$, to be adopted in the internal **while** cycle, which originates a sequence of DVR steps aiming at maximizing the frequencies of application message insertion into the aggregation buffers. For each buffer $out\_buff_{i,j}$, the frequency refers to the interval between the instant of the first message insertion into $out\_buff_{i,j}$ and the send time of the aggregate, i.e. the lifetime of the aggregate maintained into that buffer. We recall that finding an optimal schedule is not viable in practice since the outcome of DVR steps (and thus the effects of DVR steps on aggregation) depends on the current state of the graph at the time the steps are performed. Such a state is unpredictable at the time the schedule sequence must be determined. For this reason we have taken the practical approach to define PB as a heuristic that finds the best suited schedule sequence on the basis of the current graph state at the time of the determination of the schedule sequence, i.e. before starting the execution of the internal **while** cycle.

Let us define a precedence relation among messages in the set M.

Definition 1. *Message $m \in M$ precedes message $m' \in M$, denoted $m \prec m'$ if, and only if, $REMOTE(e_m) > REMOTE(e'_m)$.*

In other words, the relation $\prec$ defines a partial order $\widehat{M} = (M, \prec)$ among all the messages belonging to M. Introducing messages into the schedule sequence according to any linear extension of the partial order defined by the relation $\prec$ has the effect to maximize the predictable number of application messages to be inserted into the aggregation buffers per tick of the $c_{i,j}$ counters. This, in its turn, allows the maximization of the average message insertion frequency, computed overall those buffers, predictable at the time the schedule is defined. (We use the term predictable just because those quantities refer to the current state of the graph at the time the schedule must be defined.) Therefore PB is a heuristic that gives extraction priorities to messages according to the message ordering associated with the linear extension. The algorithm for PB is as follows:

```
⟨find a linear extension $EXT$ of $\widehat{M} = (M, \prec)$⟩;
⟨$S = (M, \overset{sc}{\rightarrow})$ is such that, given $m \in M$ and $m' \in M$, $m \overset{sc}{\rightarrow} m' \Rightarrow m$ precedes $m'$ in $EXT$⟩;
```

3.1 Implementation Issues

Approximating the Function $REMOTE$. The function $REMOTE(e_m)$ measures the amount of application messages that would be produced by the composition of the edge e_m on the basis of the current graph state evaluated at the time of the determination of the schedule sequence of message extractions from $incoming_i$. PB needs to compute the value of this function to determine the relation $\prec$ (see Definition 1) required to construct the partial order $\widehat{M}$. Computing this function could be a time consuming operation since it requires: (i) label check for all the edges insisting on the node $e_m.target$ in order to determine the non-null compositions associated with the edge e_m and (ii) identification of the edges insisting on $e_m.target$ having sources hosted by remote machines. These operations require $O(n)$ time if n is the amount of edges currently in the list $nodes_i(e_m.target).composed$ (i.e. the edges currently insisting on $e_m.target$). To reduce the time complexity we have decided to approximate the function $REMOTE(e_m)$ with its upper bound value, namely with the maximum number of remote application messages that would been produced by the composition of the edge e_m (considering the current state of the graph) in case all the compositions themselves were non-null. The upper bound, that we denote as $UB(REMOTE(e_m))$ can be computed in $O(1)$ time by simply maintaining two counters associated with the structure $nodes_i(e_m.target)$. One keeps track of the total amount of egdes currently in the list $nodes_i(e_m.target).composed$. The other keeps track of the amount of edges in the list $nodes_i(e_m.target).composed$ having sources hosted by remote machines. The counters are updated when inserting an edge into the list ([1]).

On the Use of Priority Queues. Although the complexity of the evaluation of the function $REMOTE$ can be reduced to $O(1)$ through the upper bound approximation presented in the previous paragraph, the cost of PB remains at least $O(n \times log_2 n)$ where n is the amount of messages in the buffer $incoming_i$ at the time the schedule sequence must be determined (i.e. $n = |M|$). This cost is due to the construction of the linear extension EXT of the partial order $\widehat{M}$. To bound the cost due to the schedule determination, we have taken the design choice to implement an approximation of PB, which is based on the use of multiple priority queues for buffering the incoming application messages. We call this approximation Multiple Queues Priority Based (MQPB) scheduling.

To implement MQPB, incoming application messages are received and buffered into a set of N distinct buffers, namely $incoming_i^0, incoming_i^1, \ldots, incoming_i^{N-1}$. The buffers are associated with different priorities that decrease with the increase in the buffer index. When an application message m is received, the function $UB(REMOTE(e_m))$ is evaluated (this takes $O(1)$ time). Then the priority level, i.e. the index of the destination buffer for the message, is computed as:

$$\text{index} = \lfloor (N-1) \times (1 - \frac{UB(REMOTE(e_m))}{MAX\_UB} \rfloor \qquad (1)$$

[1] In PELCR each edge explicitly carries the information on the location of its source.

where MAX_UB is the relative maximum of the UB functions observed during the current determination of the schedule sequence (^{2}). While executing the internal **while** cycle in line 3 of the engine main loop, the messages are extracted starting from the buffer with the highest priority and then going to the one with the lowest priority. Messages into the same buffer are extracted according to their order of insertion into the buffer itself. This type of extraction is actually an approximation of the sequence determined by PB since it does not guarantee that given two messages m and m', with $UB(REMOTE(m)) > UB(REMOTE(m'))$, m precedes m' in the schedule sequence. However, it guarantees that messages associated with larger values of $UB(REMOTE)$ are likely to be extracted before. The operational advantage of MQPB is that the assignment of a message to a given buffer, which determines the position of the message in the extraction sequence, can be computed in $O(1)$ time upon the receipt of the message in line 2 of the engine structure.

4 Performance Data

We report in this section some performance data in order to evaluate the capability of MQPB in increasing the effectiveness of message aggregation. As hardware platform we have used a Sun Ultra Enterprise 4000 machine, which is a shared memory multiprocessor equipped with 4 CPUs SuperSparc 167 MHz. The experiments have been performed using a classical benchmark for parallel executions, namely `DD4` [7], corresponding to the λ-term $(\delta)(\delta)\underline{4}$ where $\delta = \lambda x(x)x$ represents the self application. The normal form of this term represents the Church's integer $(4^4)^{4^4}$, whose graph representation manipulated by PELCR has a number of nodes which is in the order of hundreds of thousands.

We report two series of plots (see Figure 2). The first one relates to speedup as a function of the number of used CPUs (from 2 to 4). Speedup results are plotted for the case of parallel execution with both FCFS schedule and MQPB schedule. The speedup is computed against the execution time on a single CPU when FCFS schedule of message extractions is used (FCFS schedule is not penalizing in case of execution on a single CPU since aggregation is activated only in case of execution on multiple CPUs). Also, for the case of MQPB we report results for three different values of the number of priority levels, say 3, 5 and 10. The second series of plots report the average number of application messages (ANAM) delivered through a single MPI message. This parameter is an indicator of the effectiveness of message aggregation. For all the series, each reported value is the average of ten runs. The data show two major tendencies. First, MQPB achieves higher values of ANAM. This reduces the communication overhead, especially for the case of a larger number of CPUs, where application message exchange occurs more frequently. As a consequence, we get an acceleration of

^{2} Actually, using for MAX_UB the relative maximum during the current schedule sequence determination, instead of the relative maximum overall the schedule determinations performed since the beginning of the execution, allows PELCR to better react to dynamics in the graph rewriting.

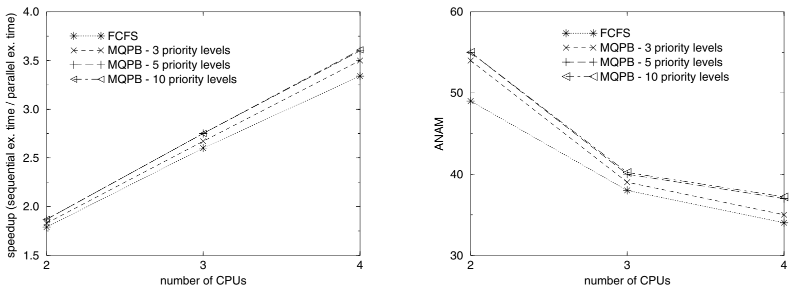

Fig. 2. Speedup and ANAM vs the number of CPUs

the parallel execution (as compared to that achievable with FCFS schedule) that for the case of 4 CPUs is in the order of 10%. The second tendency is related to the behavior of MQPB vs the number of priority levels. Specifically, moving it from 3 to 5 allows improvements in the aggregation process (see the values of ANAM in Figure 2). Instead, moving it from 5 to 10 does not produce additional advantages, i.e. the behavior of MQPB stabilizes when the number of priority levels gets over a given threshold. As a final observation, the speedup with MQPB is between 85% and 88% of the ideal one. This is an indication that MQPB actually brings PELCR near an ideal parallel execution.

References

1. V. Danos, M. Pedicini, and L. Regnier. Directed virtual reductions. In M. B. D. van Dalen, editor, *Computer Science Logic, 10th Int. Workshop, CSL '96*, volume 1258 of *Lecture Notes in Computer Science*, pages 76–88. EACSL, Springer Verlag, 1997.
2. V. Danos and L. Regnier. Local and asynchronous beta-reduction (an analysis of Girard's EX-formula). In *Logic in Computer Science*, pages 296–306. IEEE Computer Society Press, 1993. Proceedings of the Eight Annual Symp. on Logic in Computer Science, Montreal, 1993.
3. J.-Y. Girard. Geometry of interaction 1: Interpretation of system F. In R. Ferro, C. Bonotto, S. Valentini, and A. Zanardo, editors, *Logic Colloquium '88*, pages 221–260. North-Holland, 1989.
4. G. Gonthier, M. Abadi, and J.-J. Lévy. The geometry of optimal lambda reduction. In *Proc. of 17th Annual ACM Symp. on Principles of Programming Languages*, pages 15–26. Albuquerque, New Mexico, January 1992. ACM Press.
5. J. Sousa Pinto. Parallel implementation models for the lambda-calculus using the geometry of interaction. In *5th International Conference on Typed Lambda Calculi and Applications*, pages 385–399. Krakow, Poland, May 2001. LNCS 2044.
6. J. Lamping. An algorithm for optimal lambda calculus reduction. In *Proc. of 17th Annual ACM Symp. on Principles of Programming Languages*, pages 16–30, San Francisco, California, January 1990. ACM Press.
7. M. Pedicini, and F. Quaglia. A Parallel implementation for optimal lambda-calculus reduction. In *Proc. of 2nd ACM Int. Conference on Principles and Practice of Declarative Programming*, pages 2–13, Montreal, Canada, September 2000. ACM Press.