Characterizing asynchronous broadcast trees for multifrontal factorizations

Patrick R. Amestoy, Jean-Yves L'Excellent, Wissam M. Sid-Lakhdar

To solve sparse systems of linear equations, multifrontal methods [2] rely on partial LU decompositions of dense matrices called fronts. The dependencies between those decompositions form a tree, which must be processed from bottom to top in a topological order. We consider a parallel asynchronous setting where 1D acyclic pipelined decompositions are used. At each node N_i of the multifrontal tree, factored panels have to be broadcast to other processes involved in N_i. Because of the asynchronous environment considered, we use $w-$ary broadcast trees aiming at better controlling communication memory and pipeline efficiency than, for example, a binomial tree or a standard `MPI_IBCAST` primitive.

In our asynchronous model, memory is needed for the communication of factored panels. In particular, a process involved in a broadcast tree will store a factored panel (e.g., on reception), will relay (or just send, for the root of the broadcast tree) it to all its successors in the broadcast tree, and the memory for the panel will be freed only when all successors have received the panel sent. When memory for communications is limited (for a large problem, a typical panel to be sent might require 200 Mbytes) deadlocks may appear. In this work, we aim at avoiding deadlocks while designing efficient communication patterns, using the available communication memory as much as possible for performance.

Let us examine a simple case of deadlock. Let 1, 2, x, y, a and b be processes involved in the computation of two (fronts) tasks T_1 and T_2. Fig. 1 (left) shows broadcast tree branches of T_1 and T_2, with arrows representing messages paths. On Fig. 1 (right), an arrow $i \to j$ indicates that freeing a memory resource on j (corresponding to a message sent to i) depends on i performing the associated reception. Depending on the order of messages, there can be a cycle (in red) in the dependency graph between resources. Assuming that each process only has one communication buffer, if 1 receives a message from x and 2 receives a message from y, when 1 relays its message to 2, 2 is already full and will not be able to receive the message from 1. Similarly, 1 will not be able to receive the message from 2 leading to a deadlock [1].

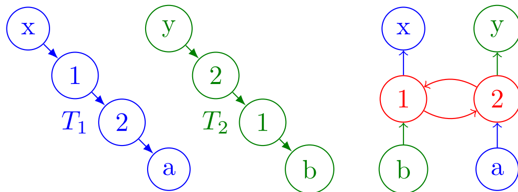

Figure 1: Branches in broadcast trees (left) with a possible cycle between resources (right).

Assuming that each process has two memory buffers for communication instead of one, the deadlock may still happen if process 1 fills its two buffers with messages from x (thus not receiving messages from 2) and 2 receives two messages from y (thus not receiving messages from 1). The only way to avoid the deadlock is to receive and relay messages to the leaves a and b before resources are full of messages in the cycle shown in Fig. 1 (right). Reserving one resource for T_1 and another one for T_2 on both 1 and 2 also prevents the deadlock, although resources may be wasted if T_1 and T_2 are not both active together. A more dynamic approach consists in receiving messages in natural order as much as possible, and then avoiding deadlocks by forcing the last available resources to be used for messages that can be relayed outside the cycles. In this situation, deadlock avoidance could here consist in having 1 and 2 receiving messages that can be relayed to the leaves a and b. Property 1 provides a simpler solution, not requiring any knowledge of distributed (dynamic) cycles [3]:

Property 1. *Assume that a global order has been defined between tasks (nodes). If, each time a process P_i only has one remaining free communication resource (others being busy), it dedicates this ressource to communications involving the smallest task it is mapped on, then deadlocks cannot occur.*

Coming back to 1D asynchronous factorizations, the overall approach is sketched in Algorithm 1 and relies on fairly standard hypothesis for a fully asynchronous context: **(H1)** Computation and relay operations associated to a message are atomic (line 3 of the algorithm). In particular, a message arriving too soon is not relayed before local operations can be done. **(H2)** At each node of a broadcast tree, if memory is available, the message is sent to all successors in the broadcast tree (send to all or to no one). **(H3)** If m_1 is sent from P_i to P_j before m_2, then m_1 is received by P_j before m_2.

```
1: while (! global termination) do
2:     if (some received messages can be processed) then
3:         process them (computations followed by relay in
           broadcast trees)
4:     else
5:         check whether a new local node can start: activation
           of new broadcast tree (multiple non-blocking sends)
6:     end if
7: end while
```

Algorithm 1: Asynchronous multifrontal scheme.

Fig. 2 shows an example of partial decomposition of a dense matrix, that can be interpreted in the multifrontal method as a chain of fronts in the multifrontal

tree. At each node of this chain (here with just a child C and a parent P), one process sends panels to other processes using broadcast trees (TC for child process 1 and TP for parent process 2). The fact that red panels must be computed and treated before blue ones is naturally represented by a causality link between TC and TP, formally defined as follows.

Definition 1. *Let TC and TP be two broadcast trees. We define the child-parent* **causality link** *between TC and TP by the relation:* $\forall P_i \in TP$, *if* $P_i \in TC$, *all activities of* P_i *in TC must be finished before any activity of* P_i *in TP can start.*

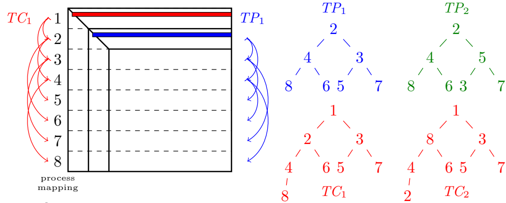

Figure 2: 1D pipelined factorization and several broadcast trees: TC for child, TP_1 or TP_2 for parent. We assume here that process mapping remains unchanged between C and P so that the root of TC does not work in TP.

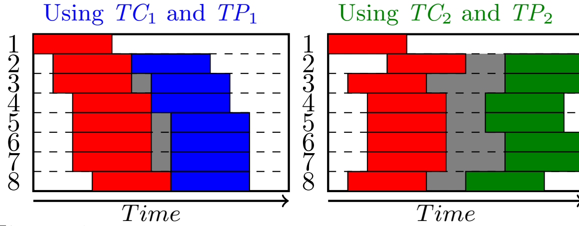

Figure 3: Gantt-charts of child (red) and parent (blue or green) operations; idle periods in gray.

On multifrontal chains or trees it may happen that local ressources of a process, say P_i, are busy because of messages arriving too early or that are not effectivly sent (receiver busy). To apply Property 1 and thus to avoid deadlocks, the last available resource should then be dedicated to communications related to the *smallest* unfinished node involving P_i (smallest in the sense of a global order compatible with causality links, that is, any topological order). In this context, the overhead will depend on the communication patterns and thus on the structure of the broadcast trees, as will be demonstrated in the following.

Definition 2. *Let TC and TP be two broadcast trees. TP is said to be* **IB-compatible** *with TC if,* $\forall N \in TP \cap TC, \exists A \in \{$*ancestors of N in TP*$\}$, *s.t.* $A \in$ *subtree in TC rooted at* $pred_{TC}(N)$, *the predecessor of N in TC.*

Property 2 (No wait)**.** *Given a child C and a parent P such that TP is IB-compatible with TC. If a process* P_i *in TP performs a blocking receive on a given message in TC to respect causality links, then* P_i *will not wait because the expected message has already been sent.*

Definition 3. ABCw trees *(Asynchronous BroadCast trees) are defined by the following characteristics, at each level of a multifrontal chain:*
(1) IB-compatibility of TP with TC (no wait);
(2) Width w determined by network topology;
(3) Number of nodes in each child subtree of any node is balanced: difference is at most 1 (this implies minimal height and balanced communications);
(4) Maximum pipeline efficiency between successive child and parent trees (e.g., Fig. 3 shows that TC_1/TP_1 *are much better than* TC_2/TP_2*).*

In order to build all ABCw trees for the chain, we start from a tree at the bottom node respecting (2) and (3). We then apply successive *Ascensions* to build parent trees from child trees. An ascension builds TP from TC by taking the branch in TC whose nodes are roots of the heaviest child subtrees at each level, and then making each node in that branch replace its parent. An example is given in Fig. 2, where $1 \rightarrow 2 \rightarrow 4 \rightarrow 8$ in TC_1 is replaced by $2 \rightarrow 4 \rightarrow 8$ in TP_1. The mapping of the processes at the bottom node induces the mapping of processes in the ABCw trees such as to respect the following properties: (i) at each level, the root of TP is a child of the root of TC; and (ii) the order of the roots of the successive ABCw trees in the chain follows the mapping of processes in the initial front (here 12345678). It can then be proved that by construction ascensions guarantee (1) and (4) and that, if the initial broadcast tree (at chain bottom) has properties (2) and (3), they will propagate on all broadcast trees in the chain.

To conclude, we have proposed in the context of asynchronous multifrontal methods properties avoiding deadlocks and broadcast trees providing good performance in the case of chains of nodes with no remapping between nodes. It can be shown that the lost process at each node of the chain may be reused anywhere in the multifrontal tree with no risk of deadlock. As an extension to this work, we work on using ABCw trees to: (i) the case where rows are remapped between a child and a parent (ii) the case of general trees. In both cases, the causality definition can be modified to take into account messages exchanges between child and parent nodes in the elimination tree.

References

[1] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Comput. Surv.*, 3(2):67–78, June 1971.

[2] I. S. Duff and J. K. Reid. The multifrontal solution of unsymmetric sets of linear systems. *SIAM Journal on Scientific and Statistical Computing*, 5:633–641, 1984.

[3] A. N. Habermann. Prevention of system deadlocks. *Commun. ACM*, 12(7):373–ff., July 1969.

A Appendix

Proof of Property 1. Two necessary conditions for deadlock are resource starvation and the presence of cycles. As long as enough buffer memory is available, no deadlock can occur. Moreover, as no cycle exists in a broadcast tree, a cycle may only occur between distinct broadcast trees. Hence, the dependencies in a cycle are related to two or more tasks. Thus, if the processes in this cycle respect a global order between tasks when they have critically low buffer memory, we guarantee that at least two dependencies in the cycle cannot coexist simultaneously, as the processes will first communicate following the first dependency before starting / continuing the communications in the other dependency. The cycle is then broken. □

Proof of Property 2. Let P_i be a process mapped on $NC \in TC$ and on $NP \in TP$, which has received a message from $pred_{TP}(NP)$, but has not finished its work in TC. Respecting causality implies that as soon as P_i has only a single buffer resource available, it must post the reception and treat messages `msg` in TC (coming from $pred_{TC}(NC)$). The only way to guarantee that a message `msg` has already been sent is to find a path linking this event "P_i has posted the reception of `msg` from $pred_{TC}(NC)$" with the event "$pred_{TC}(NC)$ sends `msg` to NC". As the reception of a message of TP from $pred_{TP}(NP)$ by P_i means that all the ancestor processes of P_i in TP have relayed all the messages in TC (respect of causality) and as one of P_i's ancestors (A) in TP is also mapped in TC in the subtree rooted at $pred_{TC}(NC)$ (IB-compatibility of TP with TC), this implies that all the processes between this ancestor and $pred_{TC}(NC)$ in TC (in particular $pred_{TC}(NC)$) have relayed all the messages in TC (in particular `msg`). Hence, `msg` is guaranteed to have been already sent. More precisely, it has been sent by $pred_{TC}(NC)$ to the sibling of NC subtree that contains A, and thus – thanks to **(H2)** – it has also been sent to NC. □

Generalization of Property 2. Property 2 only considers a child and a parent but can be generalized to a chain of nodes where each child tree is IB-compatible with the corresponding parent tree: if P_i must receive a message corresponding to the smallest (lowest) active front P_i is mapped on, then it can be proven that the message has already been sent. The basic idea of the proof is that, if a path linking events between two succesive fronts exists, it also exists in a chain of successive fronts. This generalization is actually necessary to obtain the "No wait" property of ABCw trees not only between child and parent but also between grandchild and grandparent, . . .