Fault-tolerant matrix factorisation: a formal model and proof

Camille Coti, Laure Petrucci, Daniel Alberto Torres González

LIPN, CNRS UMR 7030, Université Paris 13, Sorbonne Paris Cité
99, avenue Jean-Baptiste Clément
F-93430 Villetaneuse, FRANCE
{camille.coti,laure.petrucci,torres}@lipn.univ-paris13.fr

Abstract. As the size of high performance systems grows, tolerating failures has become a major issue in parallel computing. In this paper, we present a Coloured Petri Net model for a fault tolerant matrix factorisation algorithm. On the model, we prove resiliency properties and completion of the algorithm in presence of failures whatever the size of the input. This illustrates how formal modelling and verification techniques can help designing proofs on distributed algorithms.

1 Introduction

The node increase in High Performance Computing (HPC) makes platforms more subject to failures. Hence, failures can arise anytime, stopping partially or totally the execution (crash-type failures) or providing incorrect results (bit errors) [3]. Therefore, such algorithms should be designed to expect failures and take suitable actions, thus providing a fault-tolerant environment and ensuring the failure-free execution of critical algorithms.

Recent works [2, 1] have exhibited interesting properties for fault-tolerant algorithms. These properties are *not very intuitive*. Nevertheless, *formal models* allow for better *readability* and *understandability*. Here, we have chosen Coloured Petri Nets (CPN) [6] because of the facilities they provide for modelling and validating properties of parallel and distributed algorithms. We thus present a formal model and the associated verifications for the Fault-Tolerant Tall and Skinny QR Factorisation algorithm (FT-TSQR), and prove it tolerates the failures, providing low overhead in a failure-free execution and guaranteeing that the final results are always correct [4].

Outline. Section 2 presents FT-TSQR algorithm. In section 3, a Coloured Petri Net models the fault-tolerant algorithm. Section 4 presents the most relevant properties issued from the CPN and demonstrates that the main properties of the algorithm are correct by construction. Finally, section 5 concludes and draws some perspectives.

2 The FT-TSQR algorithm

Many applications in linear algebra depend entirely on the execution of the Tall and Skinny QR (TSQR) Factorisation algorithm. Its fault tolerant version uses a set of t processes $PROC = \{P_0, P_1, \ldots, P_{t-1}\}$ to calculate the QR factorisation of a tall and skinny matrix $A^{m \times n}$, i.e. a matrix with m rows and n columns, $m \gg n$. The algorithm enjoys data redundancy between processes [5]. The algorithm is fault tolerant thanks to this data redundancy: if a process fails while performing a local operation, there exists another process doing the same operation and holding the same data. Thus the data of the failed process is not lost [3].

Algorithm 1 describes the steps of the fault tolerant TSQR algorithm and figure 2 displays a failure-free execution of the algorithm with four processes. At step s of its execution, a process P_i has a partner process P_{p_i} to exchange its data with. $p_i = i + 2^s$ if $i \mod 2^{s+1} = 0$, and $p_i = i - 2^s$ otherwise. When a data exchange between partners has been performed, both processes are updated and they increment step s (lines 2 and 10).

Initially, matrix $A^{m \times n}$ is divided into sub-matrices $A_i^{\frac{m}{t} \times n}$. Each sub-matrix $A_i^{\frac{m}{t} \times n}$ is factorised as $\tilde{Q}_i$ and $\tilde{R}_i$ by a local QR factorisation on process P_i (line 1). The resulting $\tilde{R}_i$ is sent to the partner process P_{p_i} to perform a local QR factorisation (line 5). P_i waits for the resulting matrix computed by its partner. If the exchange fails, both processes finish either by a failure or at lines 6–7. Otherwise they concatenate their $\tilde{R}_i$ and $\tilde{R}_{p_i}$ matrices to form a matrix $\tilde{R}_{i,p_i}$ (line 8) and compute its local QR factorisation at line 9. The same procedure is repeated. At the end of the computation all surviving processes have the final Q and R matrices.

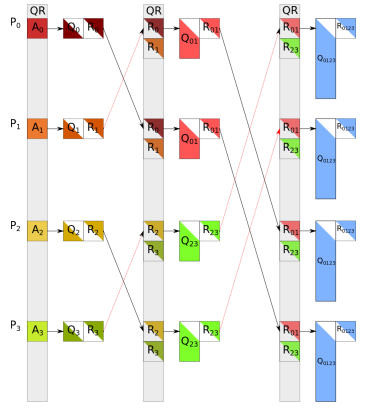

Fig. 1. Failure-free execution steps in redundant TSQR for $t = 4$ processes. Light gray columns represent where the QR Factorization takes place.

Algorithm 1: Redundant TSQR

```
   Data: Submatrix A
1  Q, R = QR(A);
2  s = 0 ;
3  while ! done() do
4      p_i = myPartner(s) ;
5      f = sendRecv(R, R', p_i) ;
6      if FAIL == f then
7          return;
8      A = concatenate(R, R');
9      Q, R = QR(A);
10     s = s + 1 ;
   /* All the surviving processes reach
      this point and own the final R    */
11 return R;
```

3 Model

Figure 2 depicts the Coloured Petri net modelling the algorithm. It focusses on the structure for the functionning of the algorithm, its different steps and the communications between processes, and not the actual computation. Place Processes contains triples (q, s, k) where q is a process number, s the current step and k the index of the $\tilde{R}$ matrix it has already computed. Transition *compute* finds a partner process q' and executes a step of the algorithm. Transition *nop* captures the case when no partner can be found, and the process thus moves to the next step. Finally, place MaxFail contains pairs (s, f) which indicates how many failures are still allowed at step s. It thus limits the number of occurrences of transition *failure*. The model mostly depends on a single parameter: the number of processes in the system. The other ones (number of steps, maximum number of failures) depend on the number of processes.

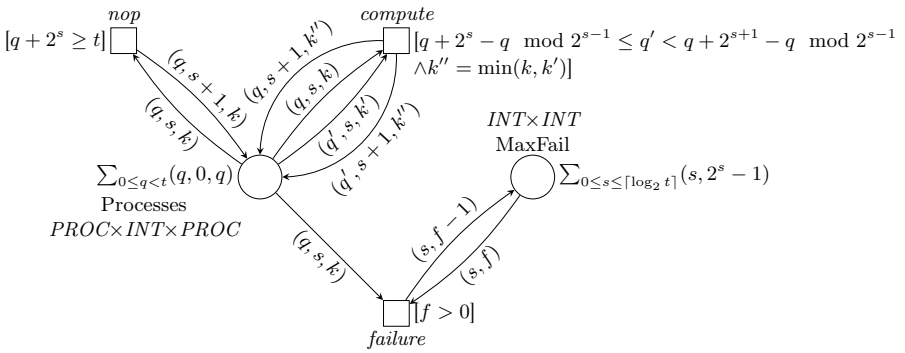

Fig. 2. Model corresponding to Algorithm 1.

4 Properties

We now prove that (a) the system can reach the end of the computation (prop. 1) and (b) the final result is unique and, therefore is the expected one (prop. 2). We use projection functions Π_x to select the xth element of a token which has a tuple value $\Pi_{x,y}$ to select the xth and yth elements to form a pair. Π_x^s denotes the value of Π_x when the step number is s.

Property 1. At every step $s > 0$, the system can tolerate at most $2^s - 1$ failures.

Proof. During the first step, each process performs a local computation. Then at every step $s > 0$, transition *compute* takes the $\tilde{R}$ and $\tilde{R}'$ from two processes q and q' and produces $\tilde{R}''$ on both q and q' or transition *failure* consumes a process.

By a trivial recursion, at each step $s > 0$, it holds that: $|\Pi_3^s(M(Processes))| + \Pi_2^s(M(MaxFail)) = 2^s$. However, the guard on transition *failure* ensures that $0 \leq \Pi_2^s(M(MaxFail)) \leq 2^s - 1$. Therefore, we have at each step $s > 0$: $1 \leq |\Pi_3^s(M(Processes))| \leq 2^s$: at least one process holds each intermediate $\tilde{R}$, which is sufficient for the computation to proceed with the next step. □

Property 2. At the end of the computation, if the system did not suffer too many failures (as specified in property 1), at least one process holds the final R.

Proof. This property is easily derived from the proof of property 1. At each step $s > 0$, we have $|\Pi_3^s(M(Processes))| \geq 1$. This is sufficient for the algorithm to reach the final step. We also need to prove that this final $\tilde{R}$ is unique (and therefore, is the final R). We know that, for each $\tilde{R}$: $|\Pi_3^s(M(Processes))| + \Pi_2^s(M(MaxFail)) = 2^s$. Moreover, $0 \leq \Pi_2^s(M(MaxFail)) \leq 2^s - 1$. The final step is when $s = \log_2 t$. Hence, $|\Pi_3^s(M(Processes))| + \Pi_2^s(M(MaxFail)) = t$. As a consequence, all non-failed processes hold the same $\tilde{R}$, which is the final R. □

5 Conclusion and perspectives

In this paper, we have presented a model for a fault tolerant algorithm. The core contributions of this paper are how the failures are modelled, and using the abstraction provided by the model, the design of proofs of fault tolerance properties of the algorithm. The number of processes and size of matrix are parameters of the model, and thus the proof holds for any value. A perspective is to derive a general modelling and verification approach for such fault-tolerant algorithms.

References

1. Coti, C.: Exploiting redundant computation in communication-avoiding algorithms for algorithm-based fault tolerance. In: IEEE Int. Conf. on High Performance and Smart Computing (HPSC). pp. 214–219 (2016)
2. Coti, C., Herault, T., Lemarinier, P., Pilard, L., Rezmerita, A., Rodriguez, E., Cappello, F.: Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant mpi. In: SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing. pp. 18–18 (2006)
3. Coti, C.: Scalable, robust, fault-tolerant parallel QR factorization. In: Khaddaj, S. (ed.) Distributed Computing and Applications to Business, Engineering and Science (DCABES), 2016 15th International Symposium on. IEEE (2016)
4. Coti, C., Lakos, C., Petrucci, L.: Formally proving and enhancing a self-stabilising algorithm. In: Cabac, L., Kristensen, L.M., Rölke, H. (eds.) Petri Nets and Software Engineering. International Workshop, PNSE'16, Torun, Poland. Proceedings. CEUR Workshop Proceedings, vol. 1591, pp. 255–274. CEUR-WS.org (2016)
5. Demmel, J., Grigori, L., Hoemmen, M., Langou, J.: Communication-avoiding parallel and sequential QR factorizations. CoRR **abs/0806.2159** (2008)
6. Jensen, K., Kristensen, L.M.: Coloured Petri Nets: Modelling and Validation of Concurrent Systems. Springer Publishing Company, Incorporated, 1st edn. (2009)