Towards Refinement of Unbounded Parallelism in ASMs Using Concurrency and Reflection

Fengqing Jiang[1], Neng Xiong[1], Xinyu Lian[1], Senén González[2], and Klaus-Dieter Schewe[1(✉)]

[1] UIUC Institute, Zhejiang University, Haining, China
{fengqing.18,neng.18,xinyul.18,kd.schewe}@intl.zju.edu.cn
[2] TMConnected, Linz, Austria

Abstract. The BSP bridging model can be exploited to support MapReduce processing. This article describes how this can be realised using a work-stealing approach, where an idle processor can autonomously grab a thread from a partially ordered pool of open threads and execute it. It is further outlined that this can be generalised for the refinement of an unboundedly parallel ASM by a concurrent, reflective BSP-ASM, i.e. the individual agents are associated with reflective ASMs, i.e. they can adapt their own program.

Keywords: MapReduce · Work stealing · Reflection · Abstract State Machine · BSP bridging model

1 Introduction

The *bulk synchronous parallel* (BSP) bridging model [6] is a model for parallel computations on a fixed number of processors comprising sequences of alternating computation and communication phases. In a computation phase each processor works independently without any form of interaction until it completes the local computation. When all processors have completed their local computations, they continue with a communication phase to exchange data. With all agents having completed their communication, they return to a new computation phase and thus begin a new superstep. BSP computations are specific concurrent algorithms, and as such they are captured by restricted communicating concurrent Abstract State Machines (ASMs) [2] called BSP-ASMs as shown in [3].

A MapReduce computation comprises a *map* phase processing input data to obtain intermediate key-value pairs, a *shuffle* phase redistributing the data and a *reduce* phase aggregating intermediate key-value pairs to yield the final result. In [3] it was shown how BSP-ASMs can be exploited to specify and analyse MapReduce. Examples how MapReduce is realised based on grounds of the BSP model can be found in [4].

In this short paper we first show that the scheduling effort can be minimised by adopting a work stealing approach as introduced in [1]. This approach places

A. Raschke and D. Méry (Eds.): ABZ 2021, LNCS 12709, pp. 118–123, 2021.
https://doi.org/10.1007/978-3-030-77543-8_10

the decision about the next thread to be executed into the individual agents, i.e. whenever an ASM associated with a processor becomes idle, it can autonomously grab a thread from a partially ordered pool of open threads and execute it. We proceed with an outline of work in progress generalising the approach to a general refinement method for unbounded parallelism in ASMs. The problem is that in an implementation only finitely many processors are available, which suggests to look for a refinement of an ASM by a BSP-ASM. In the general case it becomes necessary that the individual agents are associated with reflective ASMs [5], i.e. they must be able to adapt their own program.

In Sect. 2 we present a specification of MapReduce with work stealing by BSP-ASMs, which is based on the work in [3]. In Sect. 3 we describe our observation that this can be used for a general approach to the refinement of unboundedly parallel ASMs by concurrent, reflective ASMs.

2 BSP-ASMs for MapReduce with Work Stealing

For the processing of MapReduce we can assume a shared environment that is accessible by all processes contributing to the computation. In order to simplify synchronisation among the different processes we assume that there exists an environment host machine H as the center spot of the concurrent computation. For the assignment of tasks we require a task queue TQ holding map and reduce tasks. Map and reduce tasks have a similar structure comprising a job name J, associated (bulk) data D, a function label in $\{Map, Reduce\}$, a function F, a set of indices $I \subseteq \mathbb{N}$, and a processing flag in $\{Undo, Processing\}$.

Map tasks are generated by H as computation tasks and corresponding data is associated with them. For storing intermediate results of map functions we require a reduce container RC. When adding a computation job to H, H will also generate a reduce flag rf and add it to RC. When a reduce flag is completed, H generates a reduce task, adds it to TQ and removes the flag from RC. Reduce tasks are generated only by the reduce container RC, if all map tasks of a job have been completed successfully.

The environment is initialized with data and corresponding user-defined map/reduce functions. The environment-host machine H splits the data into m pieces, then creates and pushes m map tasks into the task queue TQ, and adds a corresponding reduce flag rf into the reduce container RC.

ENVIRONMENT_INITIALIZATION =
  Forall $\langle J \in Job, MapF, ReduceF, Data\rangle$ **Do**
    LET $DataBulks = Split(Data)$
    Forall $DataBulk \in DataBulks$ **Do**
      $Task := \langle J, DataBulk, Map, MapF, i, Undo\rangle$
      $H.TQ := H.TQ \cup \{Task\}$

Further computation tasks are added to the environment before the computation starts by the family of machines.

TASK_ASSIGN =
  If Fetch_Request_from(P_i)

Then $Task := H.TQ.findUndo()$
$Task.PF := Processing$
If Fail_Information_from(P_i)
Then $Task.PF := Undo$
If Success_Information_from(P_i)
Then $TQ := TQ - \{Task\}$

REDUCE_TASK_GENERATION =
If Receive_from_Map_Task
Then $rf := RC.find(Job)$
$rf.DataSet(i) := ReceiveData$
If Completed(rf)
Then $Task := \langle Job, DataSet, Reduce, ReduceF, i, Undo\rangle$
$H.TQ := H.TQ \cup \{Task\}$

Each individual process P_i will run its computation task autonomously. If the program detect its availability P_i will communicate with H and fetch a task T_i from TQ, which is either a map or a reduce task.

$P_i =$ $Task := FetchRequest()$
$Task$ execute
If Task_Success
Then Send $SuccessInformation$ to H
If $Task.FT = Map$
Then Send result to $H.RC$
If $Task.FT = Reduce$
Then OUTPUT
Else
Send $FailureInformation$ to H
Restart/reboot process

In case of a map task the process will execute a map phase. While in this phase, the process applies the given map function to the data sequence resulting in multiple $(Key, Value)$ pairs. When the task is completed successfully, the intermediate result is transferred to H. In case of a reduce task the process will execute a reduce phase. Firstly, the process will sort the $(key, valuelist)$ pairs according to Key. Then it will merge the pairs as the output of the computation phase.

An intermediate result returned to H will be stored into one rf based on the job name. The environment host machine H will monitor, whether each reduce flag is completed after passing back from each map invocation. Since there will be m map operations for a certain assignment in total, only when the reduce flag is loaded with whole m intermediate results, H will generate a reduce task based on the completed rf and add to TQ as well as removing this rf from the reduce container.

3 Reflective Refinement of Unbounded Parallel ASMs

We now look at the use of reflective BSP-ASMs in the refinement of an ASM. Our problem is to deal with (unbounded) parallelism in case of finitely many processors. We identify the agents of the target BSP-ASM with the available processors, so our set of agents will be $\{a_i \mid 1 \leq i \leq k\}$ plus the synchronisation agent a_0 which only executes a SWITCH rule. Whenever a machine encounters a parallel construct, it posts program fragments that need to be executed to a *thread pool*. The thread pool should be partially ordered reflecting that some of the open threads depend on others. Then the gist of the refinement is that rules of ASMs need to be modified such that new threads can be posted to the pool and each agent can fetch his next thread from the pool.

Reflective ASMs as defined in [5] use a 0-ary function symbol *self*, which in each state S takes a tree value representing the current signature and the current rule. In each state S the rule represented by a subtree of $val_S(self)$ is considered not just a tree value, but an executable rule $r(S)$, which is used to yield the next state $S + \Delta_{r(S)}(S)$. For this duality we require functions *raise* and *drop* to switch between these two views: *raise* turns a tree value representing a rule into an executable ASM rule and a tree value representing a signature into a set of locations; *drop* turns rules and locations into tree values [5].

The location *self* can be updated like any other location defined by the current signature, i.e. both the signature and the rule can change in every step. However, to support multiple updates of only parts of the tree, reflective ASMs permit also partial updates. Updates concerning the same location ℓ produced by partial assignments are first collected in a multiset $\ddot{\Delta}_\ell$. If the operators and arguments are compatible with each other, this multiset together with $\mathrm{val}_S(f(t_1, \ldots, t_n))$ will then be collapsed into a single update (ℓ, v_0).

In order to turn a BSP-ASM into a reflective BSP-ASM the signature $\Sigma_{i,loc}$ must contain a function symbol $self_i$, and the represented rule must be a BSP rule over Σ_i, in which the process rule $r_{i,proc}$ and the barrier rule $r_{i,comm}$ are represented by subtrees of $val_S(self_i)$. The creation of new threads and the fetching of open threads must then become part of the process rule.

A *thread* ϑ is given by a rule $r(\vartheta)$ together with a set of locations, in which the rule needs to be executed. Let us associate in addition each thread with a unique *identifier* id_ϑ and a set $pred(\vartheta)$ of those identifiers of threads that need to be executed before ϑ. The rule $r(\vartheta)$ can be represented by a tree value $t_{r(\vartheta)}$, and also the required state can be represented by a tree value representing a set of terms. It has to be understood that the terms represented in this tree represent the locations that need to be evaluated to execute the rule. Thus, the terms refer to the locations of the individual ASM associated with a particular agent, so we can add the agent a_i to the tree representation.

Thus, a thread ϑ is represented as a tree value

$$\vartheta = \mathit{label\_hedge}(\texttt{thread}, \texttt{id}\langle id_\vartheta\rangle\, \texttt{rule}\langle r_\vartheta\rangle\, \texttt{agent}\langle a_\vartheta\rangle\, t_{sig}\, t_{pred})$$

with

$$\begin{aligned} t_{sig} &= \mathit{label\_hedge}(\texttt{signature}, \\ &\qquad \mathit{label\_hedge}(\texttt{func}, \langle f_1\rangle\langle a_1\rangle) \ldots \mathit{label\_hedge}(\texttt{func}, \langle f_k\rangle\langle a_k\rangle)) \\ t_{pred} &= \mathit{label\_hedge}(\texttt{pred}, \texttt{id}\langle id_1\rangle \ldots \texttt{id}\langle id_m\rangle) \end{aligned}$$

Then the local thread pool $pool_i$ associated with an agent a_i is a set of set of such threads that is also represented by a tree.

It must be possible to make the open threads ϑ with $pred(\vartheta) = \emptyset$ available to other agents as well. This can be done in a communication phase. An agent a_i when executing a step in the computation phase and adding new threads to its thread pool also sets $bar_i :=$ **true** to indicate that it is prepared to enter its communication phase. Likewise, if an agent a_i detects $pool_i = \langle\rangle$, it also sets $bar_i :=$ **true**.

When agent a_0 sets *barrier* to **true**, agents a_i broadcast the identifiers and rules of at most $k-1$ of their open threads ϑ with $pred(\vartheta) = \langle\rangle$ to the other agents except a_0. On receipt of message containing open threads, an idle agent a_j selects the most appropriate thread adding it to its own thread pool and returning a message to the agent a_i that created the thread. The agent a_i on receiving a selection may acknowledge it by evaluating the terms and sending the data, i.e. the terms plus their values to the agent a_j, and removing the thread from its own pool. In case that a thread has been selected by more than one agent, only one selection is acknowledged, the other one is declined by sending an appropriate message. An agent whose selection has been declined makes an alternative choice until it receives a confirmation.

After receiving a confirmation of a selection or after having sent all acknowledgements an agent sets $bar_i :=$ **false**. When agent a_0 sets *barrier* to **false**, all agents resume their computation phase. So the computation phase of agent a_i must start with an initialisation making a thread from the local thread pool $pool_i$ the new active rule by updating $self_i$ and $pool_i$ (removing the thread), and adding the data associated with it (in case the thread came from a different agent a_j) to the local state.

Let us now consider a single reflective ASM $\mathcal{M}$. So let the signature and rule of $\mathcal{M}$ be represented in a location *self*. We can further turn $\mathcal{M}$ into a BSP-ASM with $\mathcal{M}$ associated with agent a_1, while all other agents a_i $(2 \leq i \leq k)$ are associated with an ASM with rule **skip**. We can further assume that the rules r_i take the form **choose** $\boldsymbol{y}$ **with** $\psi(\boldsymbol{y})$ **do Forall** $\boldsymbol{x}$ **with** $\varphi(\boldsymbol{x}, \boldsymbol{y})$ **do** $r'_i(\boldsymbol{x}, \boldsymbol{y})$. So the evaluation of $\psi(\boldsymbol{y})$ and $\varphi(\boldsymbol{x}, \boldsymbol{y})$ in the current states yields a set of rules $r'_i(\boldsymbol{x}, \boldsymbol{y})$ that need to be executed. Assuming that these rules are independent of each other they give rise to new threads to be added to $pool_i$.

Technically, this means that the **Forall**-rule is considered as a rule term, i.e. *drop* is applied to it. From this term the component rules $r'_i(\boldsymbol{x}, \boldsymbol{y})$ are extracted and the corresponding threads are added to the thread pool, i.e. the rule is refined by **choose** $\boldsymbol{y}$ **with** $\psi(\boldsymbol{y})$ **do Forall** $\boldsymbol{x}$ **with** $\varphi(\boldsymbol{x}, \boldsymbol{y})$ **do** $post(t_{r'_i(\boldsymbol{x},\boldsymbol{y})})$, where $t_{r'_i(\boldsymbol{x},\boldsymbol{y})}$ is the tree term r_ϑ representing the rule $r'_i(\boldsymbol{x}, \boldsymbol{y})$ and *post* is an ASM rule that generates an identifier id_ϑ, computes a tree representation t_{sig} of

a bounded exploration witness W from the rule representing the terms that are necessary to evaluate the rule and adds a new thread to $pool_i$.

The first thread in the pool (if non-empty) then becomes the new rule of the agent a_i, which means that the rule is followed by an update of the rule part of $self_i$ together with an elimination of the first element in the local thread pool. The next $k-1$ threads ϑ_i $(2 \leq i \leq k)$ with $pred(\vartheta) = \langle\rangle$ become subject of a posting message using a barrier rule. If there are less than $k-1$ such threads, all remaining threads are considered.

A thread created by a different agent is usually associated with the transfer of data. It is therefore advisable to first look for new local threads, so we proceed analogously with two decisive differences: (1) As we do not yet know if the conditions $\psi(\boldsymbol{y})$ and $\varphi(\boldsymbol{x}, \boldsymbol{y})$ will be satisfied, we use a conditional rule for the thread with the condition $\psi(\boldsymbol{y}) \wedge \varphi(\boldsymbol{x}, \boldsymbol{y})$; (2) The resulting threads may also depend on the threads that have been created before, so $pred(\vartheta)$ will not be empty. However, this dependence is only partial, and the threads on which the new ϑ depends may have already been executed.

4 Concluding Remarks

In this article we sketched an extension of BSP-ASMs by reflection such that the involved single-agent ASMs can adapt their own programs. We outlined our work in progress how this capability can be used to allow these ASMs to select their next rule from a pool of partially ordered rules. These rules and associated data are produced by the unboundedly many parallel branches of an ASM. In this way we envision to refine ASMs with unbounded parallelism by BSP-ASMs with workstealing modus. The advantage of the approach is that we can dispense with sophisticated scheduling. So far, in this short paper we just illustrated the idea on MapReduce as an example, which greatly benefits from the approach, though it does not require much reflection.

References

1. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. J. ACM **46**(5), 720–748 (1999). https://doi.org/10.1145/324133.324234
2. Börger, E., Schewe, K.D.: Communication in abstract state machines. J. Univ. Comput. Sci. **23**(2), 129–145 (2017). http://www.jucs.org/jucs_23_2/communication_in_abstract_state
3. Ferrarotti, F., González, S., Schewe, K.D.: BSP abstract state machines capture bulk synchronous parallel computations. Sci. Comput. Program. **184** (2019). https://doi.org/10.1016/j.scico.2019.102319
4. Pace, M.F.: BSP vs. MapReduce. In: Ali, H.H., et al. (eds.) Proceedings of the International Conference on Computational Science (ICCS 2012). Procedia Computer Science, vol. 9, pp. 246–255. Elsevier (2012)
5. Schewe, K.D., Ferrarotti, F.: Behavioural theory of reflective algorithms I: reflective sequential algorithms. CoRR abs/2001.01873 (2020). http://arxiv.org/abs/2001.01873
6. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM **33**(8), 103–111 (1990). https://doi.org/10.1145/79173.79181