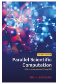
Correctness

Section 5.5 of Parallel Scientific Computation, 2nd edition

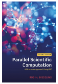
Rob H. Bisseling

Utrecht University



Correct algorithm

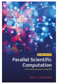
- An algorithm is correct if it does what it is supposed to do, for every legitimate input.
- We require a specification for every designed algorithm that says what the algorithm should do, so that we can prove it correct.
- A correctness proof is best obtained during the process of designing an algorithm, and not after the process has completed.
- This is because higher-level, less optimized versions of an algorithm are usually easier to reason about, and easier to prove correct.
- Our proofs are informal, where the ultimate aim is to show that our parallel matching algorithm is correct.



Termination of the basic dominant-edge algorithm

$\mathcal{M} := \emptyset$;
while $\mathcal{E} \neq \emptyset$ **do**
    pick a dominant edge $(u, v) \in \mathcal{E}$;
    $\mathcal{M} := \mathcal{M} \cup \{(u, v)\}$;
    $\mathcal{E} := \mathcal{E} \setminus \{(x, y) \in \mathcal{E} : x = u \vee x = v\}$;
    $\mathcal{V} := \mathcal{V} \setminus \{u, v\}$;
return $\mathcal{M}$;

- As part of the correctness proof, we should prove that the algorithm terminates in a finite number of steps.
- We start with finite sets of $m = |\mathcal{E}|$ edges and $n = |\mathcal{V}|$ vertices, and we remove at least 1 edge and 2 vertices in every iteration of the main loop, which can be done because the current heaviest edge is always a dominant edge.
- The total number of iterations is therefore at most $\min(m, \lfloor n/2 \rfloor)$.

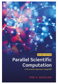


Termination of the local domination algorithm with suitors

while $Q \neq \emptyset$ **do**
    pick a vertex $v \in Q$;
    $Q := Q \setminus \{v\}$;
    $\mathrm{FindAlive}(v, Adj_v, \omega, alive, suitor, d_v)$;
    $(u, v) := \mathrm{FindPref}(Adj_v, \omega, 0, d_v - 1)$;
    $d_v := d_v - 1$;
    $x := suitor(u)$;
    if $x \neq$ **nil** **then**
        $Q := Q \cup \{x\}$;
        $alive(u, x) :=$ **false**;

    ...

- In every iteration, the algorithm either removes a vertex v from Q, or it removes a vertex v and puts a former suitor x back into Q, but then it removes a living edge (u, x).
- Thus, the number of iterations is at most $|\mathcal{V}| + |\mathcal{E}| = m + n$. In practice, it will be a lot less.

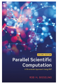


Termination of the parallel algorithm: computation

- At the start of the parallel algorithm, when $Q_s = \mathcal{V}_s$, for all s, the total queue size is

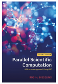
$$\sum_{s=0}^{p-1} |Q_s| = \sum_{s=0}^{p-1} |\mathcal{V}_s| = n.$$

- The local queues $Q_s \subseteq \mathcal{V}_s$ are disjoint, because the local vertex sets $\mathcal{V}_s$ are disjoint.
- At the start of a superstep, a finite number $|R_s|$ of received messages is processed by processor $P(s)$, each in $\mathcal{O}(1)$ time, possibly filling the local queue Q_s.
- This queue is then emptied in at most $|\mathcal{V}_s| + |\mathcal{E}_s|$ iterations, similar to the sequential case.
- Therefore, the computation part of every superstep terminates in finite time.



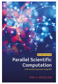
Termination of the parallel algorithm: communication

- All the communications initiated during a superstep are delayed and carried out together just before the synchronization.
- Therefore, we can view the mixed superstep as a computation superstep followed by a communication superstep.
- The algorithm terminates when no communications have been initiated, so that $R_s = \emptyset$, for all s, at the start of the next superstep.
- $Q_s = \emptyset$, for all s, at the start of every superstep, except the first.

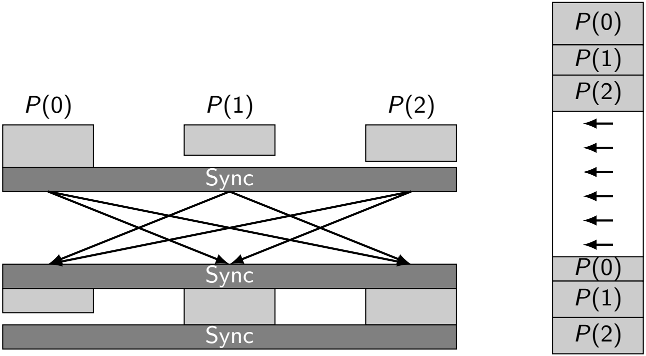


Termination of the parallel algorithm: supersteps

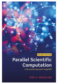
- The total number of proposals sent during the algorithm is at most $2m$, since a vertex v proposes at most once to a neighbouring vertex u (along an edge (u, v)), where mutual proposals can be made.
- Accept or reject messages are only sent in response to a proposal, but not in case of a mutual proposal.
- Therefore, the total communication volume is at most $2m$ and hence the number of supersteps is also at most $2m$ (but in practice a lot less).



Serialization of a BSP algorithm



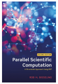
- To check that a BSP algorithm does what it is supposed to do, we serialize it, i.e., transform it into an equivalent sequential algorithm. Communications become assignments.
- This algorithm can then be checked for correctness using any of the already available sequential proof methods.



Serialized algorithm: initialization

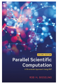
for $s := 0$ **to** $p - 1$ **do**
    for all $v \in \mathcal{V}_s$ **do**
        $\mathit{suitor}(v) :=$ **nil**;
        $\mathit{pref}(v) :=$ **nil**;
        $\mathit{splitter}_v(*) :=$ **false**;
    for all $e \in \mathcal{E}_s$ **do**
        $\mathit{alive}(e) :=$ **true**;
    $\mathcal{M}_s := \emptyset$; $R_s := \emptyset$; $Q_s := \mathcal{V}_s$;

- The initialization superstep has been transformed into a sequential loop over all p processors, where the loop iterations are ordered by increasing processor number s.
- The order does not matter, because the superstep works on disjoint variables. This implies that a single serialized computation represents all possible orderings of the corresponding computation superstep.



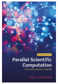
Transforming messages

- We transform sending a proposal (v, u) to the owner of u into adding the proposal message to the set R_t, where $t = \phi(u)$.
- This set acts as a buffer, storing values to be communicated until the next synchronization point.



Receive and send buffers

- When serializing, we must distinguish between messages that were received at the start of the current superstep, stored in the set R_s, and messages that will be sent at the end of the superstep, with those destined for $P(t)$ stored in a set R'_t.
- Without this distinction, a message initiated in the current superstep could already be processed in the same superstep.
- At the end of the superstep, the messages from R'_s are copied into R_s; at the start of the next superstep, R'_s is emptied.

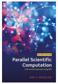


Serialized algorithm: main loop

while $\exists s \; : \; 0 \leq s < p \wedge (R_s \neq \emptyset \vee Q_s \neq \emptyset)$ **do**
    $R' := \emptyset;$
    for $s := 0$ **to** $p - 1$ **do**
        $\mathrm{ProcessReceivedMessages}(R_s, R', Q_s, \mathcal{M}_s, \mathcal{V}_s, \omega, \ldots);$

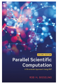
        while $Q_s \neq \emptyset$ **do**
            ...
            { Register a match or propose }
            if $u = \mathit{suitor}(v)$ **then**
                $\mathcal{M}_s := \mathcal{M}_s \cup \{(u, v)\}; \quad d_v := 0;$
                if $u \in \mathcal{V}_s$ **then**
                    $d_u := 0;$
                else
                    $R'_{\phi(u)} := R'_{\phi(u)} \cup \{\mathrm{accept}(v, u)\};$
            else if $u \notin \mathcal{V}_s$ **then**
                $R'_{\phi(u)} := R'_{\phi(u)} \cup \{\mathrm{propose}(v, u)\};$

    for $s := 0$ **to** $p - 1$ **do**
        $R_s := R'_s;$



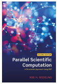
Proving the serialized algorithm correct

- The serialized algorithm retains the original superstep structure, but the termination mechanism can be simplified, because there is no need to communicate to find out whether all processors are done.
- We prove the serialized algorithm correct by showing that it is a more detailed version of the basic dominant-edge algorithm. The main arguments are:
  - The serialized algorithm only adds edges to the matching that are dominant, either when a vertex finds a mutual preference or when it proposes and gets accepted later. If there exists a dominant edge, it will be discovered, sooner or later.
  - Edges incident to the matched vertices are either explicitly removed, or they are retained but implicitly assumed dead because they can never become a match.



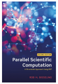
Nondeterminism

- The statement ‘pick a dominant edge’ means that every possible dominant edge is acceptable.
- Picking is arbitrary, and could even happen at random, so there may be no unique outcome and the algorithm may be nondeterministic.
- The nondeterministic pick-statement creates a wider family of algorithms, making it easier to prove algorithms equivalent.
- We use this to our advantage in our parallel matching algorithm where we pick a vertex v from the work queue Q_s, or pick a message msg from the receive queue R_s.
- For the serialized algorithm, we can view R_s as just another work queue, and because of this, the serialized algorithm fits into the overall family of dominant-edge algorithms.



Nondeterminism in communication

- When serializing BSP algorithms, nondeterminism arises because the order in which messages arrive at their destination during the communication superstep is not fixed.
- This nondeterminism is exactly the feature that enables communication optimization by the BSP system in the parallel case.
- Transforming BSP algorithms to a sequential version thus means allowing permutation of the communications between the same source and destination processor.



Summary

- An algorithm is correct if it can be shown that it does what it is supposed to do, for every legitimate input.
- Proving correctness includes proving termination.
- Parallel algorithms can be proven correct by serializing them, and then proving the resulting sequential algorithm correct.
- We have done this for the parallel matching algorithm by showing equivalence of the serialized version to the basic dominant-edge algorithm.
- Serialization turns computation supersteps into a loop over the computation parts of the different processors and it turns communication supersteps into memory copies.
- Here, it does not matter in which order the computation parts of a superstep are carried out, or the messages in the communication supersteps are sent.