1 Recurrence

$S \leftarrow$ SortNodes(G): Sequence of nodes sorted using Alg. 7

$$min\_cost(v, SS_u) = \min_{cfg_v} \{vert\_cost(v, cfg_v) + \sum_{n \in N_u(v)} edge\_cost(v, n, cfg_v, SS_u(n)) + \sum_{n \in N_p(v)} min\_cost(n, SS_u \cup \{cfg_v\})\}$$

where, $N_u(v)$ is the set of unprocessed neighbors of v, $N_p(v)$ is the set of processed neighbors of v, $\sigma : V^* \rightarrow V^*$ is a function that maps a set of processed vertices to their dependent unprocessed vertices, and SS_u is a map from $\sigma(N_p(v))$ to their configurations.

2 Proof of optimality

3 Algorithm

Algorithm 1 Algorithm to determine optimal parallelization strategy for a computation DAG

procedure FINDOPTSTRATEGY($G = (V, E)$, $nprocs$)
  for all $v \in V$ **do**
    $v.processed \leftarrow \bot$
    $v.neigh \leftarrow \{u \in V \mid (u, v) \in E \vee (v, u) \in E\}$
    $v.depends \leftarrow v.neigh$
    $v.tbl \leftarrow \{\{(v, (c_1, c_2, \ldots, c_{v.dim}))\} \mid \forall_{i \in [1, v.dim]}[c_i \in \mathbb{Z}^+] \wedge \prod_{i=1}^{v.dim} c_i \leq nprocs\}$
    ▷ Set of all possible configurations
  end for

  $v \leftarrow$ GETNEXTVERTEX($V, \bot$)
  $opt \leftarrow \emptyset$
  while $v \neq \bot$ **do**
    $n_p \leftarrow \{n \in v.neigh \mid n.processed = \top\}$ ▷ Processed neighbors of v
    $n_u \leftarrow v.neigh \setminus n_p$ ▷ Unprocessed neighbors of v

    ▷ Generate sub-strategies for v and its surrounding nodes, by combining configs/sub-strategies of v and its neighbors. Equivalent to `natural join` operation on the tables of $\{v\} \cup v.neigh$.
    $tbl \leftarrow v.tbl$
    for all $n \in v.neigh$ **do**
      $tbl \leftarrow \{ss_i \cup ss_j \mid ss_i \in tbl \wedge ss_j \in n.tbl \wedge$ ISVALID$(ss_i, ss_j)\}$
    end for
    $costs \leftarrow$ GETCOSTS(v, n_u, n_p, tbl)

    $v.costs \leftarrow$ GETMINCOSTS($costs, n_u$)
    $v.tbl \leftarrow \{cost(1) \mid cost \in v.costs\}$
    $v.processed \leftarrow \top$

    $U \leftarrow \{cfg(1) \mid cfg \in v.tbl\}$ ▷ U is the vertex set in the sub-graph that was processed in this iteration
    for all $u \in U$ **do**
      $u.tbl \leftarrow v.tbl$
      $u.costs \leftarrow v.costs$
    end for
    $opt \leftarrow v.tbl$
    $v \leftarrow$ GETNEXTVERTEX(V, v)
  end while

  ▷ $v.tbl$ of the last vertex processed has exactly a single entry, that corresponds to the optimal strategy for G.
  return opt
end procedure

Algorithm 2 Returns $\top$ if combining two sub-strategies is valid. Two sub-strategies are considered to be invalid if there exists at least one vertex with different configurations in the two sub-strategies.

```
1: procedure IsValid(ss_1, ss_2)
2:     for all cfg_1 ∈ ss_1 do
3:         for all cfg_2 ∈ ss_2 do
4:             if cfg_1(1) = cfg_2(1) ∧ cfg_1(2) ≠ cfg_2(2) then
5:                 return ⊥
6:             end if
7:         end for
8:     end for

9:     return ⊤
10: end procedure
```

Algorithm 3 Computes costs for different sub-strategies.

```
1: procedure GetCosts(v, n_u, n_p, tbl)
2:     costs ← ∅
3:     for all ss ∈ tbl do
4:         cost ← 0
5:         ss_p ← {cfg ∈ ss | cfg(1) ∈ n_p}                ▷ Processed neighbors' configs

6:         for all cfg ∈ ss do
7:             u ← cfg(1)
8:             if u = v then
9:                 ▷ Add vertex cost for the current vertex v.
10:                cost ← cost + GetVertexCost(v, cfg)
11:            else if u ∈ n_u then
12:                ▷ Add edge costs for edges between v and its unprocessed neighbor u.
13:                cost ← cost + GetEdgeCost(v, u, cfg)
14:            else if u ∈ n_p then
15:                ▷ Add the costs for sub-strategies of processed neighbor u from its tbl.
16:                cost_p ← ∃c_p[c_p ∈ u.costs ∧ IsValid(c_p(1), ss_p)]
17:                cost ← cost + cost_p(2)
18:            end if
19:        end for

20:        costs ← costs ∪ {(ss, cost)}
21:    end for
22:    return costs
23: end procedure
```

Algorithm 4 Returns $\top$ if configs of n_u are the same in all sub-strategies of $costs(1)$

procedure HasSameSS($costs, n_u$)
  for all $cost_i \in costs$ **do**
    for all $cost_j \in costs$ **do**
      $ss_i \leftarrow \{cfg \in cost_i(1) \mid cfg(1) \in n_u\}$
      $ss_j \leftarrow \{cfg \in cost_j(1) \mid cfg(1) \in n_u\}$
      if IsValid(ss_i, ss_j) $= \bot$ **then**
        return $\bot$
      end if
    end for
  end for
  return $\top$
end procedure

Algorithm 5 Minimizes *costs* over processed neighbors.

procedure GetMinCosts($costs, n_u$)
  ▷ Partition *costs* into mutually disjoint sets, such that all the elements of a set have the same sub-strategy for unprocessed vertices n_u.
  $\mathcal{P} \leftarrow \{C_1, C_2, \ldots \mid i \neq j \implies C_i \cap C_j = \emptyset \wedge \cup_i C_i = costs \wedge \forall_i[\text{HasSameSS}(C_i, n_u)]\}$

  ▷ Extract the strategy with minimum cost from each set of $\mathcal{P}$.
  $min\_costs \leftarrow \emptyset$
  for all $P \in \mathcal{P}$ **do**
    $min\_cost \leftarrow \arg\min_{cost \in P} cost(2)$
    $min\_costs \leftarrow min\_costs \cup \{min\_cost\}$
  end for

  return *min_costs*
end procedure

Algorithm 6 Returns the next vertex to be processed.

procedure GetNextVertex(V, v_{prev})
  if v_{prev} **then**
    for all $v \in v_{prev}.depends$ **do**
      $v.depends \leftarrow (v.depends \cup v_{prev}.depends) \setminus \{v_{prev}, v\}$
    end for
  end if

  $V_u \leftarrow \{v \in V \mid v.processed = \bot\}$
  if $V_u = \emptyset$ **then**
    return $\bot$
  end if

  $v_{next} \leftarrow \arg\min_{v \in V_u} |v.depends|$
  return v_{next}
end procedure

Algorithm 7 Sort vertices in the order of increasing dependent vertex count.

```
1: procedure SortNodes(G = (V, E))
2:     ▷ For each vertex v, assign all its neighbors as its unprocessed neighbor set v.un.
3:     for all v ∈ V do
4:         v.un ← {u ∈ V | (u, v) ∈ E ∨ (v, u) ∈ E}
5:         v.prior ← ()
6:     end for

7:     U ← V
8:     S ← ()                                                    ▷ Empty sequence
9:     while U ≠ ∅ do
10:        v ← arg min_{v∈U} |v.un|
11:        U ← U \ {v}
12:        v.depends ← v.un
13:        S ← S + (v)                                           ▷ Append v to sequence S
14:        for all u ∈ U do
15:            u.un ← u.un \ {v}
16:            if (u, v) ∈ E ∨ (v, u) ∈ E then
17:                u.prior ← u.prior + (v)
18:            end if
19:        end for
20:    end while
21:    return S
22: end procedure
```