Supplemental Information For: Scalable, Multi-Constraint, Complex-Objective Graph Partitioning

George M. Slota, Cameron Root, Karen Devine, Kamesh Madduri, Sivasankaran Rajamanickam

1 Supplemental Methods

1.1 XtraPuLP Initialization

Here we introduce our novel XtraPuLP initialization algorithm (Algorithm 1), which can be considered a hybrid between the two shared-memory PuLP initialization strategies of unconstrained label propagation [1] and breadth-first search-based graph growing [2], [3], [4]. We utilize a bulk synchronous parallel approach for this and all subsequent stages, while maximizing intra-rank parallelism through threading and minimizing communication load with a queuing strategy for pushing updates among ranks.

The master rank (process 0) first randomly selects p unique vertices from the global vertex set (array $Roots$) and broadcasts the list to other ranks. Each rank initializes its local part assignments to -1, and then, if it owns one of the roots, assigns to that root a part corresponding to the order in which that root was randomly selected.

In each iteration of the primary loop of the initialization algorithm, every rank considers all of its local vertices that are yet to be assigned a part using thread level parallelism. For a given unassigned local vertex v, all neighbors' part assignments (if any) are examined. Similar to label propagation, we track all parts that appear in the neighborhood ($isAssigned$); however, unlike label propagation, we randomly select one of these parts instead of assigning to v the part that has the maximal count among v's neighbors. In practice, doing so tends to result in a slightly more balanced partition at the end of initialization. We also allow a switch that controls the maximum part sizes. We track the current size of each part during this initialization routine (like we'll show in balance and refinement stages below), and we can stop assigning new vertices to that part once some threshold is reached (e.g., 2× desired maximum imbalance).

A thread-local queue Q_{thread} is used to maintain any new part assignment to thread-owned vertices. All threads update a MPI rank-level queue which is used in ExchangeUpdates(). ExchangeUpdates() also returns a queue of updates Q_{recv} for the local rank's ghost vertices. We describe ExchangeUpdates() in Algorithm 2. Algorithm 1 iterates as long as ranks have updated part assignments. The number of iterations needed is on the order of the graph diameter, which can be very large for certain graph classes (e.g., road networks), leading to long execution times for this initialization stage. However, for the small-world networks that we are designing for, this issue is minimal. For other graph classes, alternative strategies such as random or block assignments can be used.

Algorithm 1 XtraPuLP Initialization:
$parts \leftarrow$ XtraPuLP-Init($G(V, E), p$)

```
procid ← localTaskNum()
if procid = 0 then
    Roots(1 ... p) ← UniqueRand(1 ... |V_global|)  ▷ randomly select p roots
Bcast(Roots)
parts(1 ... |V|) ← −1
for i = 1 ... p do
    if Roots(i) ∈ V then
        parts(Roots(i)) ← i
updates ← p
while updates > 0 do
    updates ← 0
    for all v ∈ V in parallel do                          ▷ across threads
        if parts(v) = −1 then
            isAssigned(1 ... p) ← false
            for all ⟨v, u⟩ ∈ E do
                if parts(u) ≠ −1 then
                    isAssigned(parts(u)) ← true
                    updates ← updates + 1
            part_new ← RandTrueIndex(isAssigned)          ▷ randomly select a part from v's neighborhood
            if part_new ≠ −1 then
                Q_thread ← ⟨v, part_new⟩
    Q_rank ← Q_thread                                     ▷ merge thread into rank queue
    Q_recv ← ExchangeUpdates(parts, Q_rank, G)
    for all ⟨v, part_new⟩ ∈ Q_recv in parallel do         ▷ across threads
        parts(v) ← part_new
for all v ∈ V in parallel do                              ▷ across threads
    if parts(v) = −1 then
        parts(v) ← Rand(1 ... p)                          ▷ randomly assign vertices not reached by the main traversal
        Q_thread ← ⟨v, parts(v)⟩
Q_rank ← Q_thread, Q_thread ← ∅                            ▷ merge thread into rank queue
Q_recv ← ExchangeUpdates(G, parts, Q_rank)
for all ⟨v, part_new⟩ ∈ Q_recv in parallel do             ▷ across threads
    parts(v) ← part_new
```

1.1.1 ExchangeUpdates

This method, given in Algorithm 2, sets up and does an Alltoallv exchange of part assignments for vertices within the 1-hop boundary of neighboring MPI ranks. The input queue Q_{rank} contains ⟨vertex, assignment⟩ pairs to be communicated. The returning queue Q_{recv} are the updates

received via the exchange, and they will be processed once this subroutine completes.

Algorithm 2 XTRAPULP Communication Routine: $Q_{recv} \leftarrow$ ExchangeUpdates($parts$, Q_{rank}, $G(V,E)$)

```
procid ← localTaskNum()
nprocs ← numTasksMPI()
sendCounts(1 ... nprocs) ← 0                    ▷ send counts input for MPI Alltoallv call
for all v ∈ Q_rank in parallel do               ▷ across threads
    toSend(1 ... nprocs) ← false                ▷ array to check if we've already sending v's assignment to some rank
    for all ⟨v, u⟩ ∈ E do
        rank ← getRank(u)                       ▷ rank which owns vertex u
        if rank ≠ procid and toSend(rank) = false then
            toSend(rank) = true
            sendCounts(rank) ← sendCounts(rank) + 2   ▷ atomic update
sendOffsets(1 ... nprocs) ← parallelPrefixSums(sendCounts)   ▷ offsets in send buffer for MPI Alltoallv call
tmpOffsets ← sendOffsets                        ▷ create copy, to use and increment while filling the send buffer
for all v ∈ Q_rank in parallel do               ▷ across threads
    toSend(1 ... nprocs) ← false
    for all ⟨v, u⟩ ∈ E do
        rank ← getTask(u)
        if rank ≠ procid and toSend(rank) = false then
            toSend(rank) = true
            offset ← {tmpOffsets(rank) ← tmpOffsets(rank) + 2}   ▷ atomic capture
            sendBuffer(offset) ← v              ▷ fill the send buffer with a vertex and its current assignment
            sendBuffer(offset + 1) ← parts(v)
Alltoall(sendCounts, recvCounts)                ▷ exchange send counts to determine MPI inputs for final Alltoallv
recvOffsets(1 ... nprocs) ← parallelPrefixSums(recvCounts)
Alltoallv(sendBuffer, sendCounts, sendOffsets, Q_recv, recvCounts, recvOffsets)   ▷ final exchange of vertex assignments
```

All aspects of this communication routine are parallelized except for the communication itself. The setup of the per-rank send counts and buffer offsets follows a relatively standard approach, where we examine our local graph for vertices on rank boundaries and update the arrays as appropriate. We use the boolean array $toSend$ to avoid redundant communication by checking if a vertex and its assignment is to already be sent to a given rank.

1.2 XTRAPULP-MM Vertex Balancing Phase

The descriptions of the following algorithms in this and subsequent sections are contained within the primary article. These are included simply for reference, and this supplementary document should not be considered as stand-alone without the primary explanatory text.

Algorithm 3 gives the algorithm of the XTRAPULP-MM vertex balancing phase. This utilizes the $counts$ calculation given by Algorithm 4. Algorithm 4 examines the neighborhood of some vertex v and determines how many neighbors are assigned to each part. Algorithm 5 weights these determined counts based on how currently *underweight* a given part is. The final subroutine is Algorithm 6, which synchronizes all part assignments among tasks.

1.3 XTRAPULP-MM Vertex Refinement Phase

We give the vertex refinement phase in Algorithm 7 (XTRAPULP-VertRefine).

Algorithm 3 XTRAPULP Vertex Balancing Phase: $parts \leftarrow$ XTRAPULP-VertBalance(...)

```
nprocs ← numTasksMPI()
S_v(1 ... p) ← numVertsPerPart(1 ... p)          ▷ global part sizes
C_v(1 ... p) ← 0                                 ▷ per-iteration changes to part sizes
iter_cur ← 0
while iter_cur < I_bal do
    mult ← nprocs × ((X − Y)(iter_tot / I_tot) + Y)
    Q_thread ← ∅
    for all v ∈ V in parallel do                 ▷ across threads
        counts(1 ... p) ← getCounts(...)
        counts(1 ... p) ← weightCounts-Vert(...)
        part_cur ← parts(v)                      ▷ current part assignment
        part_new ← Max(counts(1 ... p))
                ▷ new part selected as index of maximum value in array
        if part_new ≠ part_cur then
            parts(v) ← part_new
            Update(C_v(part_new), C_v(part_cur))
                        ▷ atomically update per-part vertex counts
            Q_thread ← ⟨v, part_new⟩
    Q_rank ← Q_thread                            ▷ merge thread into rank queue
    Synchronize-Vert(...)
    iter_cur ← iter_cur + 1 iter_tot ← iter_tot + 1
```

Algorithm 4 Get per-part neighborhood counts: $counts \leftarrow$ GetCounts(...)

```
counts(1 ... p) ← 0
for all ⟨v, u⟩ ∈ E(G) do
    counts(parts(u)) ← counts(parts(u)) + 1
```

Algorithm 5 Weight neighborhood counts based on current part sizes: $counts \leftarrow$ WeightCounts-Vert(...)

```
Max_v ← Max(S_v(1 ... p), Imb_v)
for i = 1 ... p do
    counts(1 ... p) ← 0
    W_v(i) ← Max(Imb_v/(S_v(i) + mult × C_v(i)) − 1, 0)   ▷ bias weights based on estimate of how underweight the part is
for i = 1 ... p do
    if W_v(i) > 0 then
        counts(i) ← counts(i) × W_v(i)
    else
        counts(i) ← 0                            ▷ part is already too overweight
```

Algorithm 6 Synchronize part assignments among all ranks: Synchronize-Vert(...)

```
Q_recv ← ExchangeUpdates(parts, Q_rank, G)
for all ⟨v, part_new⟩ ∈ Q_recv in parallel do    ▷ across threads
    parts(v) ← part_new
Allreduce(C_v, SUM)                              ▷ determine global per-part changes
for i = 1 ... p do
    S_v(i) ← S_v(i) + C_v(i)                     ▷ update global per-part sizes
```

1.4 XTRAPULP-MM Edge Balancing Phase

We give the balance portion of XTRAPULP-MM (XTRAPULP-EdgeBalance) in Algorithm 8, and the modified subroutines from their associated vertex variants in Algorithms 9 and 10.

1.5 Generalized Multi-weight Partitioning

We give the generalized multi-weight formulation of the XTRAPULP part balancing routine in Algorithm 11. Algorithms 12, 13, and 14 gives the variations on the prior used subroutines for the weighted application.

Algorithm 7 XTRAPULP Refinement Phase:
$parts \leftarrow$ XTRAPULP-VertRefine(...)

$nprocs \leftarrow$ numTasksMPI()
$S_v(1 \dots p) \leftarrow$ numVertsPerPart($1 \dots p$), $C_v(1 \dots p) \leftarrow 0$
$iter_{\text{cur}} \leftarrow 0$
while $iter_{\text{cur}} < I_{\text{ref}}$ **do**
  $Q_{thread} \leftarrow \emptyset$
  for all $v \in V$ **in parallel do** ▷ across threads
    $counts(1 \dots p) \leftarrow$ getCounts(...)
    $part_{cur} \leftarrow parts(v)$
    $part_{new} \leftarrow$ Max($counts(1 \dots p)$)
    if $S_v(part_{new}) + mult \times C_v(part_{new}) + 1 < Max_v$ **then** ▷ don't increase current maximum imbalance
      $parts(v) \leftarrow part_{new}$
      Update($C_v(part_{new})$, $C_v(part_{cur})$) ▷ atomically update per-part vertex counts
      $Q_{thread} \leftarrow \langle v, part_{new} \rangle$
  Synchronize-Vert($G, parts, Q_{rank}, S_v, C_v$)
  $iter_{\text{cur}} \leftarrow iter_{\text{cur}} + 1$
  $iter_{\text{tot}} \leftarrow iter_{\text{tot}} + 1$

Algorithm 8 XTRAPULP Edge Balancing Phase:
$parts \leftarrow$ XTRAPULP-EdgeBalance(...)

$nprocs \leftarrow$ numTasksMPI()
$S_v(1 \dots p) \leftarrow$ numVertsPerPart($1 \dots p$)
$S_e(1 \dots p) \leftarrow$ numEdgesPerPart($1 \dots p$)
$S_c(1 \dots p) \leftarrow$ numCutsPerPart($1 \dots p$)
$C_v(1 \dots p), C_e(1 \dots p), C_c(1 \dots p) \leftarrow 0$
$R_e \leftarrow 1, R_c \leftarrow 1$
$iter_{\text{cur}} \leftarrow 0$
while $iter_{\text{cur}} < I_{bal}$ **do**
  $mult \leftarrow nprocs \times ((X - Y)(\frac{iter_{tot}}{I_{tot}}) + Y)$
  $Q_{thread} \leftarrow \emptyset$
  if $Max_e > Imb_e$ **then**
    $R_e \leftarrow R_e \times (Max_e / Imb_e)$ ▷ edge-weighting bias, exponentially increase until constraint achieved
    $R_c \leftarrow 1$
  else
    $R_c \leftarrow R_c \times Max_c$ ▷ cut imbalance bias
    $R_e \leftarrow 1$
  for all $v \in V$ **in parallel do** ▷ across threads
    $counts(1 \dots p) \leftarrow$ getCounts(...)
    $counts(1 \dots p) \leftarrow$ weightCounts-Edge(...)
    $part_{cur} \leftarrow parts(v)$ ▷ current part assignment
    $part_{new} \leftarrow$ Max($counts(1 \dots p)$) ▷ new part assignment
    if $part_{new} \neq part_{cur}$ **then**
      $parts(v) \leftarrow part_{new}$
      Update($C_v(part_{new})$, $C_v(part_{cur})$) ▷ atomically update per-part vertex counts
      Update($C_e(part_{new})$, $C_e(part_{cur})$) ▷ atomically update per-part edge counts
      Update($C_c(part_{new})$, $C_c(part_{cur})$) ▷ atomically update per-part cut counts
      $Q_{thread} \leftarrow \langle v, part_{new} \rangle$
  $Q_{task} \leftarrow Q_{thread}$ ▷ merge thread into task queue
  Synchronize-Edge(...)
  $iter_{\text{cur}} \leftarrow iter_{\text{cur}} + 1$, $iter_{\text{tot}} \leftarrow iter_{\text{tot}} + 1$

1.6 Complexity Discussion

For this discussion, we use $n = |V|$ and $m = |E|$ to represent the cardinalities of the vertex set and the edge set of the global input graph. We also use t to represent the number of parallel MPI ranks, p to represent the number of parts being computed, w as the number of vertex weights, and d_{max} as the maximum degree of the input graph. We consider the common definitions of space complexity, total work complexity, and parallel time, all using standard big-O notation.

As per our distributed graph representation, each rank gets an approximately equal fractional portion of the input graph. With a CSR representation on an unweighted graph, the edge list requires $O(\frac{m}{t})$ space per rank with the index array requiring $O(\frac{n}{t})$. We also require storage of w vertex weights per vertex for our generalized algorithm, which gives us $O(\frac{nw}{t})$ weights per rank. The other necessary storage, such as global–local translations for vertex identifiers and rank assignments of ghost vertices, are all also dependent on the number of vertices assigned to a rank.

Algorithm 9 Weight neighborhood counts based on current part sizes:
$counts \leftarrow$ WeightCounts-Edge(...)

$Max_v \leftarrow$Max($S_v(1 \dots p)$, Imb_v)
$Max_e \leftarrow$Max($S_e(1 \dots p)$, Imb_e)
$Max_c \leftarrow$Max($S_c(1 \dots p)$)
for $i = 1 \dots p$ **do**
  $W_e(i) \leftarrow$ Max($Max_e / (S_e(i) + mult \times C_e(i)) - 1, 0$)
  $W_c(i) \leftarrow$ Max($Max_c / (S_c(i) + mult \times C_c(i)) - 1, 0$)
for $i = 1 \dots p$ **do**
  if $S_v(i) + mult \times C_v(i) + 1 < Max_v$ **or** $S_e(i) + mult \times C_e(i)$+degree($v$) $< Max_e$ **then**
    $counts(i) \leftarrow counts(i) \times (R_e W_e(i) + R_c W_c(i))$
  else
    $counts(i) \leftarrow 0$ ▷ part is too overweight with respect to vertices or edges

Algorithm 10 Synchronize part assignments among all ranks:
Synchronize-Edge(...)

$Q_{recv} \leftarrow$ ExchangeUpdates($parts$, Q_{rank}, G)
for all $\langle v, part_{new} \rangle \in Q_{recv}$ **in parallel do** ▷ across threads
  $parts(v) \leftarrow part_{new}$
Allreduce(C_v, SUM) ▷ determine global per-part changes
Allreduce(C_e, SUM)
Allreduce(C_c, SUM)
for $i = 1 \dots p$ **do**
  $S_v(i) \leftarrow S_v(i) + C_v(i)$ ▷ update global per-part sizes
  $S_e(i) \leftarrow S_e(i) + C_e(i)$
  $S_c(i) \leftarrow S_c(i) + C_c(i)$

Algorithm 11 XTRAPULP Generalized Balancing Phase:
$parts \leftarrow$ XTRAPULP-Balance(...)

$nprocs \leftarrow$ numTasksMPI()
for $j = 1 \dots w$ **do**
  $S_j(1 \dots p) \leftarrow$weightsPerPart($1 \dots p$)
  $C_j(1 \dots p) \leftarrow 0$
$S_c(1 \dots p) \leftarrow$ numCutsPerPart($1 \dots p$)
$C_c(1 \dots p) \leftarrow 0$
$iter_{\text{cur}} \leftarrow 0$
while $iter_{\text{cur}} < I_{bal}$ **do**
  UpdateWeighting(...)
  $mult \leftarrow nprocs \times ((X - Y)(\frac{iter_{tot}}{I_{tot}}) + Y)$
  for all $v \in V$ **in parallel do** ▷ across threads
    $counts \leftarrow$ GetCounts-Weighted(...)
    $part_{cur} \leftarrow parts(v)$
    $part_{new} \leftarrow$ Max($counts(1 \dots p)$)
    if $part_{cur} \neq part_{new}$ **then**
      for $j = 1 \dots w$ **do**
        Update($C_j(part_{cur})$,$C_j(part_{new})$) ▷ atomically update per-weight part sizes
      Update($C_j(part_{cur})$,$C_c(part_{new})$) ▷ atomically update per-part cut counts
      $parts(v) \leftarrow part_{new}$
      $Q_{thread} \leftarrow \langle v, part_{new} \rangle$
  $Q_{task} \leftarrow Q_{thread}$ ▷ merge thread into task queue
  Synchronize-Weighted(...)
  $iter_{\text{cur}} \leftarrow iter_{\text{cur}} + 1$
  $iter_{\text{tot}} \leftarrow iter_{\text{tot}} + 1$

Algorithm 12 Update the current weighting biases: UpdateWeighting(. . .)

for $j = 1 \ldots w$ **do**
  $Max_j \leftarrow$Max($S_j(1 \ldots p)$, Imb_j),
  if $Max_j > Imb_j$ **then**
    $R_j \leftarrow R_j \times (Max_j / Imb_j)$
  else
    $R_j \leftarrow 1$
$Max_c \leftarrow$Max($S_c(1 \ldots p)$)
for $i = 1 \ldots p$ **do**
  for $j = 1 \ldots w$ **do**
    $W_j(i) \leftarrow Max_j / (S_w(i) + mult \times C_w(i)) - 1$
  $W_c(i) \leftarrow$ Max($Max_c / (S_c(i) + mult \times C_c(i)) - 1, 0$)

Algorithm 13 Get per-part neighborhood counts/gains: $counts \leftarrow$ GetCounts-Weighted(. . .)

$counts(1 \ldots p) \leftarrow 0$
for all $\langle v, u \rangle \in E$ **do**
  $counts(counts(u)) \leftarrow counts(counts(u)) + E_W(\langle v, u \rangle)$
$gains(1 \ldots p) \leftarrow 0$
for $i = 1 \ldots p$ **do**
  for $j = 1 \ldots w$ **do**
    $gain(i) \leftarrow (W_j(i) - W_j(x)) \times (V_{W_j} R_j)$
  $counts(i) \leftarrow counts(i) \times gain(i)$

Algorithm 14 Synchronize part assignments among all ranks: Synchronize-Weighted(. . .)

$Q_{recv} \leftarrow$ ExchangeUpdates($parts$, Q_{rank}, G)
for all $\langle v, part_{cur} \rangle \in Q_{recv}$ **in parallel do** ▷ across threads
  $parts(v) \leftarrow part_{cur}$
for $j = 1 \ldots w$ **do**
  Allreduce(C_j, SUM) ▷ determine global per-part, per-weight changes
Allreduce(C_c, SUM)
for $i = 1 \ldots p$ **do**
  for $j = 1 \ldots w$ **do**
    $S_j(i) \leftarrow S_j(i) + C_j(i)$ ▷ update global per-part, per-weight sizes
  $S_c(i) \leftarrow S_c(i) + C_c(i)$

Our algorithms all also require arrays that track updates and current sizes and weightings on a per-part basis, which requires $O(p)$ space. Overall, our baseline algorithm requires $O(\frac{n}{t} + \frac{m}{t} + p)$ space, while our generalized algorithm requires $O(\frac{wn}{t} + \frac{m}{t} + p)$ space.

In terms of work complexity, all of our primary methods use (heavily) modified variants of label propagation for their processing approach. Label propagation has a known linear work complexity when run for a fixed number of iterations of $O(n + m)$. For our initialization, we require a number of iterations bounded by the diameter of the graph D; however, as each vertex and edge is processed once, as in BFS, the work complexity remains $O(n + m)$. For our XTRAPULP-MM algorithms, we must also iterate over all possible p parts when processing each vertex, giving us a total work complexity of $O(np + m)$. For our generalized algorithms, we must also consider some arbitrary number of w weights for each vertex for each possible part, so we correspondingly have $O(npw + m)$ work complexity.

All of our t MPI ranks run in parallel. Within each rank, our parallelization approach is thread assignments on a per-vertex basis for the label propagation phase and per-update in the communication and update phases. In such per-vertex work assignment, the maximum number of edges a thread will examine for a given vertex is bounded by the maximum degree in the graph d_{max}. We are able to naïvely parallelize all other phases (communication, buffer updating, etc.) of our partitioning algorithms, with the exception of when we must serially update the R, S, W, and Max arrays and values. These updates use nested loops over the number of parts $O(p)$ for XTRAPULP-MM and the number of parts and weights $O(pw)$ for the generalized algorithm. As mentioned for our part initialization approach, we require $O(D)$ synchronizations and iterations for the BFS hybrid. All together, we claim a parallel time complexity of $O(p + d_{max} + D)$ for our baseline algorithm and $O(pw + d_{max} + D)$ for our generalized algorithm when running on $O(m)$ processors.

2 SUPPLEMENTAL RESULTS

2.1 Single-constraint Comparisons

We provide additional comparisons to the recent state-of-the-art partitioner of Meyerhenke et al. [5] (ParHIP) within KaHIP version v2.00. ParHIP uses size-constrained label propagation during the graph coarsening phase, optimizing for edge cut and balancing vertices per part. In its most recent version, it can also balance edges per part, though still as just the single-constraint problem. For comparison, we run our XTRAPULP, PULP [2], and ParMETIS [6] optimizing for the same problem. All codes are run using 16-way parallelism on a single node to allow a direct comparison to shared-memory PULP. We partition 2-256 parts of the *LiveJournal*, *RMAT-22*, and *uk-2002* graphs with a 3% load imbalance constraint. In Figure 2, we compare edge cut (top) and execution time (bottom).

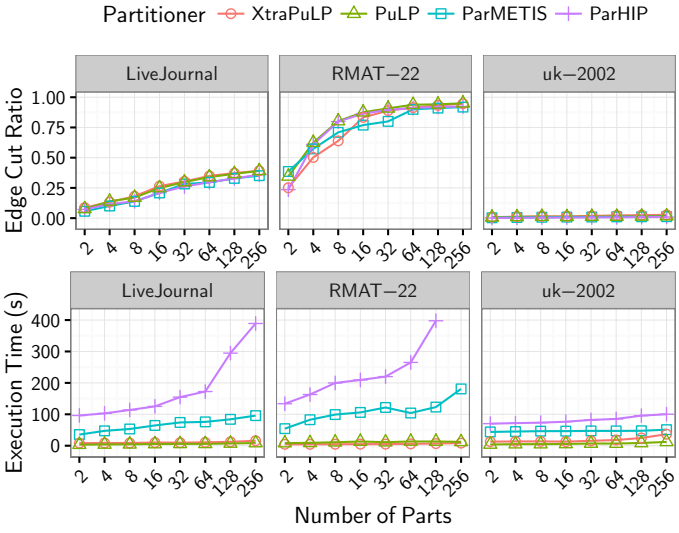

Fig. 1. Partitioning quality (top) and execution time (bottom) for multiple partitioners solving the single objective single constraint partitioning problem.

We define a performance ratio as the cut/time for a given partitioner-graph-#parts combination divided by the best over all partitioners, averaged over all graph-#parts combinations. Performance ratios for cut quality on this limited test set are 1.05 for ParHIP, 1.23 for ParMETIS, 1.51 for PULP, and 1.61 for XTRAPULP. Performance ratios

for execution time are 1.27 for PuLP, 1.73 for XtraPuLP, 11.81 for ParMETIS, and 26.5 for ParHIP. Although we only report the results for 16-way parallelism, testing at a slightly larger scale between XtraPuLP, ParMETIS, and ParHIP reveal similar relative performance. These results in general demonstrate the efficiency tradeoff between quality and time to solution, the choice for which to optimize being application-dependent. However, we emphasize again that we provide these results only to establish a relative baseline for comparison of the performance of XtraPuLP, as the engineering decisions driving its design were made to enable scalability to partition graphs several orders-of-magnitude larger than the graphs presented here.

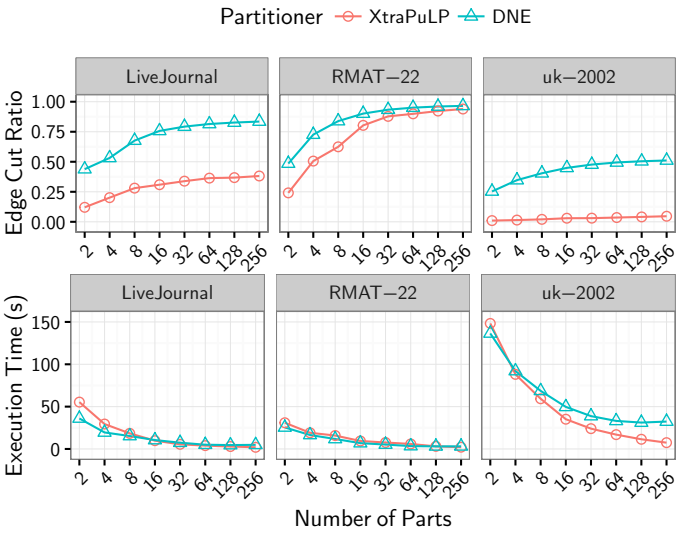

Fig. 2. Partitioning quality (top) and execution time (bottom) for multiple partitioners solving the single objective single constraint partitioning problem.

We also compare against Distributed Neighbor Expansion (DNE) of Hanai et al. [7] and give results for the same graphs in Figure 2. DNE balances edges per part and solves the vertex cut minimization problem. Their code requires one MPI rank per part being computed, so we limit XtraPuLP to one MPI rank per part for a one-to-one comparison; note in practice we can strong scale XtraPuLP with additional MPI ranks. We modify XtraPuLP to balance only edges per part without a vertex constraint. In line with the experimental methodology of Hanai et al., we approximate their resultant edge cut by assigning cut vertices to one part where they have a connection. We measure a performance ratio for cut of 1.0 for XtraPuLP and 6.9 for DNE. It is highly likely that a more complex approach for modifying vertex cuts to edge cuts would likely improve upon this figure. The performance ratios for execution time were measured at 1.35 for XtraPuLP and 1.96 for DNE. We finally note that XtraPuLP scales better versus number of ranks/parts being computed in these test.

2.2 Tuning of Parameters

We also analyze the effect that varying the X and Y parameters have on the final partition quality. Using *LiveJournal*, *uk-2002*, *RMAT-22*, and *nlpkkt160* as representative examples for each graph class, we computed from 2-128 parts each on 2-16 compute nodes of *Compton* (all powers of 2 in between) with X and Y values between 0.0 and 4.0. We plot heatmaps of the average results in Figure 3, with white indicated higher quality or better balance and black indicating poorer quality or balance. For the two quality plots, we omit results that exceed the 10% balance constraint by at least 5% or more. For the balance plots, solid white indicates that all tests conducted for that X, Y achieved the balance constraints.

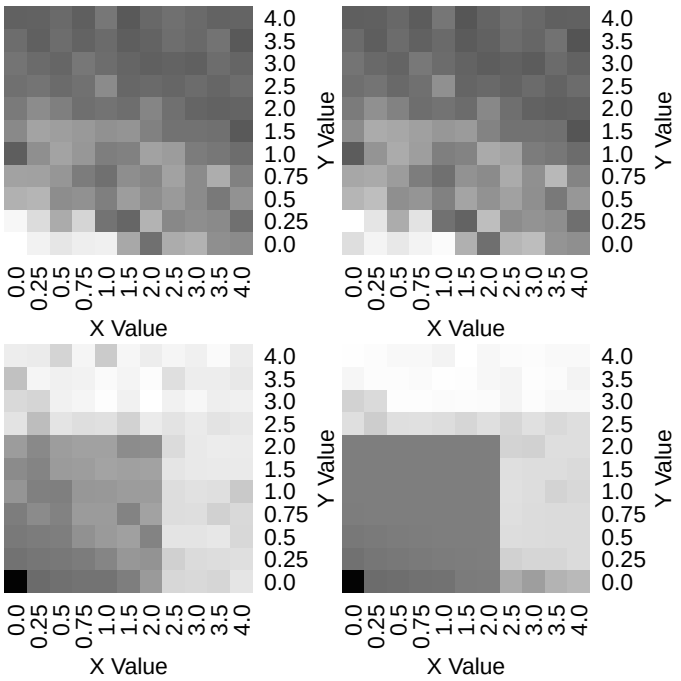

Fig. 3. From the left: Edge Cut versus X, Y; Max Cut versus X, Y; Vertex Balance versus X, Y; Edge Balance versus X, Y. Lighter shading indicates a lower average edge cut and lower average imbalance over all tests.

The top two plots give the edge cut versus X, Y (left) and the max per-part cut versus X, Y (right). From these plots we observe two trends. Most obviously, a lower X and Y indicates a higher quality cut. This is because lower values for these parameters allow the highest number of part reassignments and therefore the greatest overall refinement. Second, we notice that a higher X value relative to Y will, on average, also result in a better cut. This is due to how a higher initial limit on part reassignments (Y) and a lower final limit (X) can greatly refine the initial parts while limiting the potential imbalance possible on the final iterations.

The bottom two plots show overall average edge balance (left) and vertex balance (right). In general, the level of balance achieved is opposite the quality of cut. The optimal X, Y pair of values should therefore be selected along the *threshold*, where high quality and balance are concurrently achieved. We selected our test values of $X = 1.0$ and $Y = 0.25$ empirically, as they gave us the overall best quality in terms of cut and balance on our test suite.

We also examine the impact of our iteration parameters on total execution time, edge cut, and imbalance. In Figure 4 we plot the results of a parametric study on these parameters. We use the same test graphs from before (*LiveJournal*,

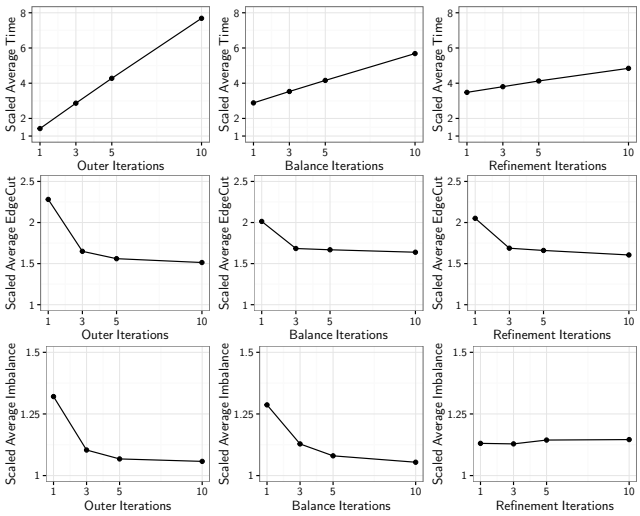

Fig. 4. Parametric study of iteration counts for time (top), edge cut (middle), and average imbalance (bottom). We plot scaled average outputs when varying outer iteration counts (left), balance iteration counts (middle), and refinement iteration counts (right).

uk-2002, *RMAT-22*, and *nlpkkt160*) and vary the outer iterations, balance iterations, and refinements iterations as 1, 3, 5, and 10. We do a full sweep of all combinations. For each graph and iteration combination, we scale the resultant output (time, cut, imbalance) versus the best reported for that specific graph. We then plot the averages over all four tests graphs.

Figure 4 (top) gives scaled average time vs. outer (left), balance (middle), and refinement (right) iterations. We note that increasing refinement iterations is relatively cheap. This is because of our queueing-based approach, where we only have to process assignments that have changed, and refinement only updates a small portion of the total graph each iteration. Figure 4 (middle) gives scaled average edge cut vs. iteration counts. We note that increasing iterations in general improves upon the cut, but that there is a distinct decrease in the slop of the curve around 3-5 iterations for all three studied outputs. Figure 4 (bottom) gives scaled average imbalance vs. iteration counts. We notice again for outer and balance iterations a significant improvement in partition quality per iteration up until about 3-5, where the curve levels off. For refinement iterations, we observe an increase in imbalance as our refinement procedure allows imbalance to increase right up to the limit; our balance procedure often decreases imbalance well below the input constraints.

Overall, in these tests we observe that increasing outer iterations to 3 or 5 is relatively expensive but necessary for cut quality. Increasing balance iterations is moderately time-expensive but shows significant improvement in cut quality to about $\sim$5 iterations. As refinement iterations are the least expensive in terms of time cost, and there is a constant negative slope in edge cut versus iteration count, a larger number of refinement iteration is a time-efficient way to improve relative partition quality. As we've stated, our choice of 3, 5, and 10 for outer, balance, and refinement iterations is heuristically chosen based on similar observed average performance from a wide range of graphs; for our software package, iteration counts can be hand-tuned by the user.

REFERENCES

[1] G. M. Slota, K. Madduri, and S. Rajamanickam, "PuLP: Scalable multi-objective multi-constraint partitioning for small-world networks," in *Proc. IEEE Int'l. Conf. on Big Data (BigData)*, 2014.

[2] ——, "Complex network partitioning using label propagation," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. S620–S645, 2016.

[3] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[4] A. George and J. W. Liu, *Computer solution of large sparse positive definite systems*. Prentice Hall, 1981.

[5] H. Meyerhenke, P. Sanders, and C. Schulz, "Parallel Graph Partitioning for Complex Networks," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 28, no. 9, pp. 2625–2638, 2017.

[6] G. Karypis and V. Kumar, "A parallel algorithm for multilevel graph partitioning and sparse matrix ordering," *Journal of Parallel and Distributed Computing*, vol. 48, no. 1, pp. 71–95, 1998.

[7] M. Hanai, T. Suzumura, W. J. Tan, E. Liu, G. Theodoropoulos, and W. Cai, "Distributed edge partitioning for trillion-edge graphs," *Proceedings of the VLDB Endowment*, vol. 12, no. 13, pp. 2379–2392, 2019.