Lock-free Vertex Clustering for Multicore Mesh Reduction

NIMA FATHOLLAHI, University of Victoria, Canada
SEAN CHESTER, University of Victoria, Canada

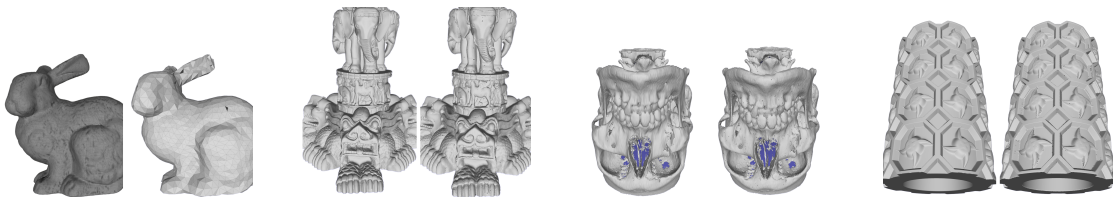


Fig. 1. Mesh reduction applied to triangle meshes to reduce the number of vertices and triangles while maintaining quality

Modern data collection methods can capture representations of 3D objects at resolutions much greater than they can be discretely rendered as an image. To improve the efficiency of storage, transmission, rendering, and editing of 3D models constructed from such data, it is beneficial to first employ a mesh reduction technique to reduce the size of a mesh. Vertex clustering, a technique that merges close vertices together, has particularly wide applicability, because it operates only on vertices and their spatial proximity. However, it is also very difficult to accelerate with parallelisation in a deterministic manner because it contains extensive algorithmic dependencies.

Prior work treats the non-trivial clustering step of this process serially to preserve vertex priorities, which fundamentally limits to mid-single digits the acceleration rates that are possible for the process overall. This paper introduces a novel lock-free parallel algorithm, P-Weld, that exposes parallelism with a graph-theoretic lens that iteratively peels away layers of a mesh that have no remaining dependencies. Concurrent updates to shared data are managed with a linearisable sequence of atomic instructions that exactly reproduces the serial clustering. The resulting parallelism and improved spatial locality yield a 3.86× speedup on a standard 14-million vertex mesh and a 2.93× speedup on a 400-million vertex LiDaR point cloud covering the city of Vancouver, Canada, relative to a popular open source library.

CCS Concepts: • **Theory of computation → Shared memory algorithms**; • **Computing methodologies → Mesh geometry models**; • **Information systems** → *Clustering*.

Additional Key Words and Phrases: lock-free algorithms, mesh simplification, spatial clustering, multi-core parallelism

ACM Reference Format:
Nima Fathollahi and Sean Chester. 2023. Lock-free Vertex Clustering for Multicore Mesh Reduction. In *SIGGRAPH Asia 2023 Conference Papers (SA Conference Papers '23), December 12–15, 2023, Sydney, NSW, Australia.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3610548.3618234

1 INTRODUCTION

When one collects data to reconstruct a 3D model of an object, it generally makes sense to collect it at the highest resolution available. However, that resolution can be expensive to process and transmit, especially if the discreteness of the image space maps many points on the surface of the object to the same pixels [Low and Tan 1997].

Mesh reduction methods (surveyed by [Talton 2004]) aim to simplify a mesh while still achieving visual complexity reminiscent of the original mesh. At a high-level, there are three main approaches. Vertex decimation [Schroeder et al. 1997] deletes single vertices and re-tessellates the resulting hole. Edge contraction [Hoppe et al. 1993] contracts single edges into vertices and updates all edges incident to that. In contrast, vertex clustering [Rossignac and Borrel 1993] reduces a mesh by globally merging all vertices within a pre-defined range, ϵ. It supports meshes of arbitrary topological structure and, as we will show in this paper, can be effectively and losslessly parallelised on multi-core architectures.

The vertex clustering algorithm, illustrated in Figure 2, iterates vertices sequentially. For each unclustered vertex v that it encounters, it constructs a new cluster with all other unclustered vertices within a Euclidean range of ϵ of v. The sequential processing is critical, because it adds determinism to the clustering and also because the vertices are ordered by a pre-computed visual importance [Low and Tan 1997], ignoring which significantly distorts the mesh. However, this ordering quite obviously introduces algorithmic dependencies that challenge parallelisation.

One can attempt to trivially parallelise the algorithm by voxelising the space and distributing voxels to threads [Lindstrom 2000; Rossignac and Borrel 1993], but there is no assurance that workload will be balanced. One can segment the input mesh among threads [Mousa and Hussein 2021], but identifying independent parts is not straightforward. One can index voxels in an octree and assign leaf cells to threads [Decoro and Tatarchuk 2007] to address workload balance and achieve good performance. However, all of these approaches are heuristic and miss non-trivial and cascading relationships near partition boundaries. To our knowledge, this paper proposes the first *exact* parallel algorithm for vertex clustering.

Vertex clustering is ubiquitously supported in computer graphics software. Unity [Unity Technologies 2023] refers to this as `WeldVertices`; Blender [Blender 2023] provides `MergeByDistance`; Modo [The Foundry 2023] contains the function `VertexMerge`; and Open3D [Zhou et al. 2018] supports it as `MergeCloseVertices`. We use the open source Open3D software as a baseline, which has achieved partial parallelisation by decomposing the algorithm into

SA Conference Papers '23, December 12–15, 2023, Sydney, NSW, Australia

ACM ISBN 979-8-4007-0315-7/23/12…$15.00
https://doi.org/10.1145/3610548.3618234

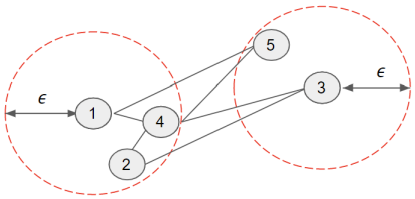


Fig. 2. An example of mesh reduction using vertex clustering method based on a fixed ϵ. A cluster is formed with vertex 1 and its neighbours. The next unclustered vertex is 3, which forms a second cluster with its neighbours.

two steps: one that builds an adjacency list in parallel using a spatial index, and one that performs a serial clustering over that graph.

Although the construction of an adjacency list is the more expensive step, the clustering is non-trivial, requiring 20-30% of execution time and imposing an absolute lower bound of 2.7 seconds on a standard Stanford 3D triangle mesh model in our experiments.

To parallelise this phase, we investigate the properties of the constructed spatial graph to expose more parallelism. We then propose a method that independently maps vertices onto clusters using atomic instructions to carefully synchronise global state. Once all threads vote that the mapping is complete, we perform a reduction phase to generate new clusters. We formally prove correctness of this lock-free approach by demonstrating that updates to the global mapping state are linearisable [Herlihy and Wing 1990].

Our experiments demonstrate a roughly 4× improvement to vertex clustering when provided with an ϵ-independent spatial index, driven by a $20-40\times$ improvement to the previously serial clustering phase, when run on a server with 2×16 cores. Some improvement arises before parallelism due to improved locality.

Motivated by our city-scale remote sensing project, we further stress the algorithms with an aerial LiDAR point cloud that contains over 400 million vertices. Given the natural error in instrumentation and the advantage that vertex clustering is independent of any actual mesh, it is reasonable to reduce a point cloud prior to 3D reconstruction. We find that, even for minor reductions in vertex count, our proposed algorithm can reduce clustering latency by over one minute (over 3×) relative to the partially parallelised baseline.

The paper proceeds by describing and revisiting the vertex clustering algorithm to more easily expose parallelism (Section 2); introducing our novel, exact, multi-core algorithm, P-Weld (Section 3); conducting a comprehensive experimental analysis of P-Weld relative to Open3D (Section 4) and concluding the paper (Section 5).

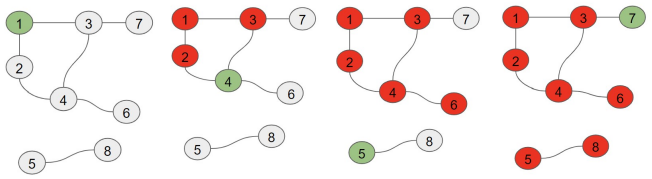
2 (PARTLY) SERIAL VERTEX CLUSTERING

Algorithm 1 (Serial-Weld or simply S-Weld) describes vertex clustering as in Open3D. ([Low and Tan 1997] describe more steps, but they are orthogonal to this work.) S-Weld first (Lines 1-2) constructs a new *spatial graph* over the same set of vertices as the input mesh, but where an edge exists between vertices u and v iff the L_2 Euclidean distance between u and v is $\leq \epsilon$. Next (Lines 3-12), it iterates through vertices by decreasing visual importance and constructs for each unclustered vertex, u, a new cluster with u and all its unclustered neighbours in the spatial graph. A representative vertex is

ALGORITHM 1: S-Weld

Input : mesh vertices ordered by grade, mesh triangles, $\epsilon \geq 0$
Output: reduced and transformed vertex set, transformed triangles

```
1  for u ∈ vertices in parallel with spatial index do
2  |  create adjList[u] ← {v | v ∈ vertices ∧ dist(u, v) ⩽ ϵ};
3  for i = 1; i ≤ numVertices; i = i + 1 do
4  |  u ← vertices[i];
5  |  if u is not clustered then
6  |  |  mark u as clustered w/ u; // new auto-increment cluster ID
7  |  |  represent ← u;
8  |  |  for v ∈ adjList[u], u ≠ v do
9  |  |  |  if v is not clustered then
10 |  |  |  |  mark v as clustered with u;
11 |  |  |  |  shift position of represent to new centre of mass;
12 |  |  newVertices.append(represent);
13 for t ∈ triangles do
14 |  replace vertex IDs of t with new representatives' IDs using map
```



(a) $u = 1$ first (b) $u = 4$ second (c) $u = 5$ third (d) $u = 7$ last

Fig. 3. Illustration of the S-Weld serial clustering process reaching Line 6 of Algorithm 1 on a toy spatial graph. Green vertices represent the centroids of each new vertex; red vertices are already clustered when visited on Line 5.

constructed on-the-fly on Line 11. Finally (Lines 13-14), the mesh is updated by replacing edge endpoints with corresponding clusters.

Spatial Graph Construction The set of edges in the spatial graph is constructed in parallel as an adjacency list: a vector of length $|V|$ in which element i is an unsorted vector of the ids of all vertices within a range of ϵ of vertex v_{i+1}. In order to identify the nearest neighbours within a given radius in $\mathbb{R}^3$, Open3D uses the KDTree FLANN algorithm [Blanco and Rai 2014], a modified version of the FLANN (Fast Library for Approximate Nearest Neighbors) library [Muja and Lowe 2009] that returns exact results. The KDTree FLANN is thread-safe on read: threads can use the same KDTree to populate the adjacency list concurrently. Reusable, we assume it as an input.

Cluster Determination Figure 3 illustrates this step on a toy graph.

Each vertex is visited in descending order of visual importance (Lines 3-4). If already clustered, it is skipped (Line 5). Otherwise, it initiates the next new cluster (Lines 6-7) with itself and its as-yet-unclustered neighbours in the spatial graph (Lines 8-11). An unordered map is maintained to quickly retrieve later the cluster of a vertex. The centre of mass (or, equivalently, some other notion of cluster representative) for the new vertex that will represent the cluster is adjusted as new vertices are added (Line 11). Clearly, the number of output vertices is determined by how often Line 6 is reached, which itself depends on the structure of the spatial graph.

Retriangulation As final clean-up, facets are reduced by converting the vertices of each triangle into their new representatives with the map constructed during cluster determination (Lines 13-14).

2.1 Observations and Algorithmic Intuition

It is clear why S-Weld is difficult to parallelise without distorting the output mesh: The strict ordering on computation fundamentally affects which and how many vertices initiate new clusters. It also affects which neighbours should be added to which clusters and how the new vertices representing those clusters are repositioned. Ignoring this changes both the number of and the position of vertices in the output mesh. Moreover, the cluster determination phase is linear in the size of the edge set in the spatial graph. We refer to this algorithm as "partly" serial, because only the spatial graph construction is parallel and the serial portion of code is notable (c.f., Section 4). Retriangulation seems to be embarrassingly parallel, but it does not accelerate with the addition of threads in this design.

The challenge to parallelising the difficult cluster determination phase of S-Weld is that we do not know *a priori* which vertices are what we call *centroids*: those for which S-Weld will reach Line 6 of Algorithm 1. If these were all known, we could immediately initiate and populate all clusters as separate parallel tasks and need only to manage competition among threads for selecting which non-centroid vertices belong to which clusters.

Recall the example from Figure 3 and consider v_4: we cannot know if it initiates a cluster until both v_2 and v_3 are processed and we cannot know if v_2 or v_3 will initiate a cluster until we process v_1. These relationships can cascade as long as the diameter of the spatial graph. In fact, conveniently, the spatial graph is exactly the dependency graph for this problem. In contrast, consider vertices v_1 and v_5: we can know that they initiate clusters as soon as we see that they have no neighbours of lower ID, i.e., have no dependencies.

This suggests a possible approach: there exist some vertices whose cluster does not depend on other vertices and therefore are certainly centroids; moreover, their neighbours are certainly not centroids.

3 EXACT PARALLEL VERTEX CLUSTERING

Algorithm 2 presents Parallel-Weld (or simply P-Weld), our parallelisation of S-Weld; Figure 4 illustrates it on a spatial graph. Broadly, it follows a map-reduce style of first mapping vertices to correct centroids, then reducing the mapped values to repositioned vertices.

More specifically, the mapping phase (Lines 5-15) is a convergence process in which parallel tasks correspond to vertices. Once a vertex v has no unresolved dependencies (Line 9), we (conceptually) set the corresponding task to *active* and have it decrement the dependency count for all of v's neighbours (Lines 11 and 14). If v identifies itself as a centroid because its current estimate is still itself (Line 12), it updates the centroid estimates of its neighbours to the lesser of v and their current estimates (Line 13). In this way, a task becomes active after it has been correctly clustered by its predecessors. Convergence occurs when all tasks are marked as complete (Lines 10 and 14-15).

The number of unresolved dependencies for vertex u is initialised during spatial graph construction as the in-degree of u (Line 3). The reduction phase in P-Weld makes one serial pass over the map to compress the range of vertex ids (Lines 16-19), then repositions

ALGORITHM 2: P-Weld

Input : mesh vertices ordered by grade, mesh triangles, $\epsilon \geq 0$
Output: reduced and transformed vertex set, transformed triangles

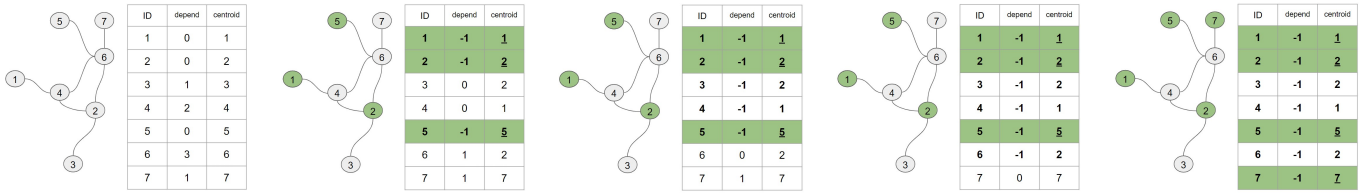
```
1  for u ∈ vertices in parallel with spatial index do
2      adjList[u]  ← {v | v ∈ vertices ∧ dist(u,v) ⩽ ϵ ∧ u < v};
3      depend[u]   ← |{v | v ∈ vertices ∧ dist(u,v) ⩽ ϵ ∧ u > v}|;
4      centroid[u] ← u;
5  while shouldContinue, i.e., at least one thread votes not to halt do
6      sync barrier
7      shouldContinue ← false // thread-local copy reduced on Line 5
8      for u ∈ vertices in parallel do
9          if depend[u] = 0, i.e., u is now a source then
10             decrement depend[u] to mark it as done;
11             for n ∈ adjList[u] do
12                 if centroid[u] == u, i.e., u is a centroid then
13                     centroid[n] ← min(centroid[n], u) CAS;
14                 atomically decrement depend[n];
15                 shouldContinue ← true;
16 for u ∈ vertices do
17     if u is a centroid then
18         Create a cluster and assign it an auto-increment ID
19     Add u to the cluster created by its centroid, centroid[u]
20 for c ∈ clusters in parallel do
21     Calculate the representative and append it to newVertices
22 for t ∈ triangles in parallel do
23     replace vertex IDs of t with their new representatives' IDs
```

vertices (Lines 20-21) and retriangulates the mesh (Lines 22-23) in a data-parallel fashion. Section 3.2 describes an async improvement.

Observe that tasks have writes that are not thread-local, namely decrementing the dependency counts of neighbour vertices (Line 14) and also improving their centroid estimate (Line 13), exemplified in Figure 11. The decrements are critical, because they determine both when to terminate the centroid determination and whether tasks should be active. This is managed with intricate sequencing.

A task i becomes active once all other tasks on which it depends announce that it can be active by collectively decrementing $depend[i]$ down to 0. An active task i immediately signals that it is complete by decrementing $depend[i]$ down to -1. It then broadcasts updates to centroid estimates for all its neighbours n before notifying them that they can become active by decrementing $depend[n]$. Because of the sequencing of these reads and writes, task i cannot begin until all updates to its centroid estimates have been completed. Task i votes for another round of convergence if it has neighbours (Line 15). Decrements use atomic fetch-and-add instructions with relaxed memory constraints, as only the total number of writes, not their order, is important (except the last one, which algorithmically cannot be initiated until after the others anyway).

S-Weld, given its iteration order, directly assigns to each vertex n the *first* centroid, i.e., smallest centroid vertex ID, that neighbours n. We are able to break this ordering by assigning the *minimum* vertex ID of all centroids that visit n (Lines 12-13). We manage the updates to the centroid estimates with an atomic compare-and-swap (CAS) loop. If centroid u neighbours n, it takes a snapshot of $centroid[n]$. If that snapshot is less than u, there is nothing to be done and we break the CAS loop. Otherwise, we perform an atomic compare-and-swap



(a) P-Weld initialisation (b) v_1, v_2, and v_5 are active (c) v_3 and v_4 are active (d) only vertex v_6 is active (e) only v_7 is active

Fig. 4. Walkthrough of P-Weld's centroid determination on a toy spatial graph with a trace of data structures *depend* and *centroid*. Each sub-figure corresponds to a while loop iteration. Green vertices and table rows are centroids discovered so far. Vertex v_i becomes active when $depend[i] = 0$.

to update $centroid[n]$ to u if the snapshot is still valid. If that fails, we loop back and take a new snapshot. This CAS loop cannot fail more than $depend[n] - 1$ times, because each failure indicates that some other thread has successfully reduced $depend[n]$ on Line 14 and therefore beat this thread to complete the CAS loop on Line 13. The ABA problem common to CAS does not apply because the value monotonically decreases. This use of CAS also guarantees that centroid estimates have been synchronised to main memory before the atomic decrements are initiated.

3.1 Algorithm Correctness

We prove correctness in three steps: 1) run on a single thread, P-Weld produces the same result as S-Weld; 2) run in parallel, the algorithm has *liveness*, i.e., at least one task completes on every iteration of convergence; and 3) run in parallel, the algorithm has *safety*, i.e., no task completes based on incorrect or incomplete information.

Lemma 3.1 (S-Weld Equivalence). *When run with one thread, P-Weld and S-Weld produce the same result*

Proof. We prove this by induction on program state: both algorithms begin the same way and induce the same state transitions.

Let Line 8 of P-Weld iterate through vertices serially in the same order as S-Weld and consider v_1. In S-Weld, neither it (Line 5) nor any of its neighbours (Line 9) have been clustered; so, they are all marked as clustered with v_1 (Lines 6 and 10). In P-Weld, $depends[1] = 0$ and $centroid[1] = 1$ by initialisation (Lines 3-4). Task 1 is therefore active on Line 9 and passes the Line 12 condition. All neighbours n have $centroid[n]$ set to 1 (Line 13). This value can never change, because 1 is the minimum in the dataset, and therefore, all n will fail their respective Line 12 conditions–i.e., they have been marked as clustered/non-centroids. So, both S-Weld and P-Weld have created a cluster of v_1 and its neighbours without modifying other vertices.

For the inductive step, consider v_i with all $v_1, \ldots, v_{i-1}$ processed equivalently between S-Weld and P-Weld. If v_i is not a centroid, S-Weld skips it on Line 5 and P-Weld skips having any effect on other vertices on Line 12. v_i is not re-clustered because Line 6 is not reached in S-Weld and P-Weld tasks only alter the centroid estimates of other vertices. If, instead, v_i is a centroid, S-Weld updates all as-yet-unclustered vertices to cluster i (Lines 8-10). P-Weld updates the centroid estimate of *all* neighbours to the min of their current estimate and i; however, as all $j < i$ have been processed correctly, Line 13 only changes the centroid estimate if no prior vertex v_j already marked the neighbour as clustered. Therefore, S-Weld and P-Weld have made the same changes to cluster membership.

As a final note, P-Weld compresses cluster ids and repositions vertices in straightforward post-processing to match S-Weld. □

Lemma 3.2 (Liveness). *On every iteration of the P-Weld while loop, at least one task completes (and therefore the algorithm terminates)*

Proof. We noted already that the CAS cannot loop indefinitely. Assume for the sake of contradiction that no task completes on this iteration, i.e., $depend[u] \neq 0, \forall u$, and the algorithm has not terminated, i.e., $\exists v, depend[v] > 0$. Let $\hat{v}$ be the smallest such v; then all vertices $v_i < \hat{v}$ had $depend[i] < 0$ prior to this iteration. But that requires each task i to reach Line 10, which implies that they reached Line 14. Since they have $\hat{v}$ in their adjacency list, per Line 2, at least $depend[\hat{v}]$ vertices must have atomically decremented $depend[\hat{v}]$ on previous iterations. That value cannot be strictly positive. □

Lemma 3.3 (Safety). *Global state is correct for all complete tasks*

Proof. Global state is maintained in two vectors: *depend* and *centroid*, which are both initialised without inter-thread communication. A vertex u is complete when it broadcasts $depend[u] = -1$. Prior to this linearisation point, it only reads $depend[u]$ and is not active until all lower ID neighbours have atomically decremented $depend[u]$ (Line 14); they cannot do this until they complete updates to $centroid[u]$. Therefore, when u broadcasts $depend[u] = -1$, $centroid[u]$ has considered all lower ID centroids. □

3.2 Implementation Considerations and Improvements

A few details can improve P-Weld performance. Using a consistent, static schedule for the parallel for loops increases affinity between data and cores, particularly the adjacency lists. This improves temporal locality. It also maximises order in the computation so that, within the workload of an individual thread, execution order resembles that of S-Weld as much as possible.

Accelerating the serial reduction is limited by data movement. However, the subsequent code does not depend on it; so, it can be run asynchronously on one thread while the remaining threads complete the repositioning and retriangulation. We experiment with a P-Weld-Async version that tracks which centroids are discovered by which threads, uses a parallel prefix sum [Blelloch 1990] to determine write locations for each centroid, and completes the memory-intensive repositioning asynchronously.

Table 1. Datasets used in the experiments below and merge radii (ϵ values) corresponding to 0.1%, 1.0%, 10%, and 50% mesh reductions

Dataset name	Abbr	# vertices	# triangles	Merge radii (ϵ)
Stanford Bunny	*Bun*	35,947	69,451	[2.800e-4, 4.280e-4, 9.965e-4, 1.209e-3]
Vellum Manuscript	*Manu*	2,152,840	4,305,679	[6.050e-2, 9.240e-2, 0.1000022, 0.10054]
Thai Statue	*Stat*	5,000,000	10,000,000	[5.000e-2, 7.500e-2, 0.10023, 0.1795]
Lucy	*Lucy*	14,027,872	28,055,742	[8.000e-4, 7.000e-3, 7.200e-2, 0.5009]
Vancouver LiDaR	*Van*	461,200,229	0	[0.014, 0.028, 0.067, 0.18]

4 EMPIRICAL VALIDATION

In this section, we evaluate the performance of our proposed P-Weld algorithms. The mesh that we generate, recall, is identical to prior work (e.g., [Zhou et al. 2018]); the contribution is in being able to do this in parallel. The reduced meshes are rendered in Figure 1 to illustrate visual quality, but we focus on algorithmic aspects here.

4.1 Experiment Design

4.1.1 Software. [1] We compare three algorithms: S-Weld (state-of-the-art), P-Weld (our proposed technique), and P-Weld-Async (an optimisation to P-Weld that runs the only non-allocation-related sequential code in P-Weld asynchronously).

To prepare *S-Weld*, we isolate the `MergeCloseVertices()` function from version (0.17) of the Open3D open source library [Zhou et al. 2018], as well as all of its dependencies, to create a stand-alone C++ application. This includes its spatial data structure (KDTree FLANN) whose construction we isolate in a separate method and which we pass as an additional parameter to `MergeCloseVertices()` by const-ref. We wrap this function call with a `std::chrono` timer.

We implement *P-Weld* and *P-Weld-Async* as two C++17 functions with the same signature as S-Weld that are also wrapped with a timer. The OpenMP library is used for parallelism and asynchronicity within the fork-join model is implemented with a `single nowait` region and a decremented thread count in the subsequent `omp for`.

We calculate the average time of twenty function invocations. Because these methods perturb a mesh in-place, each cold-cache trial is on a new copy of the mesh but reuses the spatial index.

4.1.2 Datasets. For the main experiments, we use standard, benchmark triangle meshes from the Stanford repository.[2] These are described in Table 1. The four meshes are selected to vary the size of the mesh to facilitate a scalability study. We determine via manual binary search the values of ϵ that lead to a consistent reduction in the number of vertices in the mesh so that we can standardise experiments across meshes. In general, we focus on the largest dataset, *Lucy*, and the two extreme mesh reduction rates, 0.1% and 50%. To study the generalisability of our results, we use a subset of large meshes from the Thingi10K repository [Zhou and Jacobson 2016].

We also evaluate performance on a 400-million vertex point cloud taken by concatenating ten adjacent tiles of the Open Vancouver dataset [City of Vancouver 2019], centred around Canada Place (a region of high density). Though using a point cloud, not a mesh, this more extreme experiment demonstrates that vertex clustering can deterministically reduce a point cloud prior to 3D reconstruction.

[1] Available at: https://github.com/nimaft97/parallel-vertex-clustering
[2] http://graphics.stanford.edu/data/3Dscanrep/

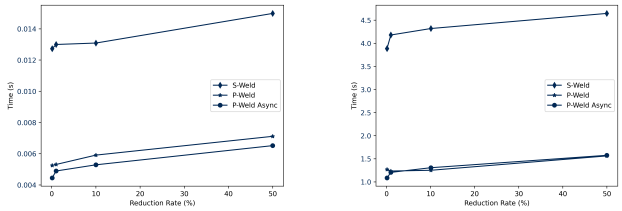


Fig. 5. Scalability with respect to mesh reduction rate (left: *Bun*; right: *Lucy*)

4.1.3 Environment. Experiments are conducted on a server running Gentoo 2.6 with 2 × Intel(R) Xeon(R) Gold 5218 processors @ 2.30GHz, providing a total of 32 cores (64 hyper-threads). Software is compiled using `gcc 9.3.0` with architecture-specific optimisations enabled. Meshes and point clouds are saved to a file and rendered on a 2022-vintage commodity laptop using MeshLab.

4.2 Results and Analysis

4.2.1 Overall Comparison. We begin by simply studying execution time relative to inputs for each algorithm. Figure 5 shows the effect of the merge radius, ϵ, on performance for the smallest and the largest dataset. The radii are set for each dataset such that they eliminate 0.1%, 1%, 10%, and 50% of vertices, respectively. Each algorithm uses as many hardware threads as it can ($t = 64$), except that P-Weld does not use hyper-threading on *Bun* (explained later). Additional datasets (*Manu* and *Stat*) are shown in Figure 10.

Observe first that the trends are quite consistent. The P-Weld algorithms provide about a 4× improvement relative to S-Weld across a range of mesh sizes and reduction rates. As more of the mesh is reduced, there is a gently-sloped, linear degradation in execution time common to all methods. This mostly arises from the increased cost of the spatial range queries, though it also increases the number of neighbours that each vertex visits during clustering.

As can be observed by the y-axis values, the larger meshes take longer to reduce, as predicted by the asymptotic complexity of the algorithms. The *Lucy* dataset requires about 200× longer than *Bun*. Handling larger meshes can clearly benefit from acceleration.

The asynchronous repositioning of vertices generally improves performance by about 15%, illustrating that the latency can be largely hidden. However, this is sensitive to the reduction rate. Repositioning iterates over deleted vertices, while mesh retriangulation iterates over retained vertices. As more of the mesh is reduced, the relative cost of these methods shifts. When half of the vertices are removed, retriangulation can be too fast to hide the repositioning latency.

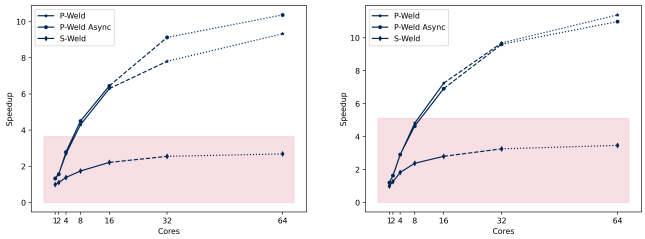


Fig. 6. Speedup relative to running single core (*Lucy*, .1% & 50%). The red shaded region indicates theoretically possible speedup for S-Weld

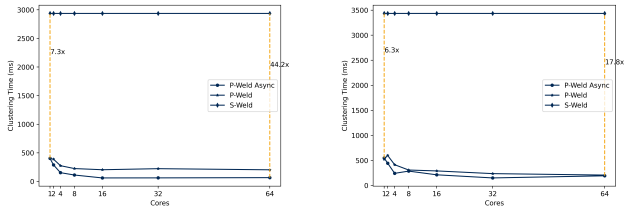


Fig. 7. Parallel performance of clustering phase (*Lucy*, .1% & 50%)

The subsequent experiments add more nuance to this analysis.

4.2.2 Parallel Performance. In the prior experiment, we assumed the use of as many cores as is useful. Here, we study how effectively the algorithms make use of those additional cores. Figure 6 shows the ratio of execution time for each algorithm run on t versus 1 cores, using the larger *Lucy* dataset and the two extreme values of ϵ. Other datasets are shown in Figure 12. The red shaded region shows speedups within the theoretical limit to parallel acceleration of S-Weld, calculated as the ratio of total single-threaded execution time to that of S-Weld's serial region. We can see that it is higher for the larger ϵ value because the construction of a denser spatial graph takes more time, thereby increasing the ratio of time spent in the parallel region relative to the sequential clustering phase.

The speedup of both P-Weld versions exceeds this theoretical limit of S-Weld already with 4-8 cores. This is consistently observed across datasets and reduction rates in Figure 12 as well, except for the small *Bun* dataset (explained later). By 16 cores, the P-Weld algorithms more than double the parallel speedup that S-Weld can achieve on *Lucy*. On smaller ϵ, the async version outperforms with the addition of a second processor ($t = 32$), likely because it exposes an extra form of parallelism that is more noticeable as execution times decrease. On *Lucy*, we see more improvement with hyper-threading (i.e., on 64 logical cores), suggesting that the computation was still memory-bound and benefitted from an ability to hide memory latencies. As the mesh size decreases, so too does the benefit of hyper-threading; on the small *Bunny* mesh, it materially degrades performance.

In Figure 7, we look closer at our novel contribution, i.e., the clustering phase. It shows the speedup, per core count, of the P-Weld versions relative to the always-single-threaded S-Weld. This data is specific to *Lucy*; other datasets can be seen in Figure 13.

Table 2. Hardware event counts for single-threaded execution

Metric	**Bun (.1% reduction)**		**Lucy (.1% reduction)**	
Instructions	129.5M	107.8M	64.4B	51.5B
CPI	0.50	0.48	0.56	0.54
Cycles Stalled	11.2M	10.9M	10.1B	7.7B
↛ Any Mem	7.1M	7.4M	6.9B	4.5B
↛ L1D	2.0M	2.3M	3.4B	1.6B
↛ L2	1.6M	1.7M	2.5B	0.9B
Algorithm	**S-Weld**	**P-Weld**	**S-Weld**	**P-Weld**

First, observe that there is already a 6-7× improvement on a single core: this is investigated in the next experiment. On 64 cores, we observe a 44× speedup (including the earlier 6-7×) when ϵ is small and a 17× speedup when ϵ is large. At $\epsilon = .0008$, only two iterations of the while loop are required to converge, but at $\epsilon = .5009$, four iterations are required. On denser spatial graphs, we expect fewer source nodes and for the diameter to increase, correlating with an increased number of rounds required to converge.

4.2.3 Single-Threaded Performance. In the previous experiment, we observed that P-Weld outperforms S-Weld already on a single core; for the clustering phase, the difference is 6.3 − 7.3× on the larger *Lucy* dataset. We investigate in Table 2 whether this can be explained by locality by measuring cache-related hardware event counters with the Linux `perf` utility. Specifically, we measure instructions retired (as a measure of work), cycles per instruction (CPI) as a measure of implementation performance, and the number of cycles stalled while requests are outstanding. Note that our machine has an inclusive cache; so, an outstanding data request to L2 includes an outstanding request to the L1 data cache.

We obtain values for these metrics by running the entire application, including spatial index construction, once and then subtracting from that the values from a run that excludes a call to the P-Weld or S-Weld function. Comparing across program launches introduces some imprecision, but the counters themselves are already imprecise and we only report two to four digits of precision.

Observe first that S-Weld (on one core) generally performs 20-25% more instructions than P-Weld, evidence that P-Weld is work-efficient. CPI is consistent for both algorithms and datasets, with roughly two instructions retired every cycle; so, instruction counts can be compared between algorithms. Observe also that on the smaller dataset (*Bun*), there is not much difference in stalls between the algorithms; however, on the larger dataset (*Lucy*), S-Weld experiences more than twice as many L1 data cache stalls than P-Weld and over 70% of those stalls appear to access LLC/L3 or main memory. Since the range queries are performed similarly in both algorithms, the differences probably arise mostly from the clustering phase.

This comes as some surprise, because P-Weld requires two iterations of the vertex set to converge in this case and also synchronises in main memory with atomic instructions. However, to manage locality and concurrent access, P-Weld exclusively uses vectors and static execution schedules. This ensures that threads are mapped consistently to the same data elements in separate parallel regions. On a single core, this appears to reduce instruction counts and cache

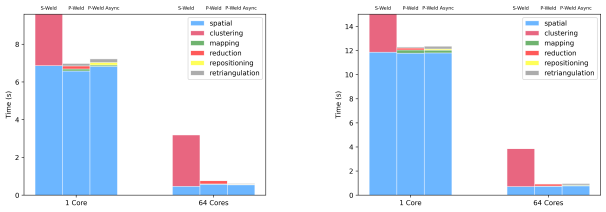


Fig. 8. Time spent in phases of computation (*Lucy*, .1% & 50%)

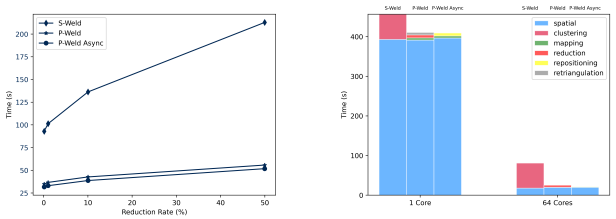


Fig. 9. Scalability and profile for LiDAR point cloud reduction (.1%)

misses because the data structure is more compact and nearby data accesses are not hashed to different cache lines. The effect is not observed on *Bunny*, because its 36K vertices only require 850KB anyway and this machine has 1.375MB of cache per core.

As a secondary note, although the retriangulation phase is embarrassingly parallel for both algorithms, parallelisation only accelerates it in P-Weld. Table 2 suggests why: this stage is memory-bound in S-Weld but those latencies can be hidden somewhat in P-Weld, due to the different approaches to mapping vertices to clusters.

4.2.4 Fine-Grained Profiling. Vertex clustering consists of a series of steps, some of which are more effectively parallelised. In Figure 8, we provide a break-down of time spent in each step and compare them at $t = 1$ and $t = 64$ (logical) cores. The intent is to see empirically if any sub-components of the computation limit parallelism.

S-Weld is decomposed into two phases: the parallel construction of a *spatial graph* that involves repeated range queries and the allocation and population of an adjacency list; and serial *clustering* that completes everything else. For P-Weld, we furthermore decompose clustering into: parallel *mapping* which assigns a centroid to every vertex; serial *reduction* that combines all vertices into newly repositioned cluster representatives; and parallel *retriangulation* that adjusts mesh edges to fit the new vertices. For P-Weld-Async, the reduction is furthermore decomposed into a parallel prefix sum calculation (not shown; too fast), an asynchronous *repositioning* of vertex centroids, and parallel retriangulation. Due to the asynchronicity, retriangulation is included as part of repositioning time.

Observe first that, despite parallelising the spatial graph construction effectively (the blue bar), S-Weld is inherently limited by the large percentage of time spent in serial clustering (the pink bar), as we noted with the red shaded region in Figure 6. This sets a lower bound of 2.7 seconds that cannot be overcome with parallelism. We also observe that the clustering phase is much faster in P-Weld and P-Weld-Async, even at $t = 1$, as discussed in Section 4.2.3.

P-Weld also has a serial sub-component of clustering, the reduction phase. At $t = 64$, this dominates the remaining execution time for clustering. Of those 200 milliseconds, nearly half are spent on allocation, which is very difficult to address. P-Weld-Async is able to hide some of this latency, and completes the entire clustering in 67 milliseconds. We conclude that the clustering phase no longer limits multi-core parallelism for the vertex-clustering approach to mesh reduction and suspect that further improvements could consider amortising the cost of range queries over repeated mesh reductions.

4.2.5 City-Scale LiDAR Point Cloud Reduction. Our final experiments stress the algorithms with a 460M-vertex point cloud. It disadvantages the P-Weld variants, because there is no retriangulation phase, but represents an interesting use case for heavy reduction.

Figure 9 repeats the experiments of Sections 4.2.1 and 4.2.4 on *Van*, using 64 cores. Observe that P-Weld-Async reduces execution time from over 1.5 minutes to about 30 seconds, relative to S-Weld, at a 0.1% reduction intensity. It outperforms P-Weld by 10%, even without the retriangulation phase to hide the latency of repositioning vertices, as it parallelises the reduction phase better. S-Weld degrades more rapidly with ϵ than on *Bun* or *Lucy*, illustrating its memory-boundedness. When reducing 50% of vertices, the P-Weld variants are over 2.5 minutes faster. This represents a sizeable savings, especially if one wants to run different values of ϵ.

We see again that it is Amdahl's Law that limits S-Weld; the clustering phase that is parallelised in this paper requires one minute in S-Weld and makes up two thirds of its execution time on 64 cores.

4.2.6 Generalisability. Figure 15 shows scalability on 456 meshes from the Thingi10K repository [Zhou and Jacobson 2016]: all meshes with $|V| \geq 100K$ except 41 with which we experience preprocessing errors. We create three overlapping workloads based on mesh size and show distributions of datasets based on observed speedup.

Figure 15a illustrates the percentage of datasets for which P-Weld (left) and P-Weld-Async (right) are slower than S-Weld, less than 20× faster, and more than 20× faster on 32 cores, respectively. P-Weld performs best with async, on large meshes, and with low reduction rates. Figure 15b shows similar results for parallel scalability: over half of datasets see a $\geq 4\times$ speedup when async reduces $< 50\%$.

5 CONCLUSION

Vertex clustering is a fundamental mesh operation ubiquitously supported by computer graphics software. We introduced a novel, lock-free parallel algorithm, P-Weld, that can accelerate vertex clustering on multi-core architectures without introducing heuristic error. Using carefully sequenced atomic operations on global state, P-Weld converges in line with the dependencies of the sequential algorithm that have been explicitly exposed.

In a comprehensive empirical evaluation, we demonstrated that our proposed algorithm can achieve a roughly 4× speedup on 64 logical cores relative to Open3D across a range of datasets and mesh reduction intensities. This is driven by a $20 - 40\times$ speedup in the clustering phase that we novelly parallelise. As the number of vertices reaches hundreds of millions, we showed that the performance improvement offered by P-Weld continues to grow.

REFERENCES

Jose Luis Blanco and Pranjal Kumar Rai. 2014. nanoflann: a C++ header-only fork of FLANN, a library for Nearest Neighbor (NN) with KD-trees. https://github.com/jlblancoc/nanoflann.

Guy E. Blelloch. 1990. *Prefix Sums and Their Applications.* Technical Report CMU-CS-90-190. School of Computer Science, Carnegie Mellon University.

Blender. 2023. Blender Software. https://github.com/blender.

City of Vancouver. 2019. LiDAR 2018. https://opendata.vancouver.ca/explore/dataset/lidar-2018/information/.

Christopher Decoro and Natalya Tatarchuk. 2007. Real-time mesh simplification using the GPU. 161–166. https://doi.org/10.1145/1230100.1230128

Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492. https://doi.org/10.1145/78969.78972

Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. 1993. Mesh optimization. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. 19–26.

Peter Lindstrom. 2000. Out-of-Core Simplification of Large Polygonal Models. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '00)*. ACM Press/Addison-Wesley Publishing Co., USA, 259–262. https://doi.org/10.1145/344779.344912

Kok-Lim Low and Tiow-Seng Tan. 1997. Model Simplification Using Vertex-Clustering. In *Proceedings of the 1997 Symposium on Interactive 3D Graphics* (Providence, Rhode Island, USA) *(I3D '97)*. Association for Computing Machinery, New York, NY, USA, 75–ff. https://doi.org/10.1145/253284.253310

Mohamed Mousa and Mohamed Hussein. 2021. High-performance simplification of triangular surfaces using a GPU. *PLOS ONE* 16 (08 2021), e0255832. https://doi.org/10.1371/journal.pone.0255832

Marius Muja and David Lowe. 2009. Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration. *VISAPP 2009 - Proceedings of the 4th International Conference on Computer Vision Theory and Applications* 1, 331–340.

Jarek Rossignac and Paul Borrel. 1993. Multi-resolution 3D approximation for rendering complex scenes. (01 1993), 455–465. https://doi.org/10.1007/978-3-642-78114-8_29

William Schroeder, Jonathan Zarge, and William Lorensen. 1997. Decimation of triangle meshes. *SIGGRAPH Comput. Graph.* 26 (06 1997), 65–70. https://doi.org/10.1145/133994.134010

Jerry O. Talton. 2004. A Short Survey of Mesh Simplification Algorithms.

The Foundry. 2023. Modo Software. https://www.foundry.com/products/modo.

Unity Technologies. 2023. Unity Software. https://github.com/Unity-Technologies.

Qingnan Zhou and Alec Jacobson. 2016. Thingi10K: A Dataset of 10,000 3D-Printing Models. *arXiv preprint arXiv:1605.04797* (2016).

Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. 2018. Open3D: A Modern Library for 3D Data Processing. *arXiv:1801.09847* (2018).

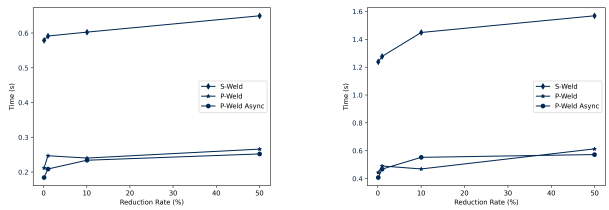
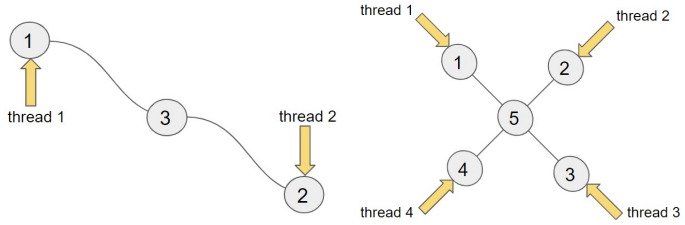


Fig. 10. Scalability with respect to reduction rate (left: *Manu*; right: *Stat*)



(a) Updating centroids of neighbours (b) Decrementing $depend[i]$

Fig. 11. Potential data races addressed by the use of CAS and atomic adds

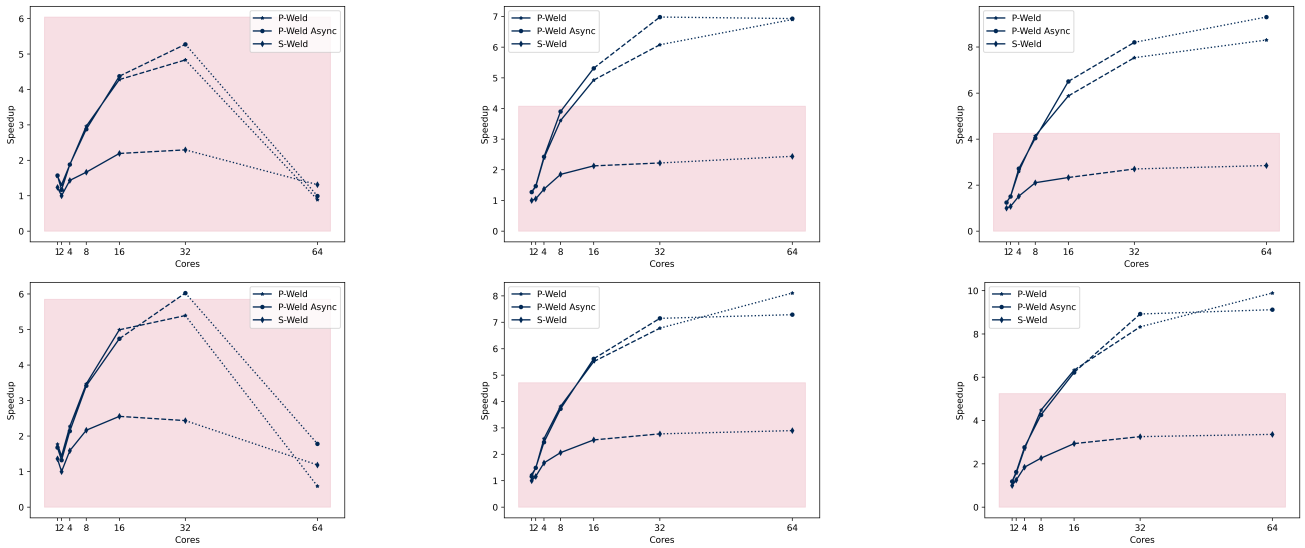


Fig. 12. Speedup relative to running single core (top: .1%; bottom: 50%; left: *Bun*; centre: *Manu*; right: *Stat*)

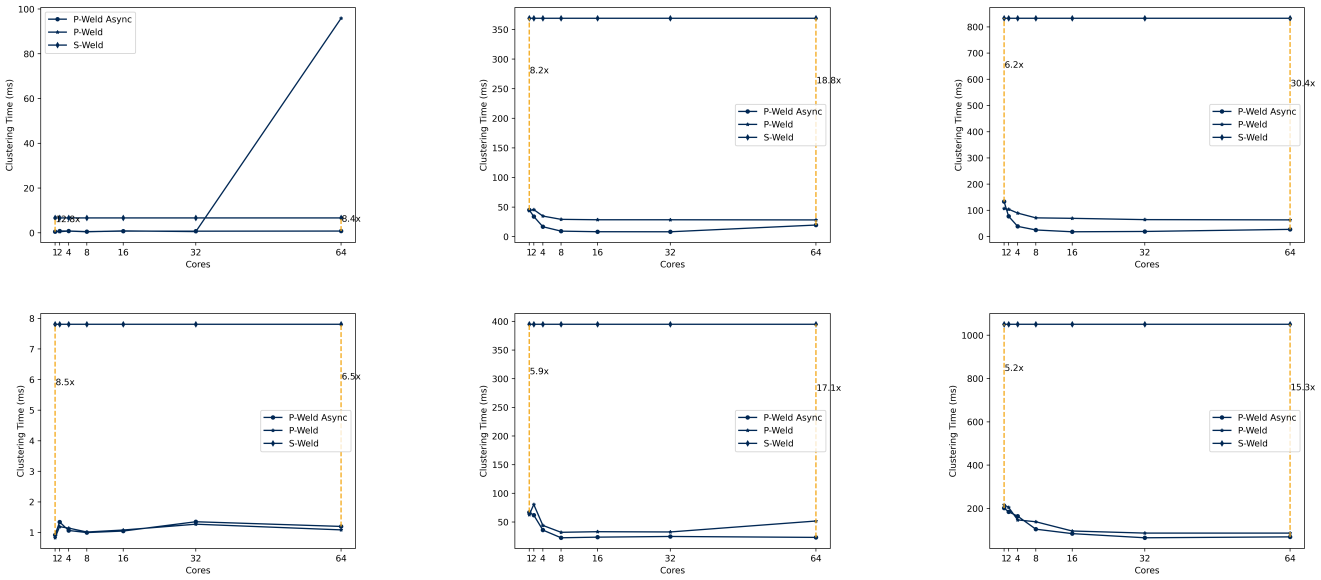


Fig. 13. Parallel performance of clustering phase (top: .1%; bottom: 50%; left: *Bun*; centre: *Manu*; right: *Stat*)

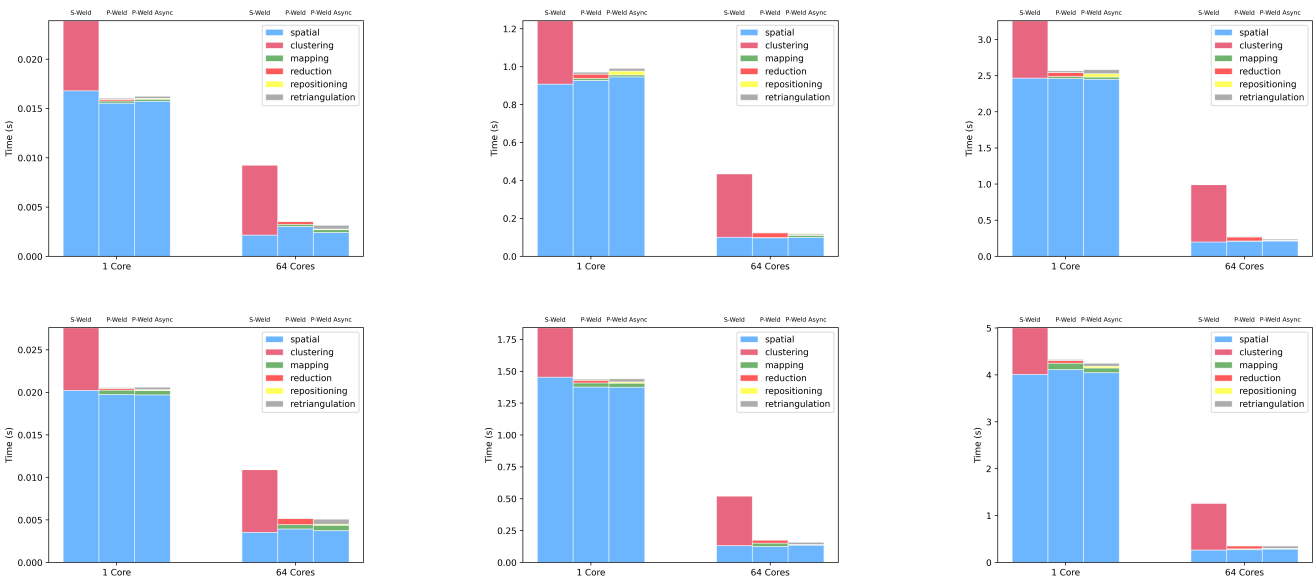
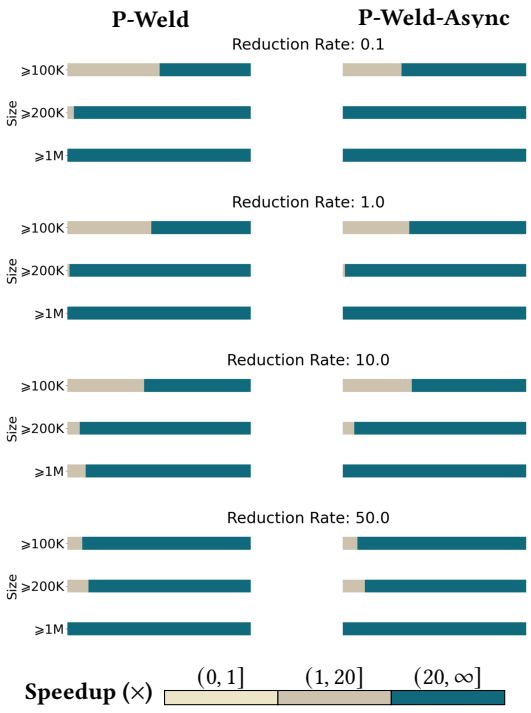
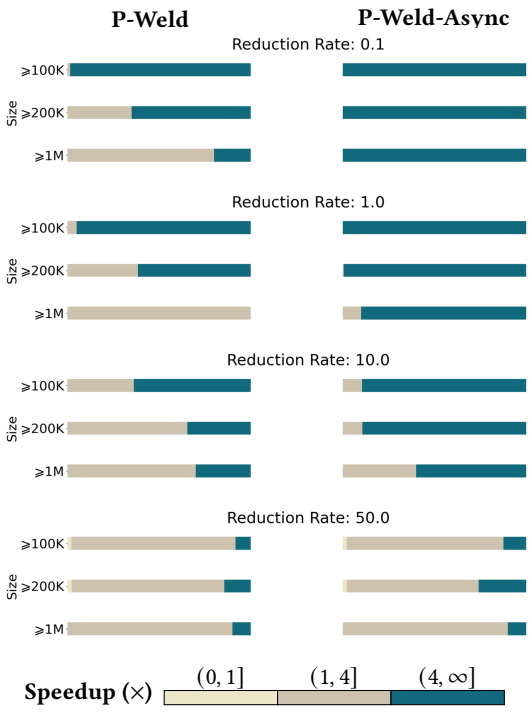


Fig. 14. Time spent in phases of computation (top: .1%; bottom: 50%; left: *Bun*; centre: *Manu*; right: *Stat*)



(a) Speedup of P-Weld and P-Weld-Async on 32 cores in clustering phase relative to S-Weld



(b) Parallel Speedup of P-Weld and P-Weld-Async on 32 cores relative to 1 core

Fig. 15. Percentage of 456 datasets from Thingi repository on which a given speedup range is observed. Each subfigure varies the percentage of mesh vertices that are reduced from 0.1% (top group) to 50% (bottom group). Within each group, we report three subsets of Thingi meshes–those with ≥ 100K, ≥ 200K, and ≥ 1M vertices–456 meshes, roughly half of those, and then the eight largest. Each bar reports the distribution of datasets by the speedup achieved.