

P-Cloth: Interactive Complex Cloth Simulation on Multi-GPU Systems using Dynamic Matrix Assembly and Pipelined Implicit Integrators

Supplementary Material

CHENG LI, Zhejiang University
 MIN TANG, Zhejiang University
 RUOFENG TONG, Zhejiang University
 MING CAI, Zhejiang University, corresponding author,
 JIEYI ZHAO, University of Texas Health Science Center at Houston
 DINESH MANOCHA, University of Maryland at College Park

ACM Reference Format:

Cheng Li, Min Tang, Ruofeng Tong, Ming Cai, Jieyi Zhao, and Dinesh Manocha. 2020. P-Cloth: Interactive Complex Cloth Simulation on Multi-GPU Systems using Dynamic Matrix Assembly and Pipelined Implicit Integrators Supplementary Material. 1, 1 (September 2020), 3 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

1 ALGORITHMS FOR TIME INTEGRATION

Some algorithms used for parallel time integration is described in the following pseudo-code.

Algorithm 1 Work Queue Generation Algorithm used for Pipelined SpMV

```

1: sw: index of the switch
2: offset: offset for sub-vector index
3: lvl: level in the binary tree, 0 for leaf nodes
4: GetGPURange:
5: Returns the indices of the GPU that the given switch intercon-
   nects. For a leaf switch, returns exact 2 GPU indices.
6: GetChildSwitches:
7: Return the indices of 2 child switches of a given switch.
8:
9: // Call GenerateWorkQueues(0, 0, log2n) for the overall
10: // work queues.
11: procedure GENERATEWORKQUEUES(sw, offset, lvl)
12:   // step (1):
13:   // Recursively generate optimal work queues for
14:   // its children.
15:   if lvl != 0 then
16:     sw0, sw1 ← GetChildSwitches(sw)

```

Authors' addresses: Cheng Li, Zhejiang University, licharmy@yahoo.com; Min Tang, Zhejiang University, tang_m@zju.edu.cn; Ruofeng Tong, Zhejiang University, trf@zju.edu.cn; Ming Cai, Zhejiang University, corresponding author, cm@zju.edu.cn; Jieyi Zhao, University of Texas Health Science Center at Houston, jieyi.zhao@uth.tmc.edu; Dinesh Manocha, University of Maryland at College Park, dm@cs.umd.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2020 Association for Computing Machinery.

XXXX-XXXX/2020/9-ART \$15.00

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

```

17:   GenerateWorkQueues(sw0, offset, lvl - 1)
18:   GenerateWorkQueues(sw1, offset, lvl - 1)
19: else
20:   // Each leaf switch interconnects exact 2 GPUs.
21:   G0, G1 ← GetGPURange(sw)
22:   node0 ← (G1, G0 + offset)
23:   node1 ← (G0, G1 + offset)
24:   Push node0 to Q(G0)
25:   Push node1 to Q(G1)
26: end if
27:
28: // step (2):
29: // Transfer minimized amount of data between
30: // its children.
31: range0 ← GetGPURange(sw0)
32: range1 ← GetGPURange(sw1)
33: for G0, G1 ∈ range0, range1 do
34:   node0 ← (G1, G0 + offset)
35:   node1 ← (G0, G1 + offset)
36:   Push node0 to Q(G0)
37:   Push node1 to Q(G1)
38: end for
39:
40: // step (3):
41: // Delegate rest of the work queue generation
42: // tasks to its children.
43: offset0 ← offset + 2lvl
44: offset1 ← offset - 2lvl
45: // An offset is applied so that the work queue is
46: // generated for the delegated sub-vectors.
47:   GenerateWorkQueues(sw0, offset0, lvl - 1)
48:   GenerateWorkQueues(sw1, offset1, lvl - 1)
49: end procedure

```

Algorithm 2 Sparse Matrix Filling Algorithm

```

1: // Index Table Allocating:
2: for each GPUi do in parallel
3:   IndexTablei ← AllocateIndexTable(i)
4: end for
5:
6: // Index Filling:

```

```

7: for each GPUi do in parallel
8:   for Element ∈ AssemblyElements do
9:     rowIdx ← GetElementRowIdx(Element)
10:    colIdx ← GetElementColIdx(Element)
11:    Push colIdx to IndexTablei[rowIdx] // atomic operator
12:   end for
13: end for
14:
15: // Index Compacting:
16: for each GPUi do in parallel
17:   for row ∈ IndexTablei do
18:     RemoveDuplication(row)
19:   end for
20: end for
21:
22: // Value Table Allocating:
23: for each GPUi do in parallel
24:   ValueTablei ← AllocateValueTable(i)
25: end for
26:
27: // Value Filling:
28: for each GPUi do in parallel
29:   for Element ∈ AssemblyElements do
30:     entry ← FindElementEntry(Element)
31:     value ← GetElementValue(Element)
32:     ValueTablei[entry] += value // atomic operator
33:   end for
34: end for

```

2 STITCHING ALGORITHM

The stitching algorithm is described in the following pseudo-code.

Algorithm 3 Stitching Algorithm

```

1: Input: Stitching node pairs NP.
2: Output: Stitched and refined cloth mesh .
3:
4: // Stitching cloth pieces with linking constraints
5: // for all the NP.
6: StitchingCoarsePieces(NP)
7:
8: // Merge the pieces together by merging
9: // all the node pairs NP
10: MergePieces(NP)
11:
12: // Refine the merged cloth mesh to higher resolution
13: // by subdividing.
14: RefinePieces()

```

3 ADDITIONAL BENCHMARKS

We use some complex cloth simulation benchmarks for regular/irregular-shaped cloth simulation (Fig. 1 and Fig. 2).

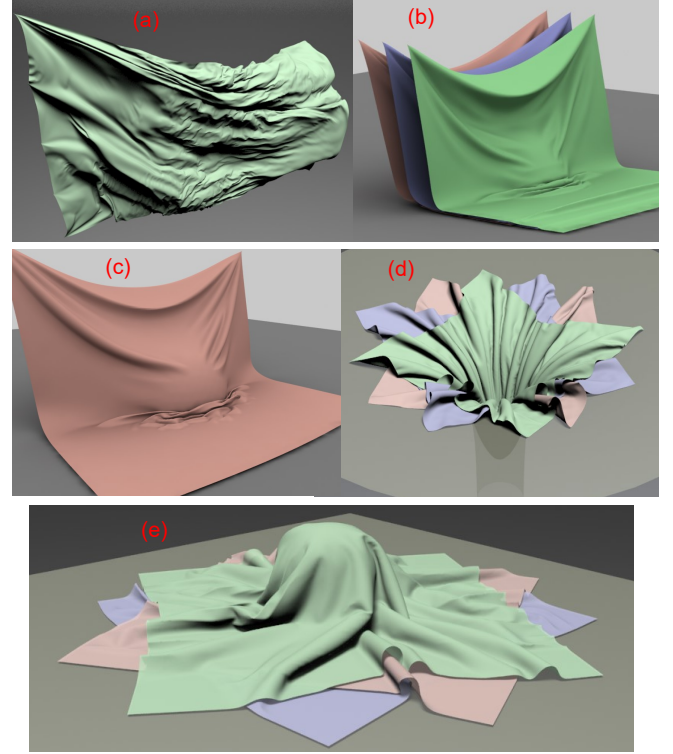


Fig. 1. **Benchmarks:** We use different multi-layered cloth simulation benchmarks ((a)Flag, (b) Sphere, (c) Sphere-1M, (d) Funnel, and (e) Twisting) for evaluation. The mesh complexity varies between 0.5 – 1.65M triangles. P-Cloth can perform cloth simulation at 2 – 5 fps on the 4-GPU workstation. The memory overhead on each GPU is between 4 – 8 GB.



Fig. 2. **Multi-layer Garment Benchmarks:** We used these benchmarks: (a) Miku with 1.33M triangles, (b) Zoey with 569K triangles, (c) Andy with 538K triangles, and (d) Kimono with 1M triangles, for multi-layer garment simulation.