P-Cloth: Interactive Complex Cloth Simulation on Multi-GPU Systems using Dynamic Matrix Assembly and Pipelined Implicit Integrators Supplemetary 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
https://min-tang.github.io/home/PCloth/

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 Supplemetary 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

```
sw: index of the switch
offset: offset for sub-vector index
lvl: level in the binary tree, 0 for leaf nodes
GetGPURange:
Returns the indices of the GPU that the given switch interconnects. For a leaf switch, returns exact 2 GPU indices.
GetChildSwitches:
Return the indices of 2 child switches of a given switch.

// Call GenerateWorkQueues(0, 0, log_2 n) for the overall
// work queues.
procedure GenerateWorkQueues(sw, offset, lvl)
    // step (1):
    // Recursively generate optimal work queues for
    // its children.
    if lvl != 0 then
        sw0, sw1 ← GetChildSwitches(sw)
        GenerateWorkQueues(sw0, offset, lvl - 1)
        GenerateWorkQueues(sw1, offset, lvl - 1)
    else
        // Each leaf switch interconnects exact 2 GPUs.
        G0, G1 ← GetGPURange(sw)
        node0 ← (G1, G0 + offset)
        node1 ← (G0, G1 + offset)
        Push node0 to Q(G0)
        Push node1 to Q(G1)
    end if

    // step (2):
    // Transfer minimized amount of data between
    // its children.
    range0 ← GetGPURange(sw0)
    range1 ← GetGPURange(sw1)
    for G0, G1 ∈ range0, range1 do
        node0 ← (G1, G0 + offset)
        node1 ← (G0, G1 + offset)
        Push node0 to Q(G0)
        Push node1 to Q(G1)
    end for

    // step (3):
    // Delegate rest of the work queue generation
    // tasks to its children.
    offset0 ← offset + 2^lvl
    offset1 ← offset - 2^lvl
    // An offset is applied so that the work queue is
    // generated for the delegated sub-vectors.
    GenerateWorkQueues(sw0, offset0, lvl - 1)
    GenerateWorkQueues(sw1, offset1, lvl - 1)
end procedure
```

Algorithm 2 Sparse Matrix Filling Algorithm

```
// Index Table Allocating:
for each GPU_i do in parallel
    IndexTable_i ← AllocateIndexTable(i)
```

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
https://min-tang.github.io/home/PCloth/, dm@cs.umd.edu.

XXXX-XXXX/2020/9-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

end for

// Index Filling:
for each GPU_i **do in parallel**
  for $Element \in AssemblyElements$ **do**
    $rowIdx \leftarrow$ GetElementRowIdx($Element$)
    $colIdx \leftarrow$ GetElementColIdx($Element$)
    Push $colIdx$ to $IndexTable_i[rowIdx]$ // atomic operator
  end for
end for

// Index Compacting:
for each GPU_i **do in parallel**
  for $row \in IndexTable_i$ **do**
    RemoveDuplication(row)
  end for
end for

// Value Table Allocating:
for each GPU_i **do in parallel**
  $ValueTable_i \leftarrow$ AllocateValueTable(i)
end for

// Value Filling:
for each GPU_i **do in parallel**
  for $Element \in AssemblyElements$ **do**
    $entry \leftarrow$ FindElementEntry($Element$)
    $value \leftarrow$ GetElementValue($Element$)
    $ValueTable_i[entry]$ += $value$ // atomic operator
  end for
end for

2 STITCHING ALGORITHM

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

Algorithm 3 Stitching Algorithm

Input: Stitching node pairs NP.
Output: Stitched and refined cloth mesh .

// Stitching cloth pieces with linking constraints
// for all the NP.
StitchingCoarsePieces(NP)

// Merge the pieces together by merging
// all the node pairs NP
MergePieces(NP)

// Refine the merged cloth mesh to higher resolution
// by subdividing.
RefinePieces()

3 ADDITIONAL BENCHMARKS

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

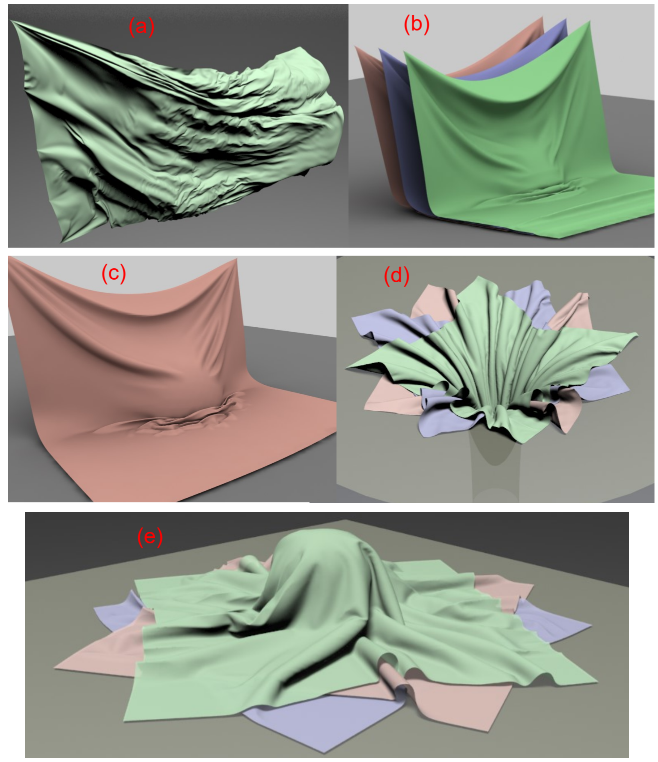

Fig. 1. **Benchmarks:** We use different multi-layered cloth simulation benchmarks ((a)Flag, (b) Sphere, (c) Sphere-1M, (d) Funnel, and (e) Twisting) for evaluation. The mesh complexity varies between $0.5 - 1.65$M triangles. P-Cloth can perform cloth simulation at $2 - 5$ fps on the 4-GPU workstation.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