

Contents

1	PRAM	6
1.1	Prerequisites	6
1.2	Definition	6
1.3	How it works	7
1.3.1	Computation	7
1.3.2	PRAM Classification	7
1.3.3	Strengths of PRAM	8
1.3.4	How to compare PRAM models	8
1.4	MVM algorithm	10
1.5	SPMD sum	12
1.6	MM algorithm	16
1.7	PRAM variants and Lemmas	17
1.8	PRAM implementation	18
1.9	Amdahl's and Gustafson's Laws	20
2	Fundamentals of architecture	23
2.1	Introduction	23
2.1.1	Simplest processor	23
2.1.2	Superscalar processor	24
2.1.3	Single Instruction, Multiple Data (SIMD) processor	25
2.1.4	Multi-Core Processor	25
2.2	Accessing Memory	26
2.2.1	What is a memory?	26
2.2.2	How to reduce processor stalls	28
2.2.2.1	Cache	28
2.2.2.2	Multi-threading	28
3	Programming models	31
3.1	Implicit SPMD Program Compiler (ISPC)	31
3.2	Shared Address Space Model	35
3.3	Message Passing model of communication	36
3.4	Data-Parallel model	37
4	Parallel Programming Models and pthreads	39
4.1	How to create parallel algorithms and programs	39
4.2	Analyze parallel algorithms	41
4.3	Technologies	44
4.4	Threads	47
4.4.1	Flynn's taxonomy	47
4.4.2	Definition	47
4.4.3	pthreads API	49
4.4.3.1	Creation	49
4.4.3.2	Termination	50
4.4.3.3	Joining	51
4.4.3.4	Detaching	52
4.4.3.5	Joining through Barriers	53
4.4.3.6	Mutexes	54
4.4.3.7	Condition variables	54

5	OpenMP v5.2	55
5.1	Introduction	55
5.2	Basic syntax	57
5.3	Work sharing	60
5.3.1	For	60
5.3.1.1	Reduction	65
5.3.2	Sections	67
5.3.3	Single/Master	68
5.3.4	Tasks	69
5.3.4.1	Task dependences	72
5.4	Synchronization	76
5.5	Data environment	79
5.6	Memory model	87
5.7	Nested Parallelism	90
5.8	Cancellation	94
5.9	SIMD Vectorization	97
6	GPU Architecture	100
6.1	Introduction	100
6.2	GPU compute mode	101
6.3	CUDA	103
6.3.1	Basics of CUDA	103
6.3.2	Memory model	107
6.3.3	NVIDIA V100 Streaming Multiprocessor (SM)	109
6.3.4	Running a CUDA program on a GPU	112
6.3.5	Implementation of CUDA abstractions	118
6.3.6	Advanced thread scheduling	121
6.3.7	Memory and Data Locality in Depth	126
6.3.8	Tiling Technique	135
6.3.8.1	Tiled Matrix Multiplication	138
6.3.8.2	Implementation Tiled Matrix Multiplication	143
6.3.8.3	Any size matrix handling	148
6.3.9	Optimizing Memory Coalescing	153
7	CUDA	162
7.1	Introduction	162
7.2	CUDA Basics	166
7.2.1	GPGPU Best Practices	168
7.2.2	Compilation	170
7.2.3	Debugging	172
7.2.4	CUDA Kernel	175
7.3	Execution Model	178
7.4	Querying Device Properties	180
7.5	Thread hierarchy	182
7.6	Memory hierarchy	185
7.7	Streams	193
7.8	CUDA and OpenMP or MPI	197
7.8.1	Motivations	197
7.8.2	CUDA API for Multi-GPUs	202
7.8.3	Memory Management with Multiple GPUs	205

7.8.4	Batch Processing and Cooperative Patterns with OpenMP	211
7.8.5	OpenMP for heterogeneous architectures	213
7.8.6	MPI-CUDA applications	216
8	Memory Consistency	220
8.1	Coherence vs Consistency	220
8.2	Definition	223
8.3	Sequential Consistency Model	225
8.4	Memory Models with Relaxed Ordering	229
8.4.1	Allowing Reads to Move Ahead of Writes	230
8.4.2	Allowing writes to be reordered	232
8.4.3	Allowing all reorderings	234
8.5	Languages Need Memory Models Too	236
8.6	Implementing Locks	238
8.6.1	Introduction	238
8.6.2	Test-and-Set based lock	240
8.6.3	Test-and-Test-and-Set lock	244
9	Heterogeneous Processing	248
9.1	Energy Constrained Computing	250
9.2	Compute Specialization	251
9.3	Challenges of heterogeneous designs	265
9.4	Reducing energy consumption	268
10	Patterns	271
10.1	Dependencies	271
10.2	Parallel Patterns	280
10.2.1	Nesting Pattern	281
10.2.2	Serial Control Patterns	282
10.2.3	Parallel Control Patterns	284
10.2.4	Serial Data Management Patterns	289
10.2.5	Parallel Data Management Patterns	292
10.2.6	Other Parallel Patterns	295
10.3	Map Pattern	297
10.3.1	What is a Map?	297
10.3.2	Optimizations	299
10.3.2.1	Sequences of Maps	299
10.3.2.2	Code Fusion	300
10.3.2.3	Cache Fusion	301
10.3.3	Related Patterns	302
10.3.4	Scaled Vector Addition (SAXPY)	304
10.4	Collectives operations	307
10.4.1	Reduce (or Reduction) Pattern	308
10.4.2	Scan Pattern	314
10.5	Gather Pattern	325
10.5.1	What is a Gather?	325
10.5.2	Shift	329
10.5.3	Zip	331
10.5.4	Unzip	332
10.6	Scatter Pattern	333

10.6.1 What is a Scatter?	333
Index	330

10.6 Scatter Pattern

10.6.1 What is a Scatter?

The **Scatter Pattern** is a data movement pattern where **elements from a source array are distributed (or scattered) to various locations in a destination array based on specified indices**. Essentially, it's about writing data to random locations.

★ Key Features

In general, the **input** of the scatter pattern is:

- **Source data array**: value to be written.
- **Index array**: specifies where each value should be written in the destination.

Each element from the source is written to the position in the target array specified by the corresponding index. The **output is a target array where the data is scattered across the positions**.

🔗 Gather vs. Scatter

The main differences between gather and scatter are:

- **Gather** (read focused): Involves **random reads** where the read **locations are provided as input**.
- **Scatter** (write focused): Involves **random writes** with write **locations provided as input**, which can lead to race conditions.

✂ Serial implementation

The following pseudocode shows a serial implementation of the scatter pattern:

```

1  template<typename Data, typename Idx>
2  void scatter(
3      size_t n,          // Number of elements
4                          // in the output data collection
5      size_t m,          // Number of elements
6                          // in the input data and index collection
7      Data a[],          // Input data collection (m elements)
8      Data A[],          // Output data collection (n elements)
9      Idx idx[]          // Input index collection (m elements)
10 ) {
11     // Loop through each input element
12     for (size_t i = 0; i < m; ++i) {
13         // Get the target index from the index array
14         size_t j = idx[i];
15         // Ensure the index is within output array bounds
16         assert(0 <= j && j < n);
17         // Perform the scatter: write a[i] to A[j]
18         A[j] = a[i];
19     }
20 }
```

The method is characterized by:

- **Loop** (for): Iterates through each element in the input array `a[]` (length `m`).
- **Indexing** (`j = idx[i]`): Determines **where to place** `a[i]` in the output array `A[]`.
- **Boundary Check** (assert): Ensures the index `j` is **valid** (avoids out-of-bounds errors).
- **Write Operation** (`A[j] = a[i]`): **Scatters** the data from `a[i]` to `A[j]`.

The possibilities to parallelize the code are interesting, but instead of the gather pattern, here we have a problem with race conditions on write operations. So the code can be parallelized (**parallelize over for loop to perform random write**), but we **need to find some strategies to avoid race conditions**.

⚠ Race Conditions in Scatter

Unfortunately, the scatter pattern can lead to race conditions because of the way it handles writes to memory. As we know, a race condition occurs when two or more operations try to access and modify the same data at the same time, and the final result depends on the order in which the operations are performed.

In the scatter pattern, we're performing random writes to different locations in memory. The **problem arises when multiple write operations target the same location at the same time**. Because the **writes happen in parallel**, we can't predict:

1. Which write will happen first
2. Which value will be stored last

Example 48: Race Condition in Scatter

Consider this:

- Source Data:

`A = [A, B, C, D, E, F]`

- Index Array (write locations):

`idx = [1, 5, 0, 2, 2, 4]`

This means that the two threads that will handle the 3rd and 4th positions on the index array will suffer from a race condition. Because if they try to write at the same time, the result will depend on which thread finishes last:

- If thread A (writing D) finishes last, position 2 will be equal to D.
- If thread B (writing E) finishes last, position 2 will be equal to E.