Exercise Session 8

Data Processing on Modern Hardware
263-3502-00L — Fall Semester 2012

Cagri Balkesen
cagri.balkesen@inf.ethz.ch

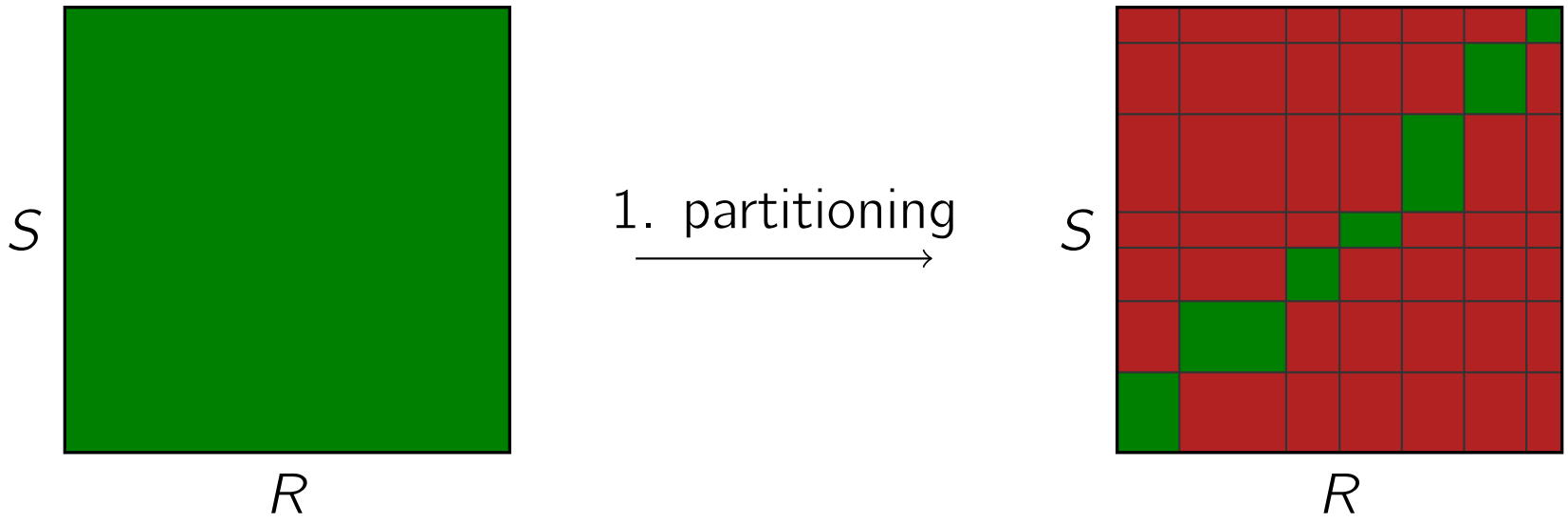
Department of Computer Science ETH Zurich, Switzerland

8 November 2012

Parallel Partitioned Hash Join

- Accelerate *Partitioned Hash Join* from Assignment 3 with hardware parallelism
- Thread-level parallelism (TLP): use n Pthreads, where $n = \#cores \times HW - threads$ in your machines
  - First partitioning phase ($\rightarrow$ more difficult)
  - Subsequent partitioning phases ($\rightarrow$ easier)
- Data-level parallelism (DLP): use SSE/AVX to further accelerate your implementation
  - Nested-loops join
  - More opportunities, e.g. building the histogram with SIMD?
- Can you join 100 million tuples in less then 1 second, as in [KSC+09]?

Subsequent Partitioning Phases



- After first partitioning phase, threads can work on different partitions without interference
- Use the *Task Queue Model* [MKJ91] to avoid load imbalances

Creating POSIX Threads (Pthreads)

- Standard for threads that defines an API for creating and manipulating threads
- Initializing a thread

```
typedef struct {
  ...
} thread_ctx_t;

void *run(void *arg) {
  thread_ctx_t *ctx = (thread_ctx_t *) arg; ...
}
pthread_t thread;
pthread_attr_t attr;
thread_ctx_t ctx;

pthread_attr_init(&attr);
pthread_create(&thread, &attr, run, &ctx);
```

Running Pthreads

- Create N threads
- Run threads using `pthread_create(...)` and wait for them to finish with `pthread_join(...)`

```
for (i=0; i<N; i++) {
  pthread_join (threads[i], NULL);
}
```

- Pass *task queue* as argument to threads
- Threads should terminate once task queue is empty
- `pthread_join(...)` can be used to synchronize threads

Thread synchronization: Mutual Exclusion (Mutex)

- Avoid simultaneous use of common resources
  $\rightarrow$ critical sections in code
- Access to task queue should be handled as critical section (check queue, fetch task, update queue)

```
pthread_mutex_t mutex;
if(pthread_mutex_init(&mutex, NULL)) {
  printf("Unable to initialize a mutex\n");
  return -1;}

pthread_mutex_lock(&mutex);
size = queue_size(ctx->taskqueue);
if(size>0) task = queue_pop(ctx->taskqueue);
pthread_mutex_unlock(&mutex);

pthread_mutex_destroy(&mutex);
```

Thread synchronization: Barriers

- Initialize and destroy barrier

```
pthread_barrier_t barr;
if(pthread_barrier_init(&barr, NULL, NUMTHREADS)) {
    printf("Could not create a barrier\n");
    return -1;}
...
pthread_barrier_destroy(&barr);
```

- Synchronize threads inside "run" function

```
int rc = pthread_barrier_wait(&barr);
if(rc != 0 && rc != PTHREAD_BARRIER_SERIAL_THREAD) {
    printf("Could not wait on barrier\n");
    exit(-1);}
```

Parallelize the first Partitioning Phase

- Can be implemented in three steps (synchronize after every step)
  - Equally divide the input tuples amongst τ' tasks and compute histogram $Hist_i$ for each task
  - Compute *prefix sum* for each task
  - Iterate second time over tuples of each task and scatter tuples to their final destination
- Set $\tau' = 4\tau$, where τ is the number of threads

Data-level parallelism (DLP)

- Nested-loops can be parallelized with SIMD [ZR02]
  - duplicate-inner
  - duplicate-outer
  - rotate-inner ($\rightarrow$ use shuffle)
  - rotate-outer ($\rightarrow$ precompute rotations)
- With lack of efficient scatter/gather SIMD-operations not much opportunities for DLP in hash join

Questions

- Which optimization has the most impact?
- If your machine supports hyper-threading: does it improve performance?
- What is a good granularity for TLP—when are we bottlenecked by lock contention?
- How much speedup can we expect by DLP?
- How does your implementation perform with skewed data ($\rightarrow$ use zipf data generator)?

References

[HJ86] W. Daniel Hillis and Guy L. Steele Jr.

Data parallel algorithms.

Commun. ACM, 29(12):1170–1183, 1986.

[KSC+09] Changkyu Kim, Eric Sedlar, Jatin Chhugani, Tim Kaldewey, Anthony D. Nguyen, Andrea Di Blas, Victor W. Lee, Nadathur Satish, and Pradeep Dubey.

Sort vs. hash revisited: Fast join implementation on modern multi-core cpus.

PVLDB, 2(2):1378–1389, 2009.

[MKJ91] Eric Mohr, David A. Kranz, and Robert H. Halstead Jr.

Lazy task creation: A technique for increasing the granularity of parallel programs.

IEEE Trans. Parallel Distrib. Syst., 2(3):264–280, 1991.

[ZR02] Jingren Zhou and Kenneth A. Ross.

Implementing database operations using simd instructions.

In *SIGMOD Conference*, pages 145–156, 2002.