Disclaimer

* EVC - Die Fortsetzung

Das ist nur eine Zusammenfassung und kein Ersatz für das Skriptum / die VOs. Keine Garantie darauf, dass alles so stimmt, wie es hier steht. Das ist nur meine Interpretation der Inhalte. Falls etwas unklar sein sollte, bitte im Skriptum nachschauen.
Second Block ist nicht inkludiert, da dieser nicht Teil vom Prüfungsstoff ist. (meines Verständnisses nach)

Viel ~~Spaß~~ beim ~~Lernen~~! ERFOLG BETEN

(und viel Glück beim Test)

Inhaltsverzeichnis

1. Block: "Shared Memory" Systeme

cache = small, fast memory ⇒ reduces acess times by reusing read values, create overhead if no reuse (less accesses to (slow) main memory needed)
↳ segmented in blocks (typically of 64 Bytes size), granularity: unit of memory blocks
↳ cache line: stores one memory block + meta information (bits, flags)
↳ direct mapping (1-way set associative): map each block on predetermined line; fully associative: map block on any cache line
set associative: map each block on any line within predetermined set; k-way set associative: k-set sizes (k=2,4,8,...)

READ
is block that contains read word in the cache? yes = cache hit ⇒ fast read / no = cache miss ⇒ read block into c. line
compulsory (cold) miss: empty cache, contains no adress blocks
capacity miss: full cache, some cache line has to be evicted
conflict miss: full set, frequent in directly mapped caches, either of k lines can be evicted ⇒ eviction policy (usually least recently or least frequently used (LRU/LFU))

WRITE
is block of the adress written to in the cache? yes = overwrite it / no = allocate or update address in memory (write no-allocate)
write through: update block that is in cache line by writing value to memory
write back: keep value in cache line until it is evicted

types of locality of access: temporal locality – several reuses of memory address contents in brief succession
spatial locality – several reuses of memory addresses in same block

MCC PROBLEMS

Work/Seq. Time of Matrix-Matrix Multiplication Algorithm: $O(nml)$ miss-rate determined by line size,
Factor of 20-40 between best and worst loop orders because of row-major storage in C ⇒ cache stores 1 row
Work of recursive, divide-and-conquer Matrix-Matrix Multiplication Algorithm: $O(\log^2 n)$ or even $O(n^{2.81})$
Cache-Aware Algorithm: small enough submatrices to fit in the cache / cache oblivious: everything above

Multi-Core Caches: Hierarchy of Caches in increasing Size (L_1 = lowest level, fastest, smallest, closest to CPU)
↳ L_1: usually divided into data and instruction caches ⇒ has another cache: translation look-aside buffer (TLB)
↳ L_1 and sometimes L_2: private to one processor-core, as opposed to shared higher-level caches
cache coherence problem: memory block is in private L_1 cache of two cores
↳ coherent cache system: cache line gets updated in both / non-coherent cache system: it never gets updated in both
↳ updated = modified with the new value or invalidated (= next access is a miss)
↳ non-trivial task, requires cache coherence protocol, fulfills local consistency, update transfer & write consistency
false sharing problem: caused by granularity ⇒ updates to two addresses by two cores create coherence traffic
↳ avoided by allocating frequently used variables on different cache lines (e.g. padding = one address per block)

write buffer: buffers writes to the main memory to process them in the appropriate pace, makes them appear fast
may be FIFO or sorted, usually coalesce writes to same address, NUMA to the main memory
memory controller: ensures connection between CPU core and main memory, memory is banked along the controller
access times per core depend on "closeness" to controller
"first touch" policy: virtual memory page will be put in the memory bank closest to the core ⇒ performance boost

super-linear speed-up: working set on p processors ⇒ $\frac{1}{p}$ of the working set on one processor
if p grows ⇒ $\frac{1}{p}$ shrinks and fits in faster caches ⇒ speed-up of kp, $k>1$
roofline performance model: distinction between memory-bound and compute-bound applications:
memory-bound app: operations performed per unit read/write take less time than reading/writing a unit
↳ memory performance (= access times) determines system performance
compute-bound app: operations performed per unit read/write take more time than reading/writing a unit
↳ nominal processor performance determines system performance

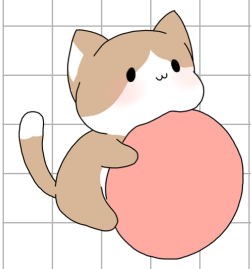
programm order: seq. execution ⇒ reads/writes happen in execution order of the program's instructions
↳ proves properties of state invariants
↳ concurrent, asynchronous execution of two programs ⇒ interleaving neccessary to ensure memory = program order
sequential consistency: type of memory consistency, considers possibility of different interleavings
↳ not fulfilled in modern multi-core systems, due to per-core write buffers
↳ well-defined memory state gets ensured by hardware mechanisms (memory fences, atomic operations) instead

main problems of MCC: cache coherence – Different Cores write/read Same Memory Address
memory consistency – Different Cores write/read Different Memory Addresses

3. Block: Die Schnittstelle/Sprache OpenMP

Open Multi Processing

main characteristics:
1. implicit parallelism through work sharing, SPMP { all threads execute same program
2. Fork join parallelism (master spawns threads, join at the end of parallel area)
3. unique int thread id per parallel area, consecutively numbered ↑ between 0 and thread number - 1
4. number of threads can exceed CPU number
5. Constructs for sharing work across threads
6. Threads can share variables and have private variables
7. Unprotected, parallel updates may lead to data races and errors
8. Synchronisation Constructs for preventing data races
9. Consistent memory state after Synchronisation



requirements in C: OpenMP-capable compiler, function prototype header (omp.h), "omp_" prefix on all functions and predefined objects, "OMP_" prefix for special variables

parallel region construct: structured C statement, once started, number of threads in parallel region unchangeable

```
#pragma omp parallel [clauses]
<structured statement>
```

joining threads through barrier synchronisation, only master remains active after join

control of thread number by runtime environment or by num_threads()
↳ if runt. env.: default value (CPU number) or env. variable OMP_NUM_THREADS

oversubscription: set OMP_NUM_THREADS larger than CPU amount ⇒ useful for debugging, not performance

openmp library calls: access to stable timer,

```
double omp_get_wtime(void); returns wall clock time in s
double omp_get_wtick(void); returns resolution of the timer
```

```
int omp_get_thread_num(void);
int omp_get_num_threads(void);
int omp_get_max_threads(void);
void omp_set_num_threads(int num_threads);
```

thread safe - can be called concurrently, in parallel without risk of interference

declaring variables: best practice to set everything to default(none) and manually share everything necessary, shared variables can be read concurrently, but the same variable must not be read & written to at the same time

```
private(<comma separated list of variables>) - not initialized
firstprivate(<comma separated list of variables>) - initialized to value before parallel region
shared(<comma separated list of variables>)
default(shared|none) - share variables declared by master by default
```

OPENMP-SYNTAX

work sharing:

```
#pragma omp parallel
{
  int i; // private i for each thread
...
#pragma omp master
  readdata(a,n);

#pragma omp barrier
  // compute
  for (i=0; i<n; i++) {
    b[i] = ...; // per thread computation from a into b
  }

#pragma omp barrier - no thread may continue before all threads have reached this point
#pragma omp single nowait
  writedata(b,n);
}
```

```
#pragma omp master
<structured statement> - all threads skip it except master, no mutal exclusion necessary
```

allows private/firstprivate

can cause data races

```
#pragma omp single [clauses] done by either of the threads
<structured statement> - has implied barrier at the end ⇒ canceled by nowait clause
```

no implicit barrier of single construct needed ⇒ eliminated by nowait
because there already is a barrier, no correctness guaranteed
without it ⇒ data races in both a and b possible

variables can be private/firstprivate

Consists of:

```
#pragma omp sections [clauses]
<section block>
```

```
#pragma omp section [clauses]
<structured statement>
```

ends with barrier ⇒ canceled by nowait
one thread pre-assigned per section

while-loops not allowed

```
#pragma omp for [clauses] for (<canonical form loop range>)
<loop body>
```

⇒ all threads must compute the same, finite loop range and loop range may not change
⇒ canonical form: upper bound must be of the form

```
#pragma omp parallel for [clauses] → does not allow nowait
for (<canonical form loop range>)
<loop body>
```

i<n, i≤n, i>n, i≥n or i≠n; increment must be of form i++, i+=inc or i=i+inc (e.g. i-- for decrements)

independence of loops: loop does array updates only; updates at most one unchanged element per iteration

LOOP FUNCTIONS

Loop scheduling: how to share loop work through threads? ⇒ loop range is divided into chunks
- ↳ static schedule — same-sized chunks, assigned in static, round-robin fashion } no overhead, fast assignments, good for same work per iteration
- ↳ dynamic schedule – same-sized chunks, each thread grabs next unassigned once finished } preferable for loops with index-step-ending conditions
- ↳ guided schedule – no fixed size, chunks get assigned dynamically to free threads

```
schedule(static[,chunksize])  – default chunksize: n/p
schedule(dynamic[,chunksize]) – default chunksize: 1
schedule(guided[,chunksize])  – default chunksize: 1
```

schedule(auto)
schedule(runtime)
↳ set in env. var: OMP_SCHEDULE

collapse nested loops:
- performance boost
- easier parallelisation

```
#pragma omp parallel for collapse(2)   // nesting depth to collapse
for (i=0; i<n; i++) {   // OpenMP will make one loop out of two
  for (j=0; j<m; j++) {
    x[i][j] = f(i,j);
  }
}
```

reduction:

```
reduction(<reduction operator>:<reduction variables>)
```
allowed operators: +, -, *, &, |, ^, &&, ||, min, max

```
#pragma omp scan exclusive(<reduction variables>)
```
for exclusive prefix sums

```
#pragma omp scan inclusive(<reduction variables>)
```
for inclusive prefix sums

work sharing-part II (tasks): gets completed at latest as possible,

```
#pragma omp task [clauses]
<structured statement>
```
→ earlier completion possible by setting barrier

```
#pragma omp taskwait [clauses]
```
→ enforces waiting for completion of new tasks

mutal-exclusion constructs: (expensive, big performance impact)

```
#pragma omp critical [(name)]
<structured statement>
```
order undefined, mutal exclusion guaranteed

```
#pragma omp atomic [read|write|update|capture]
<atomic statement>
```
(fetch and add)
supports FAA, restricted form (x++, ++x, x+=, -x, x=x)

Locks:

```
void omp_init_lock(omp_lock_t *lock);
void omp_init_nest_lock(omp_nest_lock_t *lock);
void omp_destroy_lock(omp_lock_t *lock);
void omp_destroy_nest_lock(omp_nest_lock_t *lock);
void omp_set_lock(omp_lock_t *lock);
void omp_set_nest_lock(omp_nest_lock_t *lock);
void omp_unset_lock(omp_lock_t *lock);
void omp_unset_nest_lock(omp_nest_lock_t *lock);
```

- do not have condition variables
- provides nested locks without try-lock
- does not provide reader-and-writer locks
- OpenMP is not intended for lock programming

special loops executed with SIMD instructions:

```
#pragma omp simd [clauses]
for (<canonical form loop range>)
<loop body>
```
sequential-one thread

```
#pragma omp parallel for simd [clauses]
for (<canonical form loop range>)
<loop body>
```
parallel region with SIMD loop sharing

```
#pragma omp for simd [clauses]
for (<canonical form loop range>)
<loop body>
```
loop within parallel region to be shared among threads

```
#pragma omp taskloop [clauses]
for (<canonical form loop range>)
<loop body>
```
break iteration range into smaller tasks-initiated by single thread

parallelelise loops with hopeless dependencies: by executing non-parallisable part sequentially

```
#pragma omp ordered
<structured statement>
```
- one ordered block per loop
- brings overhead, allows partial parallelisation

cilk: gives rise to fully strict computations

```
cilk_spawn <function call>   – generate task
cilk_sync                    same as taskloop
cilk_for (<canonical form iteration space>) <loop body>
```

⇒ allows merging problem to be solved work-optimally with $T_1(n) = O(n)$ and $T_\infty(n) = O(\log^2 n)$