Overheads

- each OpenMP-directive/routine comes with a runtime cost that is not present in the sequential case
- parallel-region: threads must be woken up, runtime-internal data structures are created
- work-sharing: concrete work for each thread is usually determined at runtime
- load-imbalance: some threads idle at barriers (implicit or explicit) -> don't contribute to work
- synchronization: waiting for a lock to be freed (e.g. implementation of critical region)

Try to eliminate barriers

- idle threads do not contribute to work
- barriers incur overhead
  - Implicit barriers can be redundant!
  - at the end of parallel region, for/section/single work-sharing directives
- make sure semantic is still the same!

```
#pragma omp parallel
{
  #pragma omp single nowait
  while (my_pointer) {
  #pragma omp task firstprivate(my_pointer)
  (void) do_independent_work(my_pointer);  // the task's code
  my_pointer = my_pointer->next;
  }  // no barrier here anymore
} // tasks get executed at this point now
```

Maximize parallel regions

- on the extreme end, all of a parallel program's code could be wrapped in a single parallel region → difficult with respect to sharing and load-balancing
- more parallel regions → creation and destruction of OpenMP-internal thread-management data-structures → expensive!
  - usually no thread creation, since major OpenMP implementations keep a pool of threads created at program launch
- also: within a parallel region exists a consistent OS-thread to thread-id mapping: not necessarily consistent over multiple parallel regions! This has implications for reuse of cached data!
- big parallel regions → bigger execution context → better optimization potential for compiler

Example

```
for (int i = 0; i < N; ++i)
  a[i] += b[i];
for (int i = 0; i < N; ++i)
  c[i] += d[i];
double sum = 0;
for (int i = 0; i < N; ++i)
  sum += a[i] + c[i];
```

Example – Initial Parallel iteration

```
#pragma omp parallel for schedule(static)
for (int i = 0; i < N; ++i)
  a[i] += b[i];
#pragma omp parallel for schedule(static)
for (int i = 0; i < N; ++i)
  c[i] += d[i];
double sum = 0;
#pragma omp parallel for schedule(static) \
        reduction(+:sum)
for (int i = 0; i < N; ++i)
  sum += a[i] + c[i];
```

Example – better parallel version

```
#pragma omp parallel
{
#pragma omp for schedule(static)
for (int i = 0; i < N; ++i)
  a[i] += b[i];
#pragma omp for schedule(static)
for (int i = 0; i < N; ++i)
  c[i] += d[i];
double sum = 0;
#pragma omp for schedule(static) \
        reduction(+:sum)
for (int i = 0; i < N; ++i)
  sum += a[i] + c[i];
}
```

Example – optimize for barrier use

```
#pragma omp parallel
{
#pragma omp for schedule(static) nowait
for (int i = 0; i < N; ++i)
  a[i] += b[i];
#pragma omp for schedule(static) nowait
for (int i = 0; i < N; ++i)
  c[i] += d[i];
#pragma omp barrier
double sum = 0;
#pragma omp for schedule(static) \
        reduction(+:sum) nowait
for (int i = 0; i < N; ++i)
  sum += a[i] + c[i];
}  // another implied barrier
```

Example – even less barriers

- Since OpenMP 3.1:
  - when used within the same parallel region
  - and static scheduling
  - over the same loop-trip count
- the same iterations of consecutive OpenMP for-loops will be processed by the same threads

```
#pragma omp parallel
{
#pragma omp for schedule(static) nowait
for (int i = 0; i < N; ++i)
  a[i] += b[i];
#pragma omp for schedule(static) nowait
for (int i = 0; i < N; ++i)
  c[i] += d[i];
double sum = 0;
#pragma omp for schedule(static) \
        reduction(+:sum) nowait
for (int i = 0; i < N; ++i)
  sum += a[i] + c[i];
}  // implied barrier
```

Shorten Critical sections

- used when
  - mutual exclusion between threads is needed
  - processing order between threads not important
- larger critical sections → higher probability of threads waiting on each other
- move non-critical work outside of critical section

```
double a, c, d;
#pragma omp parallel default(none) \
        shared(a) private(c,d)
{
  // ... some code initializing c
  // because: private variables are
  // UNITIALIZED by default
  c = omp_get_thread_num();
  #pragma omp critical
  {
    a += 2 * c;
    c = d * d;  // unnecessary
  }
  // continue working with c and d
  ...
}
```

Load imbalance

- different amounts of work for different threads in a team
- for-loops: consider use of schedule clause with dynamic/guided mode
  - more overhead by the OpenMP runtime
  - if load-imbalance is severe enough, this overhead could be outweighed by the speedup gained
- Pipelined Processing

```
for (int i = 0; i < NUM_CHUNKS; ++i) {
  ReadFromFile(i, ChunkSize);
  for (int j = 0; j < N; ++j)
    ProcessData();  // expensive
  WriteResult(i);
}
```

Pipelined Processing

```
#pragma omp parallel
{
 #pragma omp single
 ReadFromFile(0, ChunkSize);
 for (int i = 0; i < N; ++i) {
  #pragma omp single nowait
  if (i < N - 1)  ReadFromFile(i + 1, ChunkSize);
  #pragma omp for schedule(dynamic)
  for (int j = 0; j < ProcessingNum; ++j)
   ProcessChunkOfData();
  #pragma omp single nowait
  WriteResultToFile(i, ChunkSize);
 }
}
```

Race Conditions

- occurs when
  - at least 2 threads access a shared memory location
  - at least 1 thread is writing to that location
- Race conditions change the semantic of the program depending on OS thread scheduling and number of threads involved
  → inconsistent behavior between runs
- use default(none) clause and mark variables to be shared explicitly using shared clause
- avoids unintentional sharing due to otherwise default sharing policy
- highlights shared variables explicitly to detect explicitly introduced race conditions

Race condition - Example

```
int i = 0;
#pragma omp parallel num_threads(2) \
        default(none) shared(i)
        // is this a good idea?
{
  ++i;
}

// possibly replace with

int i = 0;
#pragma omp parallel num_threads(2) \
        default(none) reduction(+:i)
{
  ++i;
}
```

Dead-Lock

- a thread for a resource that is never going to be available
- when do dead-locks occur?
  1. resource access is exclusive
  2. thread is allowed to hold one resource while requesting another
  3. no thread is willing to free a lock for the sake of progress
- avoid any ONE of these causes
- if 2) cannot be changed: impose an order into the sequence of lock acquisition: any thread is only allowed to
  lock(A) → lock(B) → lock(C)
- consider using private copies of resource and copy/merge back after modification
- don't use synchronization within a *dynamic context*

Dead-Lock: *Dynamic context*

```
void f1() {}
void f2() {
  // wait here forever
  #pragma omp barrier
}
#pragma omp parallel sections
{
  #pragma omp section
  f1();
  #pragma omp section
  f2();
}
```

General advice

- don't use the ordered construct
  - expensive to implement for the runtime
  - threads wait …
- remember: variables declared private are UNINITIALIZED on entry
- know when using not-thread-safe libraries, or avoid if possible
  - thread-safe = shared ressources within a function static/global variables
  - mostly important when writing to shared location, but reading might also be problamatic: think about random seeding
  - STL-containers are not thread safe

Oversubscription

- too many threads for a particular amount of work/machine
- bad in two ways:
  - Too much overhead if work per thread too little
  - More than OS hardware(!) threads
    round-robin scheduling (necessary to assure progress)
    → context switches (slow)
    → cache trashing(!)
- careful when choosing value for OMP_NUM_THREADS
- measure overhead and use if/num_threads-clause

Non-Uniform Memory Architectures (NUMA) and MPI

- NUMA architectures were introduced with the need to scale SMP (symmetric multiprocessor) systems
- memory access latency order of magnitude slower than CPU compute latency
  gets even worse with more and more cores per CPU
- UMA refers to an architecture where all CPUs/cores have the same access time to main memory
  Laptop-CPU, single Xeon Phi → UMA architecture
- difficult to scale with ever increasing number of CPUs within a single node → NUMA

NUMA

- introduces the concept of local- and non-local memory
- a set of CPUs/cores is assigned a share of main memory that is local only to them
- the rest of the main memory is far-memory with respect to access time
- local memory: lower latency, higher bandwith
- non-local memory can still be accessed
- Intel: Quick-Path, AMD: HyperTransport comparatively slow inter-socket connection

NUMA and Multicore Programming

- How is allocated memory assigned to local/non-local memory?
- today's operating systems use virtual memory management → every process has own, full(!) address space
- OS maps virtual addresses to physical addresses in main memory via pages
- pages are usually 4kb in size containing contiguous memory
  - Xeon Phi allow for huge pages of 2MB

NUMA and Virtual Memory

- page assignment is usually not carried out at allocation:

```
std::vector<double> v1(0); v1.reserve(1000);
double * v2 = malloc(1000 * sizeof(double));
```

- pages are assigned on first touch: the thread first reading/writing a memory location triggers page mapping to its local memory

```
std::vector<double> v1(1000);
double * v2 = malloc(1000 * sizeof(double));
for (int i=0;i<1000;++i) v2[i] = i % 3;
```

- implicit page to memory assignment!

Implications to multi-threaded code

- local memory provides potential for memory access speedup, but...
- the moment of first touch is sometimes difficult to identifiy
- reading a memory location not local to the threads local memory
  → data travels over QPI/HyperTransport
  expensive!
- today's NUMA systems are implemented as cache-coherent NUMA (cc-NUMA) → lots of protocoll communication over QPI/HT
- OpenMP itself is not NUMA aware: it is up to the programmer to select a good OS-thread-to-thread-num assignment → difficult to get right for all of the application