Simulation and High-Performance Computing

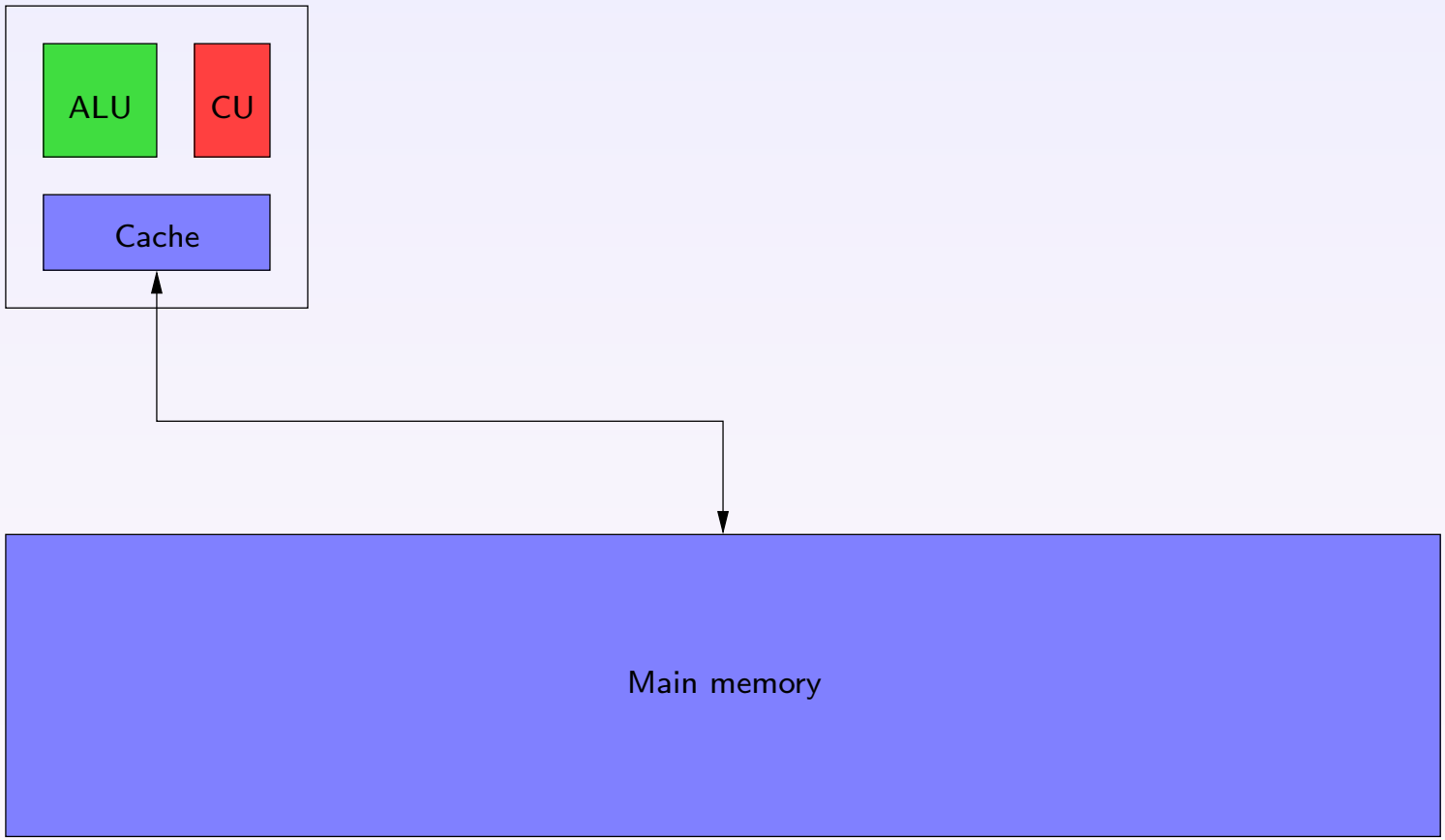
Part 11: OpenMP

Steffen Börm

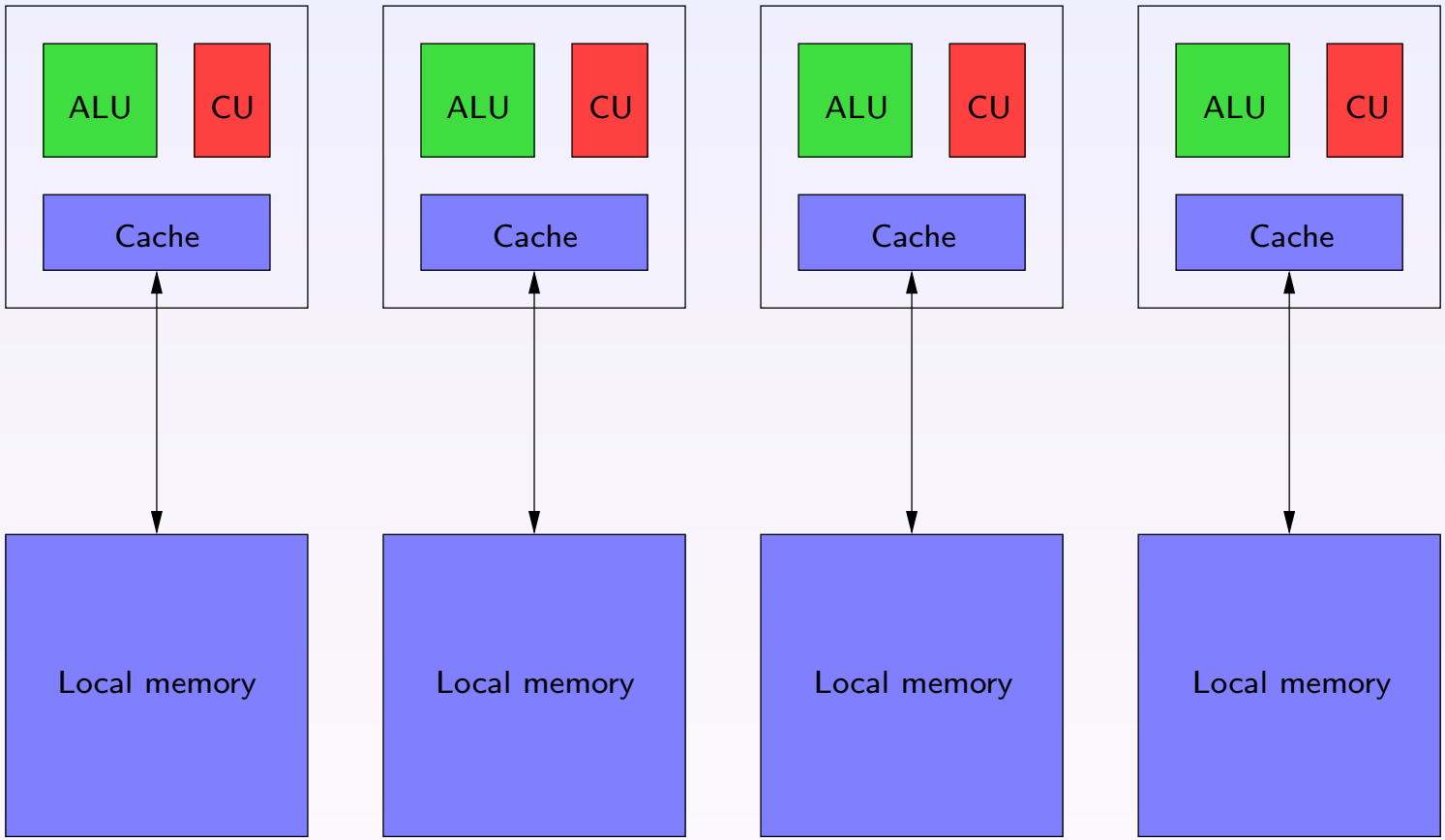
Christian-Albrechts-Universität zu Kiel

October 5th, 2020

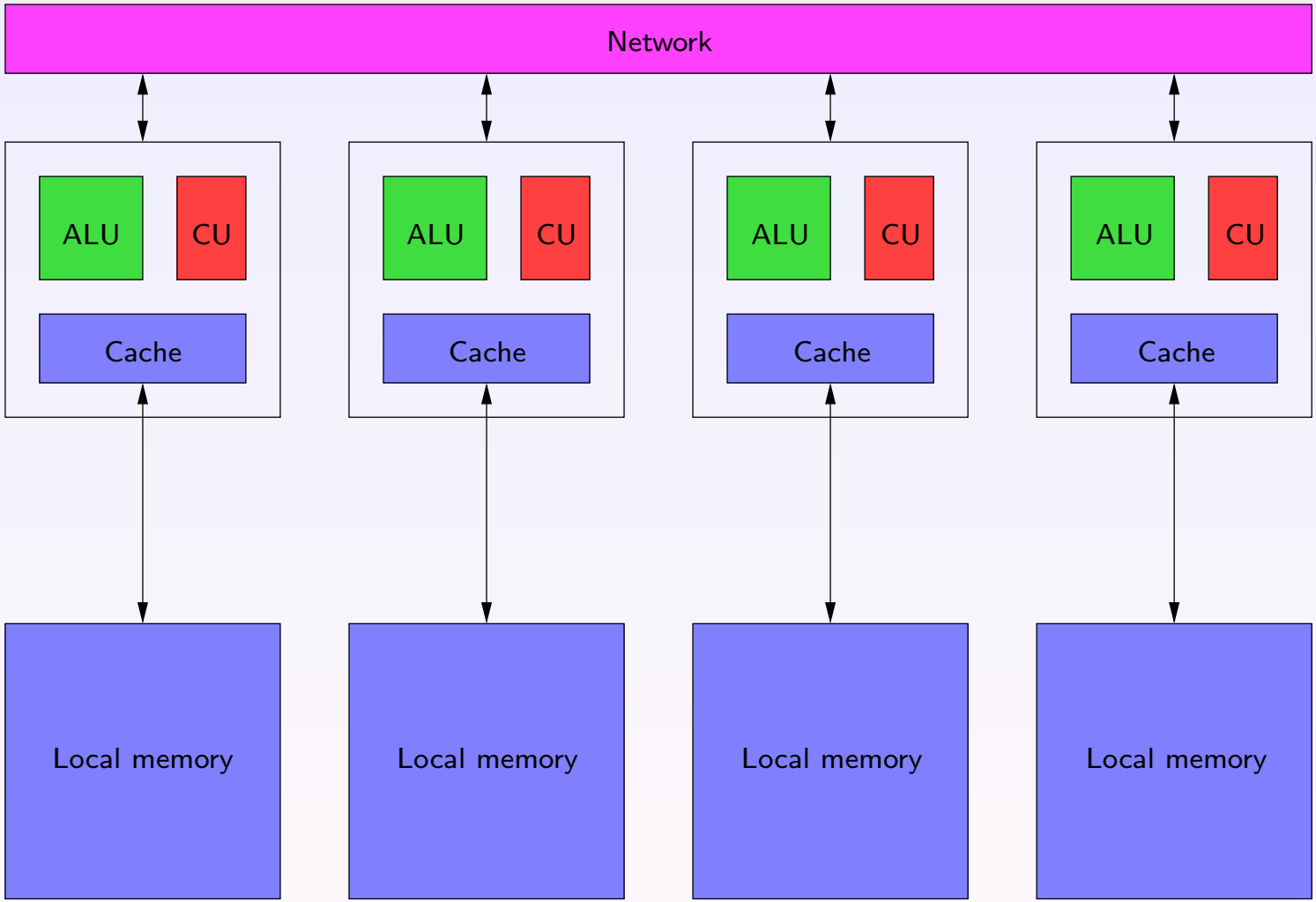
Parallel computers



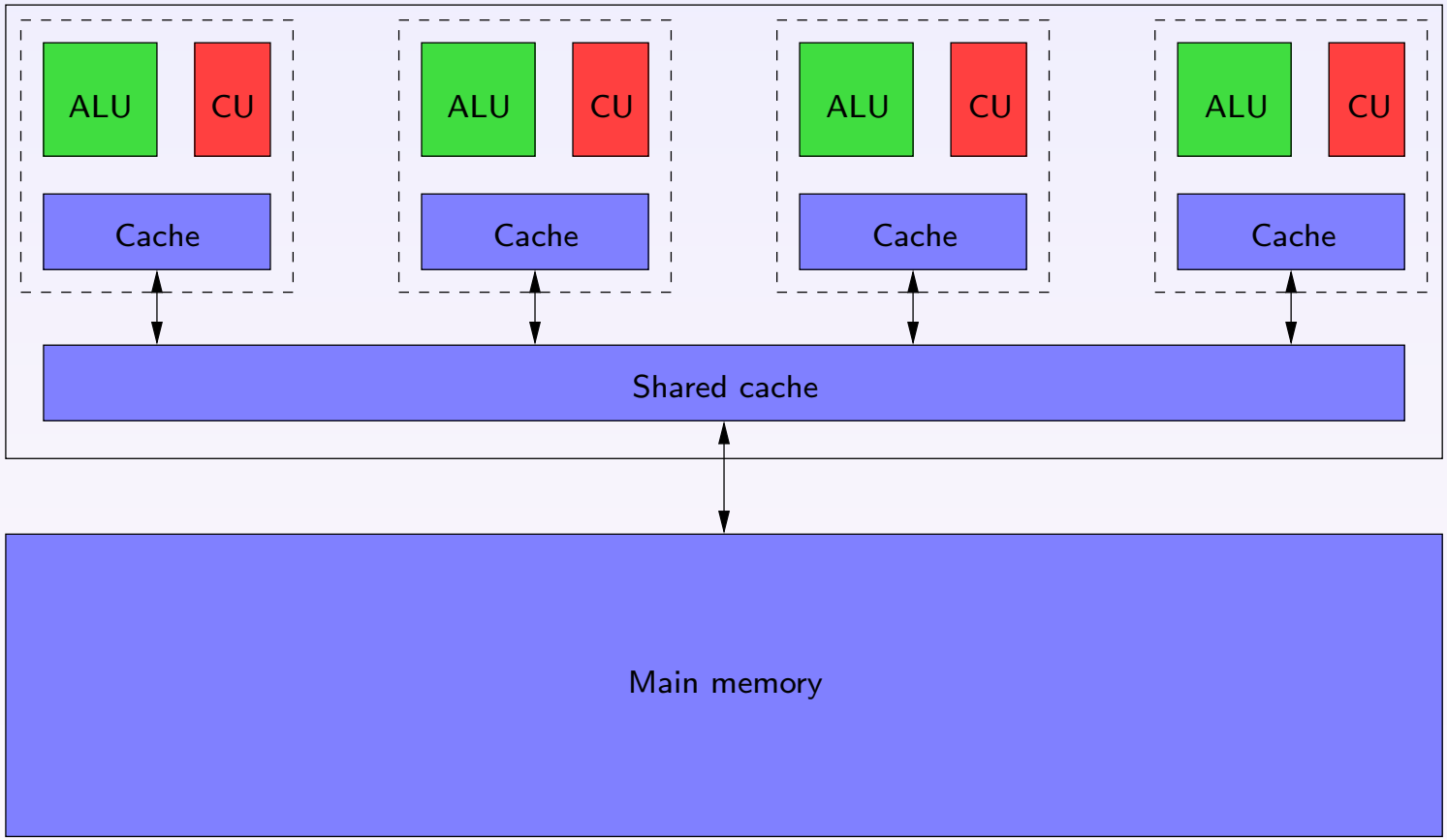
Parallel computers



Parallel computers



Parallel computers



Multithreading

Goal: Allow multiple processor cores to work together.

Multithreading:

- Programs are split into threads.
- Each thread has its own instruction pointer, registers, and stack.
- Threads are executed by processor cores.
- All threads share the same address space.

Multithreading

Goal: Allow multiple processor cores to work together.

Multithreading:

- Programs are split into threads.
- Each thread has its own instruction pointer, registers, and stack.
- Threads are executed by processor cores.
- All threads share the same address space.

Challenges:

- Synchronization: Threads may have to wait for other threads.
- Deadlocks: Threads should not wait forever.
- Data conflicts: Multiple threads accessing the same variable.
- Idle threads: Threads should not spend too much time waiting.

OpenMP

OpenMP is an industry standard for writing multithreaded programs in C, C++, and FORTRAN.
In this lecture, we will use OpenMP 4.0.

Augmentation: OpenMP extends C by allowing programmers to mark sections of a program that can be split into multiple threads.
Done mostly by `#pragma omp` directives, but also by library functions.

OpenMP

OpenMP is an industry standard for writing multithreaded programs in C, C++, and FORTRAN.
In this lecture, we will use OpenMP 4.0.

Augmentation: OpenMP extends C by allowing programmers to mark sections of a program that can be split into multiple threads.
Done mostly by `#pragma omp` directives, but also by library functions.

Concepts:

- The program starts with a thread running the `main` function.
- When a thread enters a parallel section of the program, a team of threads is created and the original thread becomes its master thread.
- The threads in a team are enumerated consecutively starting with the number zero for the master thread.
- Variables can be shared or private.

Parallel sections

```
#pragma omp parallel
{
  int rank = omp_get_thread_num();
  int size = omp_get_num_threads();

  printf("Thread %d of %d reporting for duty\n",
         rank, size);
}
```

- `#pragma omp parallel` creates a parallel section that is executed by a team of threads.
- `omp_get_thread_num()` returns the number of the current thread. for the master thread.
- `omp_get_num_threads()` returns the number of threads in the team.

Private and shared variables

```
int i = 0, j = 5;

#pragma omp parallel firstprivate(j)
{
  int k = omp_get_thread_num();

  i++;

  printf("i=%d, j=%d, k=%d\n", i, j, k);
}
```

- `i` is a shared variable.
  In this program, it may be subject to write conflicts.
- In the parallel section, `j` is a private variable that is initialized when the section is entered.
- `k` is a private variable.

Work sharing

```
int i;

#pragma omp parallel
{
  #pragma omp for
  for(i=0; i<n; i++)
    x[i] += alpha * y[i];
}
```

- `#pragma omp for` states that the following `for` loop can be parallelized, i.e., that its range can be determined in advance and that all iterations are independent.
- `#pragma omp for` distributes the iterations among all threads in the current team.

Explicit work sharing

```
#pragma omp parallel
{
  int rank = omp_get_thread_num();
  int size = omp_get_num_threads();
  int start = n * rank / size;
  int end = n * (rank+1) / size;
  int i;

  for(i=start; i<end; i++)
    x[i] += alpha * y[i];
}
```

- We can use `omp_get_thread_num()` and `omp_get_num_threads()` to assign work to each thread individually.
- We have to include the `omp.h` header file.

Example: Discrete Laplacian

```
diag = 4.0 / h / h;
off = -1.0 / h / h;

#pragma omp parallel for collapse(2)
for(j=1; j<n; j++)
  for(i=1; j<n; i++)
    yv[i+j*ld] += alpha * (diag * xv[i+j*ld]
                         + off * xv[(i-1)+j*ld]
                         + off * xv[(i+1)+j*ld]
                         + off * xv[i+(j-1)*ld]
                         + off * xv[i+(j+1)*ld]);
```

- *#pragma omp parallel for* creates a parallel section for the loop.
- The `collapse(2)` clause combines both `for` loops into one.

Example: Trapezoidal rule

```
real h = (b-a) / n;

#pragma omp parallel for reduction(+:sum)
for(i=1; i<n; i++)
  sum += integrand(a+h*i);

sum = h * (0.5 * integrand(a)
           + sum + 0.5 * integrand(b));
```

- The `reduction` clause creates private copies of the `sum` variable for all threads and writes their sum into the original `sum` variable.
- All private copies start at zero.

Write conflicts

```
int i = 0;

#pragma omp parallel
i++;

printf("%dn\", i);
```

- This program may return any value between one and the number of threads used in the parallel section.
- `i++` is typically translated into reading the current value of the shared variable `i`, adding one, and writing the result back into the variable.
- If all threads read before any thread writes, we get one.
- If every thread reads only after the previous thread has written, we get the number of threads.

Avoiding write conflicts

Atomic operations are "indivisible".

```
#pragma omp parallel
{
  #pragma omp atomic
  i++;
}
```

While one thread performs the read-add-write sequence, the variable `i` is "locked" for all other threads.

Avoiding write conflicts

Atomic operations are "indivisible".

```
#pragma omp parallel
{
  #pragma omp atomic
  i++;
}
```

While one thread performs the read-add-write sequence, the variable `i` is "locked" for all other threads.

Critical sections can only be executed by one thread at a time.

```
#pragma omp parallel
{
  #pragma omp critical access_i
  i++;
}
```

If we ensure that `i` is only accessed within critical sections named `access_i`, collisions are voided.

Locking

```
typedef struct cluster_struct cluster;
typedef struct cluster_struct {
  /* ... data ... */
  omp_lock_t lck;
};

void
change_cluster(cluster *s)
{
  omp_set_lock(s->lck);
  /* ... do something ... */
  omp_unset_lock(s->lck);
}
```

- A thread trying to set a lock waits until it is unset.
- Locks are useful, but may significantly slow down the program.

Barriers

```
rank = omp_get_thread_num();
size = omp_get_num_threads();

for(i=1; i<size; i*=2) {
  if(rank+i < size && !(rank & i))
    result[rank] += result[rank+i];
  #pragma omp barrier
}
```

- All threads in the current team have to reach the barrier before any thread can pass it.
- Very useful for algorithms that can be split into phases during which all threads work independently.

False sharing

Problem: Data is moved between the cache and the main memory in cache lines, frequently of 64 bytes.
If two threads work with different variables in the same cache line, write operations of both threads may interfere.

```
#pragma omp section
  for(i=0; i<1000000000; i++)
    a[0]++;

#pragma omp section
  for(i=0; i<1000000000; i++)
    a[k]++;
```

- If $k = 32$, the program takes 1.87 seconds.
- If $k = 1$, the program takes 6.60 seconds.

NUMA architectures

Non-uniform memory access: Although all processors can access the entire main memory, bandwidth or latency depend on how "close" the processor is to a portion of memory.

Memory mapping: Calling the `malloc` function does not actually allocate physical memory, only logical memory.
Physical memory is usually assigned when a processor first tries to access logical memory, and usually as "close" to this processor as possible.

Strategy: If we ensure that the thread that first accessed a portion of memory does most of the work with this portion, we can hope to avoid the negative effects of NUMA architectures.

Thread creation

Inside a loop: 0.230 seconds.

```
for(k=0; k<1000; k++) {
  #pragma omp parallel for
  for(i=0; i<n; i++) {
    /* ... work ... */
  }
}
```

Outside a loop: 0.207 seconds.

```
#pragma omp parallel private(k)
for(k=0; k<1000; k++) {
  #pragma omp for
  for(i=0; i<n; i++) {
    /* ... work ... */
  }
}
```

Summary

Multithreading allows us to take advantage of multicore processors and other shared-memory architectures.

OpenMP uses compiler directives like `#pragma omp parallel` or `#pragma omp for` to define parts of a program that can run in parallel and to control work sharing.

Conflicts can be avoided by using atomic operations, critical sections, locks, or barriers.

Performance can be improved by avoiding false sharing and taking NUMA architectures into account.