Exercises for Architectures of Supercomputers

9th Exercise, 22./23.1.2020

Johannes Hofmann

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Exercise Agenda

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

- Introduction to OpenMP (de-facto multi-core programming standard)
- Importance of thread pinning
- Uniform Memory Access vs. Non-Uniform Memory Access
- Multicore parallelization for vector sum

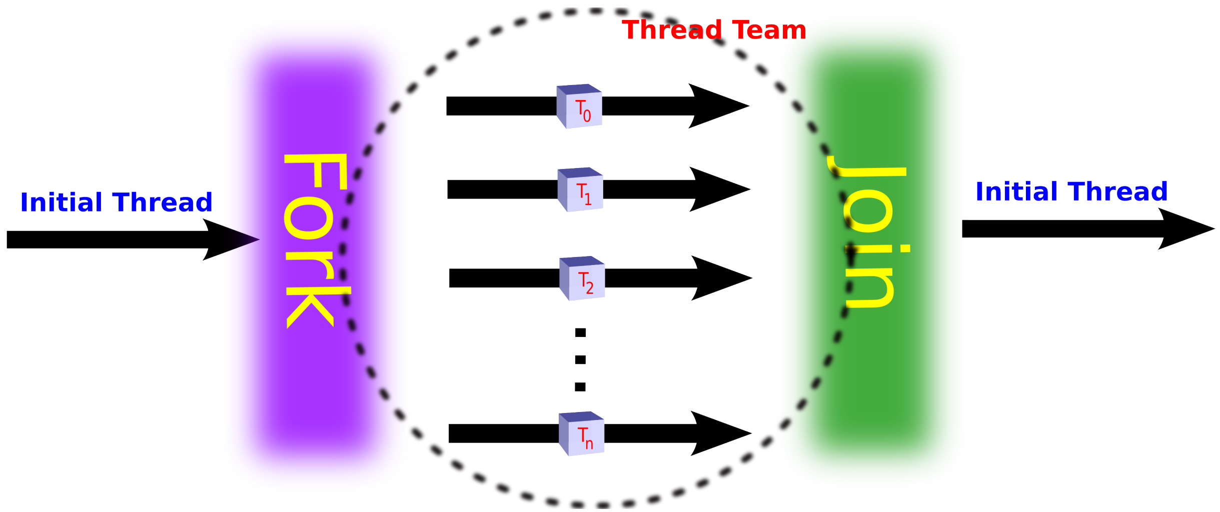
FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Design goals for OpenMP

- Easy handling
- Sequential equivalence
  - Parallel program produces “same” results as serial program
- Incremental parallelization
  - An existing program should be easy to parallelize step-by-step using OpenMP

Taking advantage of multicore parallelism

- OpenMP is based on the fork-join model
  - Processes start with a single thread
  - Additional threads (thread team) are forked in parallel regions
  - Implicit barrier (wait for all) at the end of each parallel region
    - Can be disabled
  - Join after parallel region



OpenMP

Example: “Hello world!”

Parallel version

```
$ cat hello_omp.c
#include <omp.h>
#include <stdio.h>
int main(int argc, char **argv) {
  #pragma omp parallel
  {
    printf(“Hello World!\n”);
  }
}

$ gcc -fopenmp -o hello_omp \
  helloomp.c
$ ./hello_omp
Hello World!
Hello World!
...
```

Sequential version

```
$ cat hello.c
#include <stdio.h>
int main(int argc, char **argv) {
  printf(“Hello World!\n”);
}

$ gcc -o hello hello.c
$ ./hello
Hello World!
$
```

OpenMP

- Number of OpenMP threads set by runtime system
- Can be influenced by programmer
  - Environment variable `OMP_NUM_THREADS`
    ```
    $ OMP_NUM_THREADS=3 ./hello_omp
    Hello World
    Hello World
    Hello World
    ```
  - In OpenMP pragma
    ```
    #pragma omp parallel num_threads(3)
    ```
  - Using `likwid-pin` (recommended, discussed later)

OpenMP – Querying thread information at runtime

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char **argv) {
    #pragma omp parallel num_threads(4) {
    int my_num, total_threads;
    my_num = omp_get_thread_num();
    total_threads = omp_get_num_threads();
    printf("Hello World from thread %d of %d!\n, my_num, total_threads);
  }
}
```

```
Hello World from thread 2 of 4!
Hello World from thread 0 of 4!
Hello World from thread 3 of 4!
Hello World from thread 1 of 4!
```

OpenMP – Private vs. shared variables

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char **argv) {
    int a; /* shared variable */
    #pragma omp parallel {
    int b; /* private variable */
    a = omp_get_thread_num(); // Probably not what you want
    b = omp_get_thread_num(); // Okay
    }
}
```

- Race condition: The order in which threads write to shared variable ‘a’ is not defined!

OpenMP – Work-sharing constructs

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

OpenMP has several work-sharing constructs that facilitate work distribution and thread management for the programmer

- `parallel for` construct
- `sections` constructs
- `single` constructs

In the following, the each construct is discussed in more detail

OpenMP – For Construct


```
void add(double *A, double *B, double *C, int N) {
    #pragma omp parallel {
      #pragma omp for
      for (int i=0; i<N; ++i)
          A[i] = B[i] + C[i];
    }
}
```

Work (i.e., loop iteration) distribution when using four threads, N = 100.000

Thread	Start	Ende
0	0	24999
1	25000	49999
2	50000	74999
3	75000	99999

OpenMP – Section Construct

- Parent ‘sections’ keyword to indicate multiple ‘section’s
- Each section is processed by a dedicated thread

```
#pragma omp sections {
  #pragma omp section {
  // executed by one thread
  // ...
  }

  #pragma omp section {
  // executed by another thread
  // ...
  }

  /...
}
```

OpenMP – Section Construct – Example

Sequential

```
for (int i=0; i<N; ++i) {
  sum+=A[i];
  prod*=A[i];
}
```

Parallel using OpenMP

```
#pragma omp parallel {
  #pragma omp sections {
    #pragma omp section {
      for (int i=0; i<N; ++i)
        sum+=A[i];
    }

    #pragma omp section {
      for (int i=0; i<N; ++i)
        prod*=A[i];
    }
  }
}
```

OpenMP – Single Construct

- Indicates code inside a parallel regions is only to be executed by a thread (any thread). Useful, e.g., when writing to shared variables

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char **argv) {
    int a; /* shared variable */
    #pragma omp parallel {
      int b; /* private variable */
      #pragma omp single {
        a = omp_get_thread_num(); // Okay
      }
      b = omp_get_thread_num(); // Okay
    }
}
```

OpenMP – Short notation for work-sharing constructs

- Short notation for work-sharing constructs

```
#pragma omp parallel {
  #pragma omp <for, section, single> {
  }
}
```

can be written as

```
#pragma omp parallel <for, section, single> {
}
```

OpenMP – Worksharing Constructs

- Example

```
void add(double *A, double *B, double *C, int N) {
    #pragma omp parallel {
      #pragma omp for
      for (int i=0; i<N; ++i)
          A[i] = B[i] + C[i];
    }
}

void add(double *A, double *B, double *C, int N) {
    #pragma omp parallel for
    for (int i=0; i<N; ++i)
          A[i] = B[i] + C[i];
}
```

OpenMP – Memory management

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

- Variables declared outside the parallel region, are considered shared variables

```
int my_id=0;
#pragma omp parallel {
  my_id=omp_get_thread_num(); // Bad
}
```

- The `private(<list>)` clause instructs the compiler to create local “copies” of the variables (their initial value is undefined!)

```
int my_id=0;
#pragma omp parallel private(my_id) {
  my_id=omp_get_thread_num(); // Okay
}
```

OpenMP – Reduction clause

Thread-local values are reduced to one global result according to a specified binary operator

```
double sum=0.0;
#pragma omp parallel for reduction(+:sum)
for (int i=0; i<N; ++i)
  sum+=a[i];
```

- Supported operators: +, -, *, &, |, ^, &&, ||

OpenMP – Scheduling

- `#pragma omp parallel for`-loop construct shares the loop iterations among all threads
- Scheduling strategy determines how work is distributed among threads
- Scheduling strategy can be set by the programmer (default is static)
- `#pragma omp parallel for schedule (<sched> [,chunk])`
  - `sched`: `static, dynamic, guided, runtime`
  - `chunk`: Number of work items fetched per queue access
- Different scheduling strategies can be applied in different scenarios
  - Is the time per work unit fixed? → static scheduling
  - Is the time per work unit dynamic? → dynamic / guided scheduling

OpenMP – Scheduling

- `schedule(static [,chunk])`
  - Work bundled in blocks/chunks of `chunk` (consecutive) loop iterations
  - Blocks distributed equally among threads
  - No runtime overhead, no work-load balancing
- `schedule(dynamic[,chunk])`
  - Work bundled in blocks/chunks of `chunk` (consecutive) loop iterations
  - Each thread starts working on a chunk of loop iterations, the remaining chunks are in a work queue
  - When a thread has finished its chunk, it gets a new one from the queue
  - Compared to `static` scheduling, `dynamic` scheduling has a runtime overhead (accessing the queue), but provides work-load balancing

OpenMP – Scheduling

- `guided(static [,chunk])`
  - Like dynamic scheduling but with varying chunk size
  - Chunk size decreases over time
    - Fewer queue accesses when chunks are large
    - Retain load-balancing property by reducing block size as amount of work in queue decreases

- `schedule(runtime)`
  - Scheduling strategy can by set at runtime using the environment variable `OMP_SCHEDULE`

```
$ OMP_SCHEDULE="static, 1024" ./program
```

OpenMP – Further Reading

- OpenMP specification

  http://openmp.org/wp/openmpspecifications/

- Online OpenMP tutorial

  https://computing.llnl.gov/tutorials/openMP/

Exercise Agenda

- Introduction to OpenMP (de-facto multi-core programming standard)
- **Importance of thread pinning**
- Uniform Memory Access vs. Non-Uniform Memory Access
- Multicore parallelization for vector sum

LIKWID tool suite

- Started by Jan Eitzinger, now developed by Thomas Gruber at RRZE

- Collection of multiple useful HPC tools (swiss army knife for HPC)
  - `likwid-topology` – Show information about node
  - `likwid-pin` – Set thread affinity
  - `likwid-bench` - Microbenchmarks
  - `likwid-perfctr` – Investigate hardware performance counters
  - `likwid-powermeter` – Measure energy consumption

- Make likwid available on Emmy
  - `$ module load likwid`

The `likwid-pin` tool

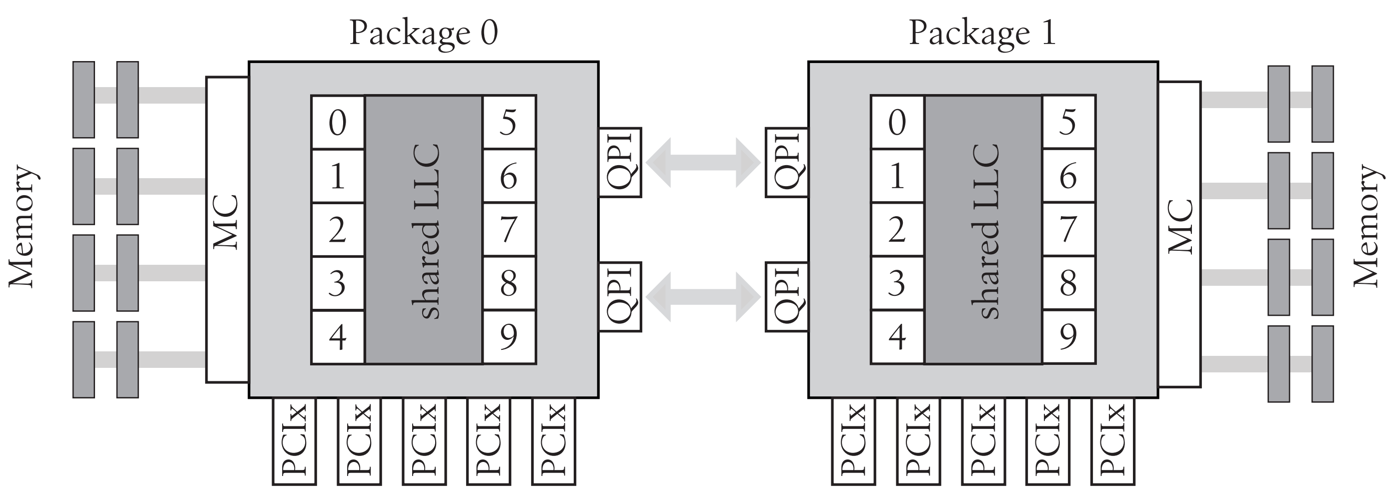
- Without arguments, all logical(!) cores will be used
  - `likwid-pin ./my_openmp_binary`
- The most generic way to specify the desired number of threads and where to execute them is the expression syntax
  - `likwid-pin -c E:<domain>:<nthreads>:<threads per core>:<maximum supported SMT threads per core>`

    where

    - `domain` is where you want the threats to run, e.g., socket 0 (S0), socket 1 (S1), or the entire node (N)
    - `nthreads` is the total number of threads you want to run
    - `threads per core` is the number of threads you want to run per core (e.g., one, two, …, *n* threads on a core that supports *n*-SMT)
    - `n-SMT` is the maximum number of SMT threads supported per core
- More details
  - https://github.com/RRZE-HPC/likwid/wiki/Likwid-Pin

The `likwid-pin` tool

- Examples
  - Use one physical core, no SMT
    - `likwid-pin -c E:N:1:1:2 ./binary`
  - Use *n* physical cores, no SMT
    - `likwid-pin -c E:N:n:1:2 ./binary`
  - Use one physical core, 2-SMT
    - `likwid-pin -c E:N:2:2:2 ./binary`
  - Use *n* physical cores, 2-SMT
    - `likwid-pin -c E:N:2*n:2:2 ./binary`

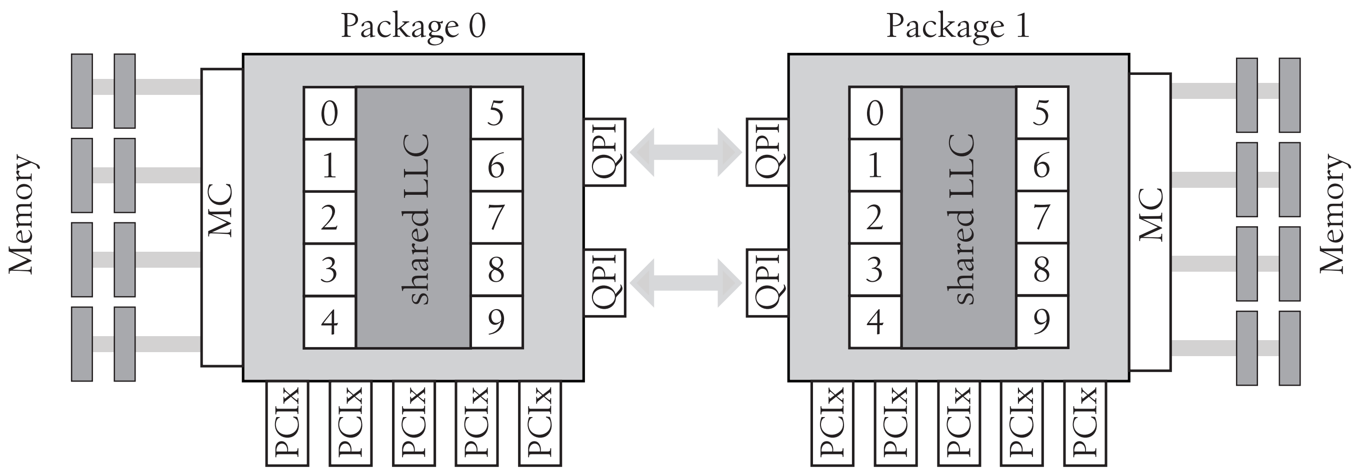


Exercise Agenda

- Introduction to OpenMP (de-facto multi-core programming standard)
- Importance of thread pinning
- **Uniform Memory Access vs. Non-Uniform Memory Access**
- Multicore parallelization for vector sum

UMA vs. NUMA

- Uniform memory access (UMA) (typically used in regular desktop/laptop/mobile designs)
  - Memory bandwidth and latency identical on all cores
- Non-uniform memory access (NUMA)
  - Memory bandwidth and latency can differ between cores
  - LINUX uses a “first-touch” policy to determine page placement
  - A memory page is placed in the NUMA node of the associated core that first reads/writes an address inside the page



Exercise Agenda

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

- Introduction to OpenMP (de-facto multi-core programming standard)
- Importance of thread pinning
- Uniform Memory Access vs. Non-Uniform Memory Access
- **Multicore parallelization for vector sum**

Exercise: Parallelization of vector sum

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

- Use OpenMP to parallelize a SIMD-vectorized implementation of the computation of the sum of a vector
    - You can use your own implementation from exercise 8 or let the compiler SIMD-vectorize the naïve version (make sure to use `-xHost`)

Exercise: Parallelization of vector sum

- Measure the performance of your code
  - Use `likwid-pin` for thread pinning
  - Create plots showing the number of **physical cores** on the *x*-axis and performance on the *y*-axis
    - Use a single plot for exercises 1 and 2, as well as exercises 3 and 4

- Multi-core scaling
  1. Measure performance for n = 1, 2, …, 10 physical cores of a processor **without SMT** for data-set sizes of:
     - n * 27kB (each core's data set will fit into its private L1 caches)
     - n * 120kB (each core's data set will fit into its private L2 caches)
     - n * 1MB (the data set will fit into the shared L3 cache)
     - n * 200MB (the data set will be in main memory)
  2. Measure performance for n = 1, 2, …, 10 physical cores of a processor **with 2-SMT** for the same prevdata-set sizes as in (1)

Exercise: Parallelization of vector sum

- UMA vs. NUMA
  3. **Without** NUMA-aware initialization, measure performance for n = 1, 2, …, 20 physical cores for a fixed data-set size of 16 GB
  4. **With** NUMA-aware initialization, measure performance for n = 1, 2, …, 20 physical cores for a fixed data-set size of 16 GB

  - I does not matter whether use SMT or not in the measurements above, just be consistitent