HPC Programming

OpenMP, Part III

Peter-Bernd Otte, 5.11.2019

Recap

Control Structures - Overview

- Parallel region construct
  - `parallel`

- Worksharing constructs
  - `for`
  - `sections`
  - `task`
  - `single`
  - `master`

- Synchronisations constructs
  - `critical`

- Defines **work load** among threads

- worksharing & sync constructs do not launch new threads
  - parallel construct creates a team of threads which execute in parallel

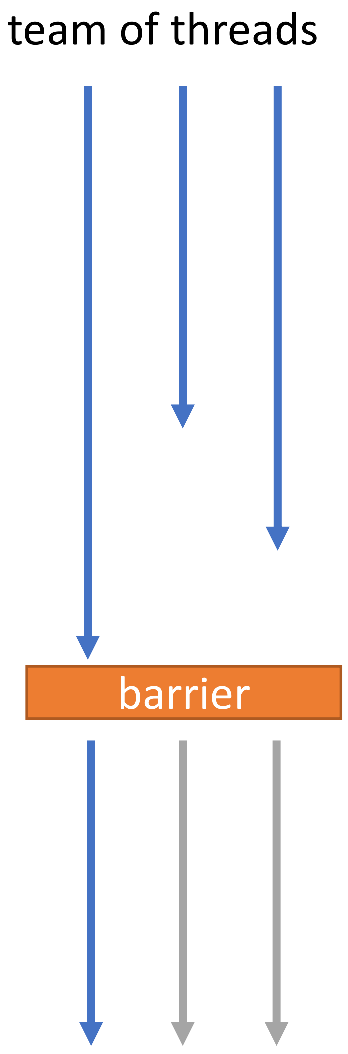
- worksharing comes with implicit barrier (threads wait until complete work finished):
  - none on entry
  - normally one at the end

OpenMP: Barrier

- barrier = all threads in a team wait until all threads reached barrier

- Implicit barrier
  - entry and exit of parallel constructs
  - exit of all other control constructs (except: nowait clause)
- Explicit barrier
  - critical directive
  - single directive

  → see later



OpenMP: for Directive (1)

- Parallelises the following for loop
    - in canonical form → see next slide.
    - loop iterations: all independent!

Allows the iteration count (of all associated loops) to be computed before the (outermost) loop is executed.

- Within parallel region
- `#pragma omp for [clause …] new-line`

```
      for-loop(s)
//end of for loop
```

OpenMP: single ⬄ critical

- single:
  - section executed by single thread
  - only once
- critical:
  - section executed by one thread at a time
  - num_threads() times

```
int a=0, b=0;
#pragma omp parallel num_threads(4)
{
  #pragma omp single
  a++;
  #pragma omp critical
  b++;
}
printf("single: %d critical: %d", a, b);
```

```
result:
single: 1 critical: 4
```

Introduction OpenMP

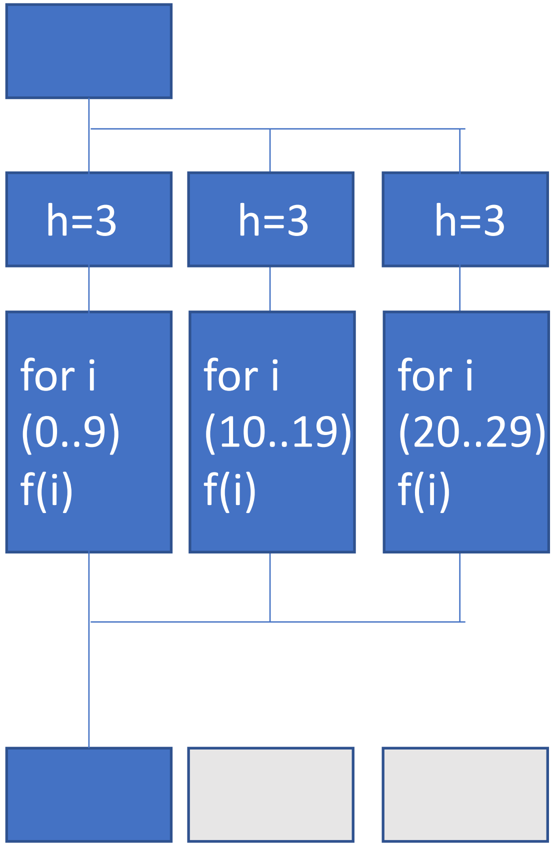
1. Hardware Anatomy
2. Motivation
3. Programming and Execution Model
4. Work sharing directives and combined constructs
5. Data environment
6. Common pitfalls and good practice (“need for speed”)

OpenMP: for Directive

- Parallelises the following for loop
    - in canonical form
    - loop iterations: all independent
- Within parallel region
- `#pragma omp for [clause …] new-line`

```
#pragma omp for [clause …] new-line
    for-loop(s)
//end of for loop
```

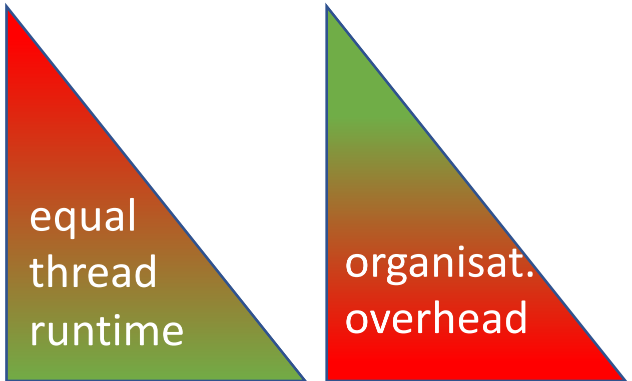
- Clauses:
    - reduction (op: list)
    - collapse (n)
      (n=const.: iterations of following n nested loops are collapsed into one larger iteration space)
    - schedule (type, chunk)
      (how the work is divided among the threads)



OpenMP: for Directive, scheduling

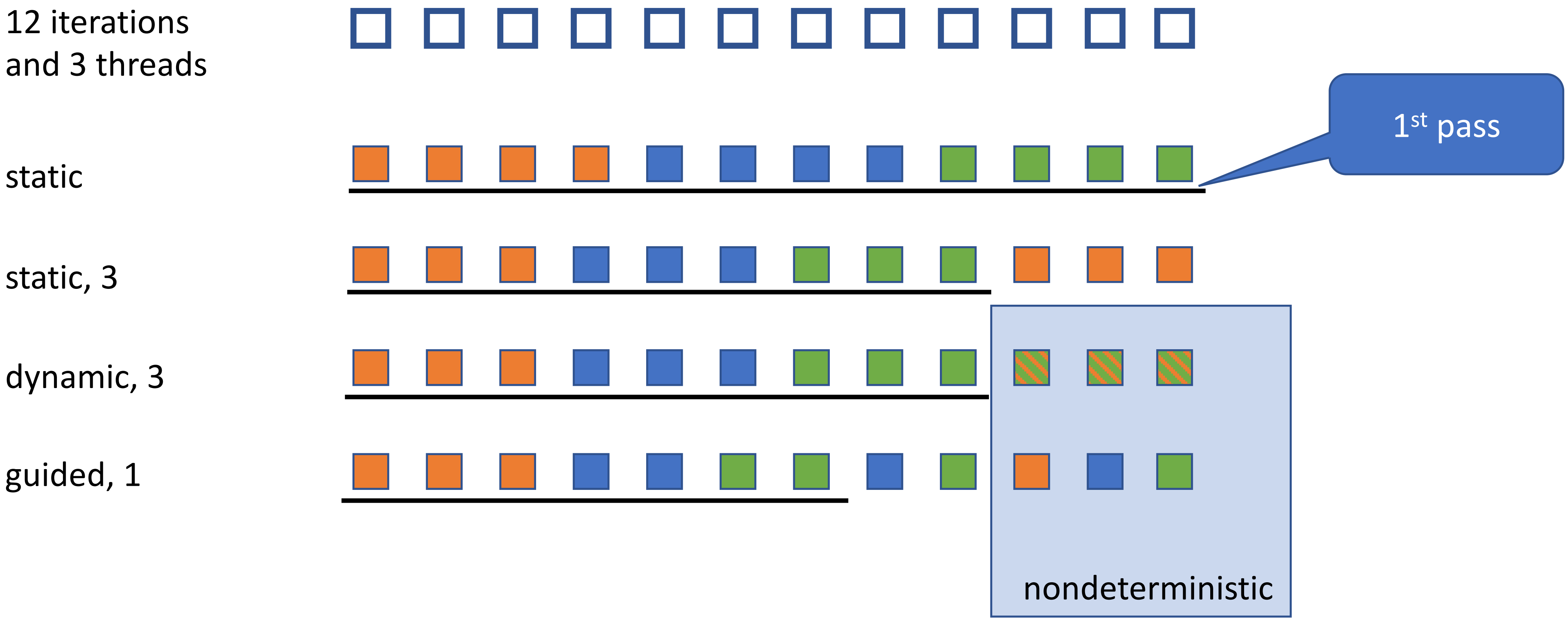
- How the work (n iterations) is divided among the p threads
  - Clause: schedule (type[, chunk])
- Type:
  - static: one chunk per thread with equal n / or with chunk size provided: chunks are statically assigned to threads.
  - dynamic: threads obtain chunks of size c when free (default: c=1 iteration).
  - guided: Like dynamic, but chunk size decays exponential with time until minimal chunk size = c.
  - auto: implementation dependent.
  - runtime: (no chunk must be provided in source code) Set OMP_SCHEDULE during runtime, eg “guided,10”

	chunk provided?	**iterations per chunk**	**N(chunks)**	**deter-ministic**
static	no	n/p	p	*yes*
static	yes	c	n/c	*yes*
dynamic	optional	c	n/c	*no*
guided	optional	n/p (beginning), exp. decreasing	$< n/c$	*no*


equal thread runtime

organisat. overhead

OpenMP: for Directive, scheduling



12 iterations
and 3 threads

static

static, 3

dynamic, 3

guided, 1

1st pass

nondeterministic

GCC standard scheduling

- What is clause “auto” in gcc?
  - https://github.com/gcc-mirror/gcc/blob/master/libgomp/loop.c#L195 and https://github.com/gcc-mirror/gcc/blob/master/libgomp/loop_ull.c#L192
  - /* For now map to schedule(static), later on we could play with feedback driven choice. */
  - >11 years ago (Jun 6th 2008)
    - Git blame: https://github.com/gcc-mirror/gcc/blame/d9dbca4b3382b7eb0504cc5ae5f9081af368b52c/libgomp/loop.c#L195

Introduction OpenMP

1. Hardware Anatomy
2. Motivation
3. Programming and Execution Model
4. Work sharing directives and combined constructs
5. Data environment
6. Common pitfalls and good practice (“need for speed”)

OpenMP: reduction clause (1)

- Syntax: `reduction (operator : list)`
    - Operator: +, *, -, &, ^, |, &&, ||, min, max
    - Variables: shared

- On loop completion, performs a reduction on the variables in list, with the operator
    - After reduction the shared variable is updated
    - internally working with local copies, like in example 3, step 6

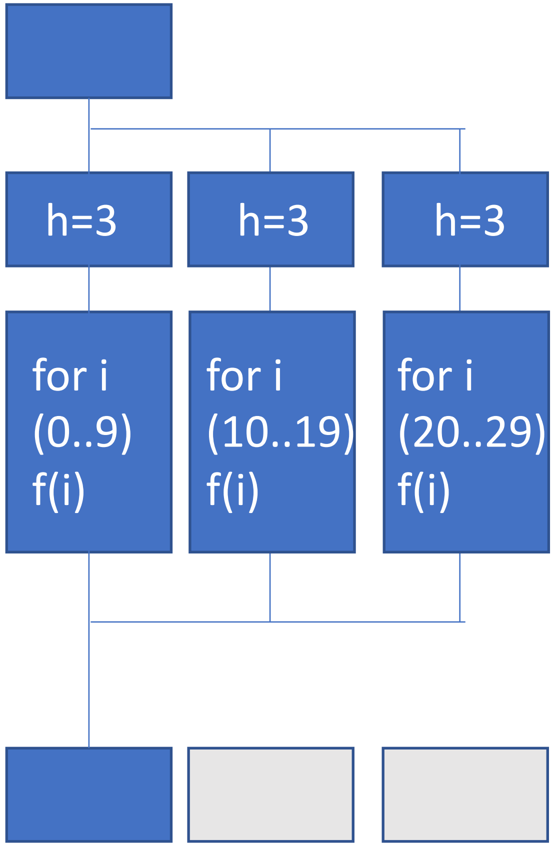
OpenMP: reduction clause (2)

```
double res;
#pragma omp parallel shared(h,res)
{
      h=3;

      #pragma omp for reduction (+:res)
      for (int i=0; i<30; i++) {
            res = res + f(i);
      }


} /* OMP end parallel


printf(“sum: %f”, res);
```



Introduction OpenMP

1. Hardware Anatomy
2. Motivation
3. Programming and Execution Model
4. Work sharing directives and combined constructs
5. Data environment
6. Common pitfalls and good practice (“need for speed”)

OpenMP: Need for Speed

- 1st: Optimise your serial program!
  - Identify, where the time get's consumed
- Can your program scale? → Amdahl's Law
- What else to check?
  - T(overhead) << T(complete runtime)
    - Test with profiler (valgrind, tau, … → see future lecture on this)
    - use different OMP_NUM_THREADS
    - only serialise time consuming parts of your serial program

    - try different schedules ("equal runtime of all cores?")
    - use private variables wherever possible
    - name your critical sections
    - use abort statement: if (…) to switch to single core if faster
  - use avoid (implicit and explicit) barriers wherever possible (clause: "nowait")
  - prevent unnecessary fork and join of parallel regions
  - try to read super-linear speed-up (better cache usage)

OpenMP: Pitfalls

- Implementation differences when moving platforms,
  - eg.N(threads), scheduling, …
- race condition:
  - >1 thread reads the same shared variable unsynchronised and min. one does writes
    → outcome depends on timing of the threads
  - reason: unintentional sharing of variables
    → use clause “default(none)”
- deadlock:
  - threads wait endlessly on a locked resource that will never be released
    → try to avoid locks – and if needed: do not nest

```
//Example race condition without
warning
#pragma omp parallel sections
{
#pragma omp section
                a= b+c;
#pragma omp section
                b = c+a;
#pragma omp section
                c = a+b;
}
printf(“%d %d %d”, a, b, c);
```

OpenMP: Pitfalls

- Missing barriers: add barrier if in doubt
- Write OpenMP code that is compatible with single core code
- “sequential equivalence”, two forms
    1. Strong SE: bitwise identical results
       (can be tested with clause “ordered” for loops)
    2. Weak SE: mathematically equivalent, but not bit wise
       (due to limited accuracy of floating point operations)
- When using threads and OpenMP, tell the compiler to use thread safe libraries.

Performance Computing

Sneak preview

RAM Access Pattern (1)

Example 1

```
int sum = 0;
int a[3][3];

for (int row = 0; row<n; row++) {
  for (int col = 0; col<n; col++) {
    sum += a[row, col];
  }
}
```

better!

Example 2

```
int sum = 0;
int a[3][3];

for (int col = 0; col<n; col++) {
  for (int row = 0; row<n; row++) {
    sum += a[row, col];
  }
}
```

Is there a difference?

Representation in memory (1 dimensional):

a[0,0]	a[0,1]	a[0,2]	a[1,0]	a[1,1]	a[1,2]	a[2,0]	a[2,1]	a[2,2]

RAM Access Pattern (2)

Example 1

```
int sum = 0;
int a[3][3];

for (int row = 0; row<n; row++) {
  for (int col = 0; col<n; col++) {
    sum += a[row, col];
  }
}
```

fast!

RAM access pattern
eg size(cache line) = 3 ints

in this example, real world core i7: 64bytes = 16 ints

1. loads first cache line:

   a[0,0] a[0,1] a[0,2]

2. computes
3. cache miss, loads next cache line:

   a[1,0] a[1,1] a[1,2]

4. computes
5. cache miss, loads next cache line:

   a[2,0] a[2,1] a[2,2]

RAM Access Pattern (3)

Example 2

```
int sum = 0;
int a[3][3];

for (int col = 0; col<n; col++) {
  for (int row = 0; row<n; row++) {
    sum += a[row, col];
  }
}
```

factor 10 slower

RAM access pattern
size(cache line) = 3 ints

1. loads first cache line:
   a[0,0] a[0,1] a[0,2]
2. computes one element
3. cache miss, loads next cache line:
   a[1,0] a[1,1] a[1,2]
4. computes one element
5. cache miss, loads next cache line:
   a[2,0] a[2,1] a[2,2]
6. computes one element

up to now, only 3 elements have been processed.

RAM Access Pattern

- Interplay with:
  - Optimising compilers
  - Out-of-order execution
  - Speculation
  - Load-Store Optimisations
  - → Result optimal/bad is not as bad as expected (T(cache)/T(RAM)).

- Non optimal memory access:
  → not fixed by OpenMP compiler!
- Inner loop should use contiguous index in the array
  - second index in C and Python (incl. NumPy) (“row-major”)
  - first index in Fortran (“column-major”)
  - other languages different
- Also true for similar memory access, not only loops

Set up your workbench

- Read the latest hints online:
  https://gitlab.rlp.net/pbotte/learnhpc/tree/master/openMP

Basic concept:

- Connect 2 times to Mogon2 / HIMster2 via SSH
  1) connection for your editor (gedit, vi, vim, nano, geany, …)
  2) second connection for compiling and running on compute node:
  ```
  srun --pty -p parallel -N 1 --time=02:00:00 -A m2_himkurs --reservation=himkurs bash
  ```
  - (no analysis on the head node!)
  - Run with: OMP_NUM_THREADS=4 ./pi
- Download the files:
  1) first time: git clone https://gitlab.rlp.net/pbotte/learnhpc.git
  2) only update: git pull
  - Check for directory: openMP/exercise3/

Hints:

- “git pull” does not work? To reset your git repository to the master: “git reset –hard”
- Check compiler version: cc -v
- Run: OMP_NUM_THREADS=4 ./pi
  or export OMP_NUM_THREADS 4
- Possible to check reservation with: squeue -u $USER

Exercise 5: Reduction

Learning objectives:

- Use of reduction clause

Steps:

1. Start with either use your result or download a starting point from lecture webpage:
   - wget https://www.hi-mainz.de/fileadmin/user_upload/IT/lectures/WiSe2018/HPC/files/04.zip && unzip 04.zip
2. Simplify example 4 step 6 by using the reduction clause.
3. Try different operators.
4. Bonus:
   1. Read https://en.wikipedia.org/wiki/Double-precision_floating-point_format
   2. Why does the result differ for OPM_NUM_THREADS=1 and =4 in the last digits?

Exercise 6: RAM access pattern

Learning objectives:

- Use of right RAM access pattern

Steps:

1. Write a c program with the both codes from slide: “RAM Access Pattern”
2. Add the CPU-timing from exercise 5
3. test with different total array numbers:
   9, 1E6, 10E6, 100E7 and give the ratio between row-wise and col-wise runtime.

Solutions

Solution 5 (1)

```
//Solution for Exercise 5
#include <stdio.h>
#include <time.h>
#include <sys/time.h>
#ifdef _OPENMP
# include <omp.h>
#endif

#define f(A) (4.0/(1.0+A*A))

const int n = 1E9;

int main() {
int i;
double w, x, sum, pi;
//Variables for timing single core
clock_t t1,t2;
struct timeval tv1,tv2;
struct timezone tz;
```

```
# ifdef _OPENMP
double wt1,wt2; //For Timing with OpenMP
# endif

# ifdef _OPENMP
# pragma omp parallel
{
# pragma omp single
printf("OpenMP-parallel with %1d threads\n",
omp_get_num_threads());
} /* end omp parallel */
# endif

//Do Start of Timing
gettimeofday(&tv1, &tz);
# ifdef _OPENMP
wt1 = omp_get_wtime();
# endif
t1 = clock();
```

Solution 5 (2)

```
/* calculate pi = integral [0..1]
4/(1+x**2) dx */
w = 1./n;
sum = 0;
#pragma omp parallel private(x)
shared(w,sum)
{
# pragma omp for reduction(+:sum)
for (i=1;i<=n;i++)
{
x=w*((double)i-0.5);
sum=sum+f(x);
}
} /*end omp parallel*/
pi=w*sum;
```

```
//Do End of Timing
t2 = clock();
# ifdef _OPENMP
wt2 = omp_get_wtime();
# endif
gettimeofday(&tv2, &tz);
printf("computed pi = %24.16g\n", pi);
printf("CPU time (clock) = %12.4g sec\n",
(t2-t1)/1000000.);
# ifdef _OPENMP
printf("wall clock time (omp_get_wtime) =
%12.4g sec\n", wt2-wt1);
# endif
printf("wall clock time (gettimeofday) =
%12.4g sec\n", (tv2.tv_sec-tv1.tv_sec) +
(tv2.tv_usec-tv1.tv_usec)*1e-6 );
}
```