HPC Programming

OpenMP, Part III

Peter-Bernd Otte, 13.11.2018

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: 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`

```
#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”)

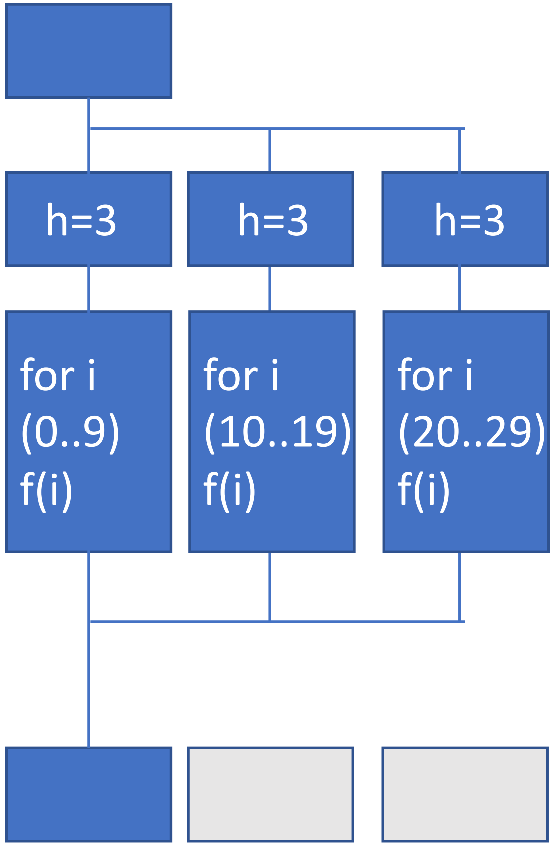
OpenMP: for Directive

- 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
```

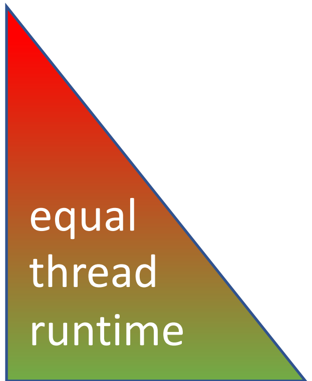
- Clauses:
  - reduction (op: list)
  - collapse (n)
    (n=const.: iterations of following n nested loops are collapsed into one larger iteration space)
  - schedule (type, chunk)
    (how the work is divided among the threads)

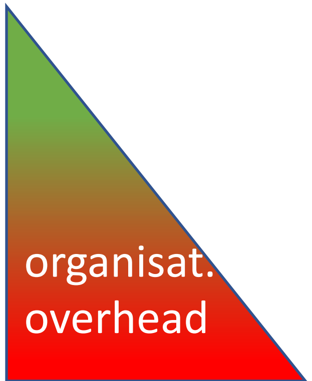


OpenMP: for Directive, scheduling

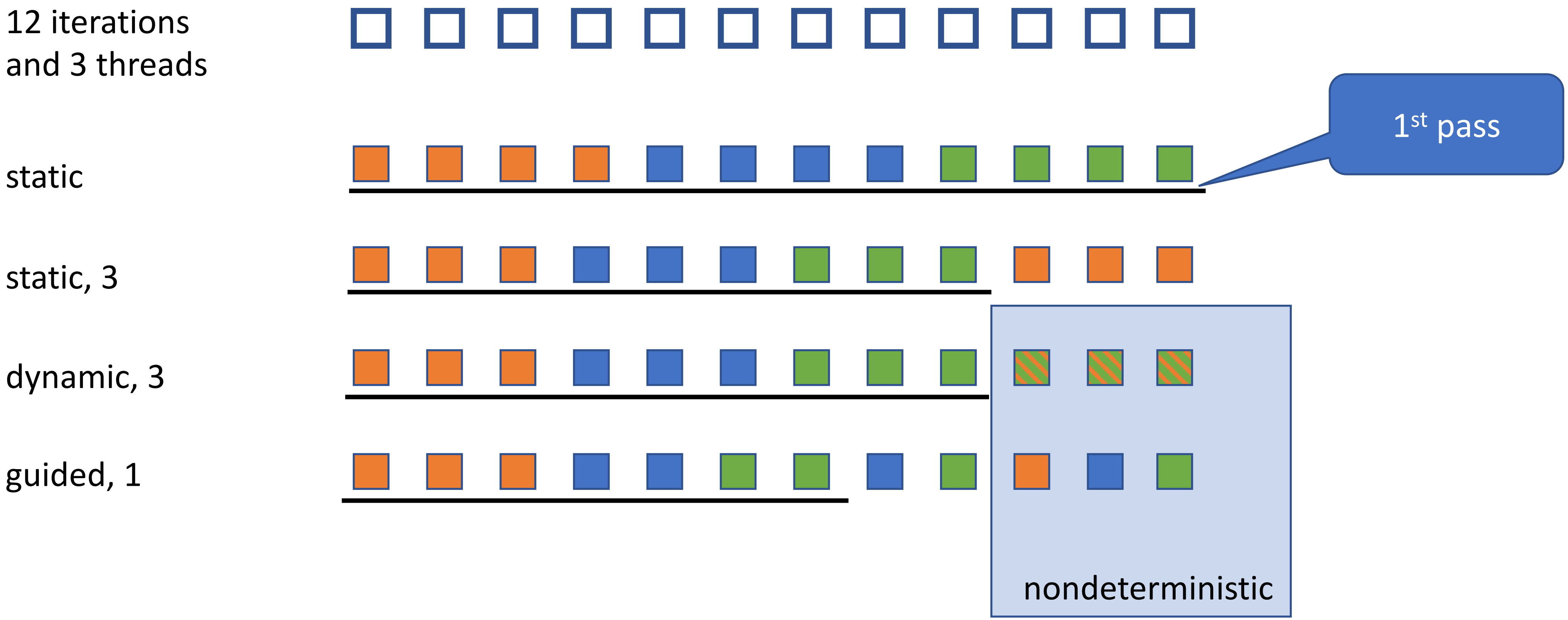
- How the work (*n* iterations) is divided among the *p* threads
  - Clause: schedule (type[, chunk])
- Type:
  - static: one chunk per thread with equal n / or with chunk size provided: chunks are statically assigned to threads.
  - dynamic: threads obtain chunks of size c when free (default: c=1 iteration).
  - 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*





OpenMP: for Directive, scheduling



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. */
  - 10 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
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 4, step 6

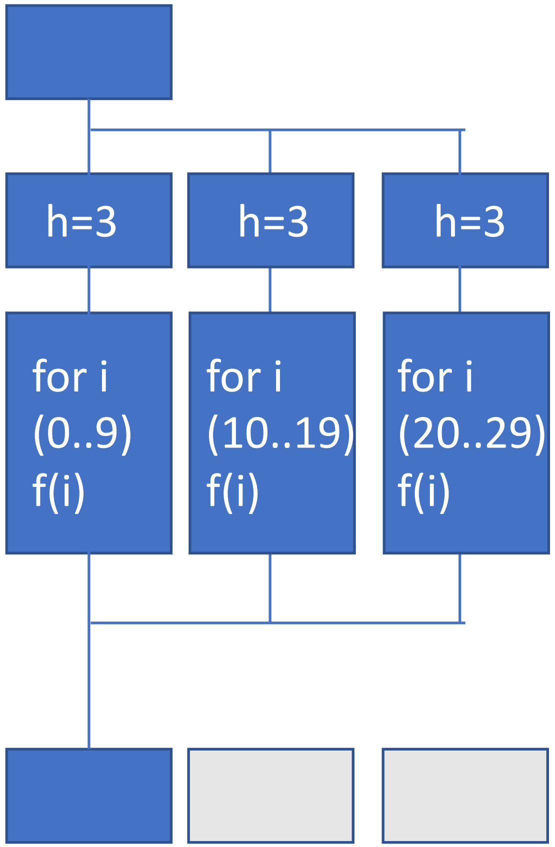
OpenMP: reduction clause (2)

```
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
5. Data environment and combined constructs
6. Common pitfalls and good practice (“need for speed”)

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?

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
size(cache page) = 3 ints

1. loads first cache page:

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

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

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

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

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 page) = 3 ints

1. loads first cache page:

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

2. computes one element
3. cache miss, loads next cache page:

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

4. computes one element
5. cache miss, loads next cache page:

   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

- Access pattern automatically optimised by serial compiler
- Not fixed by OpenMP compiler!
- Inner loop should use contiguous index in the array (second index in C, Fortran 1st, other languages different)
- Also true for similar memory access, not only loops

- See Chapter 8 in OpenMP Book Springer

OpenMP: Need for Speed

- 1^{st}: 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
    - 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.

Set up your workbench

- Connect 2 to Mogon2 / HIMster2 via SSH
  srun --pty -p parallel -N 1 --time=02:00:00 -A m2_himkurs --reservation=himkurs -C skylake bash -i
  1. Use the first SSH connection for editing (gedit, vi, vim, nano, geany) and
     compiling: cc -fopenmp -o ExecutableName SourceFileName.c
  2. Use the second connection for the interactive execution on the nodes (no execution on the head node!):
     OMP_NUM_THREADS=4 ./ExecutableName

- Download the files via: wget https://www.hi-mainz.de/fileadmin/user_upload/IT/lectures/WiSe2018/HPC/files/02.zip && unzip 02.zip

Hints:

- Check compiler version: cc -V
- Run: OMP_NUM_THREADS=4 ./ExecutableName
  or export OMP_NUM_THREADS 4
- Possible to check reservation with: squeue -u USERNAME

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.