MOVING PARALLEL WITH OPENMP

Juan Pablo Mallarino

Facultad de ciencias

Universidad de los Andes

jp.mallarino50@uniandes.edu.co

June 11, 2019

Agradecimientos a la Universidad del Rosario

1. A brief introduction on computation
    - Evolution of infrastructure
    - Why HPC?
2. How does parallelization works
    - Strategies
    - Ahmdal’s Law
3. OpenMP: Preliminaries
    - Compiling
    - Examples
4. Real Scenarios
    - Practice
    - Advanced topics
5. Bibliography

the big question

Computation

What is computation?

the big question

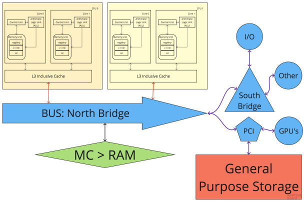


Figure: Von Neumann Architechture

Computation

What is computation?

Key infrastructure components

- Storage
- RAM
- Processing block: registries, instruction sets and clock
- FPGA's, GPU's, accelerators and other alternate processing units (RaspBerries, portable devices . . . ARM)
- Compilers - Translator to Machine language

the big question

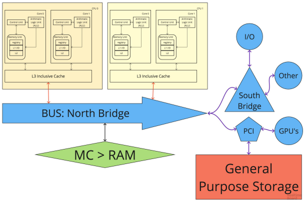


Figure: Von Neumann Architechture

Computation

What is computation?

Key infrastructure components

- Storage
- RAM
- Processing block: registries, instruction sets and clock
- FPGA's, GPU's, accelerators and other alternate processing units (RaspBerries, portable devices . . . ARM)
- Compilers - Translator to Machine language

Limitations & Complications

1. All of the above
2. Education: infrastucture topology, coding strategies, profiling & optimization
3. Interpreted languages
4. Unix like systems
5. Time - accelerating technologies and real-time applications
6. Threats ⟹ Cybersecurity (https://meltdownattack.com/)

the big question

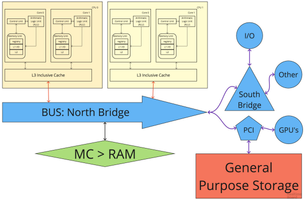


Figure: Von Neumann Architechture

History of parallelism

1. Origin dates back to the 80's
2. ILP + Vectorization: the superscalar architecture
3. Memory complexity:

CPU	$\sim 0.5ns$	$1\times$

Table: By Jeff Dean@Google: http://research.google.com/people/jeff/

4. Memory coherency: *(i.)* HW with ECC *(ii.)* Software
5. Memory topology: UMA & NUMA & *cc*-NUMA

the big question

History of parallelism

7. 1996 SGI bought CRAY and soon after formed the ARB. 1997 OpenMP was born and announced at the New York Times.
8. CPU processor development stalled: *(i.)* Quantum limit $\sim 9nm$ *(ii.)* Energy efficiency per FLOP kept dropping
9. A full scale development of *multi*-core processors
10. Later: GPGPU's & MIC's & FPGA's

using HPC infrastructure

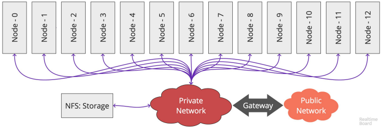


Figure: A cluster

Why the cluster ?

- † Larger Storage
- † Bigger available RAM space (V-space)
- † More CPU cores per computing unit
- † Heterogeneous computing provisioning
- ‡ Software
- ‡ Fault-safe checks (ECC)
- ‡ Connectivity
- ‡ Service availability (power & hardware vendor support)
- ⋆ Efficient use of resources (power cost per FLOP)
- ⋆ Research on topological improvements for high cost/effective throughput
- △ People, science & culture

⋆'s and △ is HPC

using HPC infrastructure

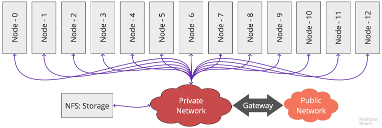


Figure: A cluster

Why the cluster ?

- † Larger Storage
- † Bigger available RAM space (V-space)
- † More CPU cores per computing unit
- † Heterogeneous computing provisioning
- ‡ Software
- ‡ Fault-safe checks (ECC)
- ‡ Connectivity
- ‡ Service availability (power & hardware vendor support)
- ⋆ Efficient use of resources (power cost per FLOP)
- ⋆ Research on topological improvements for high cost/effective throughput
- △ People, science & culture

⋆'s and △ is HPC

Advantages

1. Size of problems - RealTime Operations
2. Encryption, Meteorology, Machine Learning, Block Chain
3. Smarter code, faster or better calculations
4. Precision (Accelerators)
5. Do you need to continuously upgrade your computer?

Basics

Definition

A parallel computer is a system that is able to execute simultaneously multiple processing elements cooperatively to solve a computational problem

Requirements

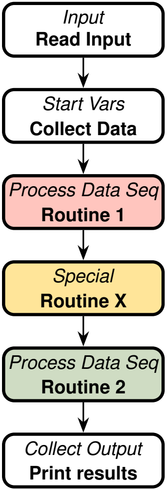
- □ Hardware
- □ OS
- □ Libraries: PThreads, TBB, OpenMP
- □ For HPC: Understand the process and data distribution model

Terminology

- △ Concurrent: A program is one in which multiple tasks can be *in progress* at any instant. Or in *multiple*-THREADS!
- △ Parallel: A program is one in which multiple tasks *cooperate closely* to solve a problem.
- △ Distributed: A program may need to cooperate with other programs to solve a problem.

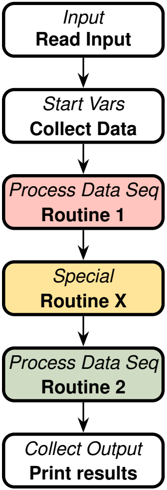
Pacheto **An instroduction to Parallel Programming**, Elsevier (2011)

The idea?



- The idea is to identify opportunities of parallelism
- Develop the application to exploit parallelism
- Run the application: identify Bugs and Improvements
- Use resources efficiently

The idea?

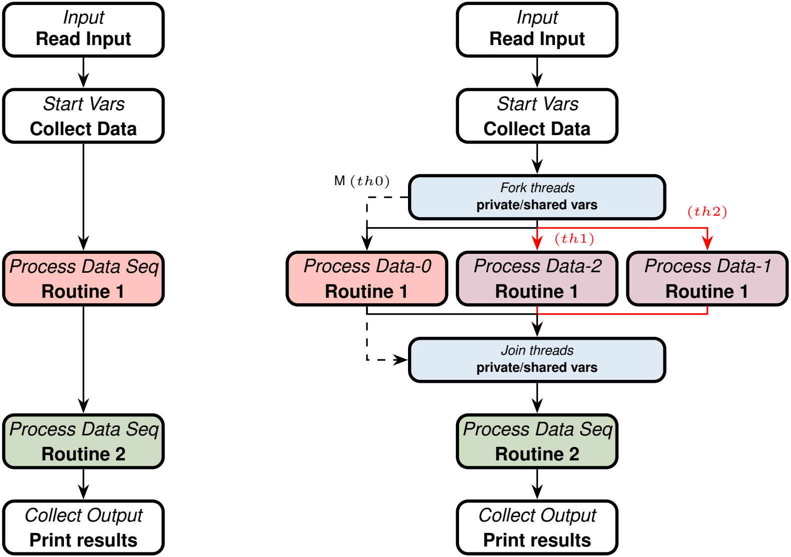


- The idea is to identify opportunities of parallelism
- Develop the application to exploit parallelism
- Run the application: identify Bugs and Improvements
- Use resources efficiently

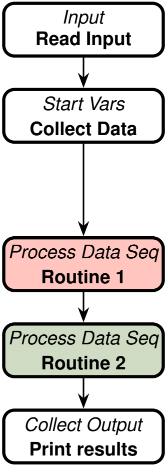
API and Libraries

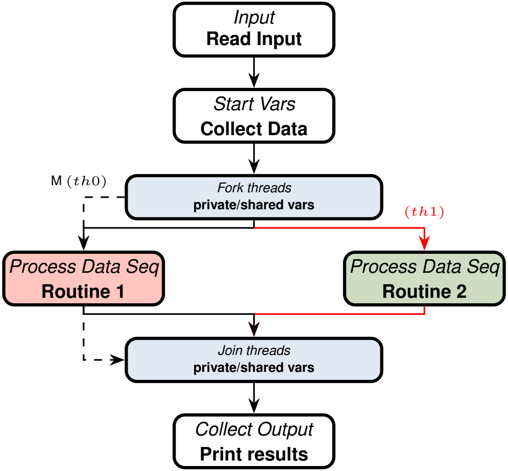
- ⋆ Pthreads (HARD): routines & variables
- ⋆ OpenMP: pragmas, routines & variables
- ⋆ Comes for C/C++ and Fortran

The idea parallelized: Data



The idea parallelized: Task





Ahmdal's Law

Due to the overheads, parallelism is only achieved at a certain level. Let t be the execution time of a sequential application, then

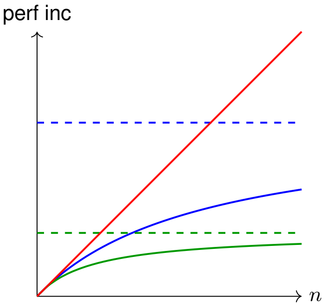
$$t = t_s + t_p,$$

with t_s and t_p the time of sequential and parallel parts. If the data or tasks are going to be split equally into n threads, then the new time t' is written as,

$$t' = t_s + n\,\delta t_{\text{oh}} + \frac{t_p}{n},$$

where δt_{oh} is the overhead time required to fire up each thread (approximately linear). Defining $f := \frac{t_s}{t_p}$,

$$\text{perf inc} = \frac{t}{t'} = n\frac{1+f}{1+n\,f+n^2\delta t_{\text{oh}}} \leqslant 1 + \frac{1}{f}$$



Open Multi Processing: preliminaries

Data environment/Declarations

1. `#pragma omp`
2. `#pragma omp threadprivate`
3. `shared/*private`
4. clauses
5. `master /threads`

Parallel construct structures

1. `omp parallel`

clauses / options

1. $\mathrm{if}(\dots)$
2. $\mathrm{num\_threads}(\dots)$
3. $\mathrm{private}(\dots)$
4. $\mathrm{shared}(\dots)$
5. $\mathrm{firstprivate}(\dots)$:
   $x(t=0) = x$ before construct
6. $\mathrm{default}(\dots)$: `none|shared`
7. $\mathrm{copyin}(\dots)$:
   x private on master copied to threads privates
8. $\mathrm{reduction}(operator|list)$

Open Multi Processing: preliminaries

Data environment/Declarations

1. `#pragma omp`
2. `#pragma omp threadprivate`
3. `shared/*private`
4. clauses
5. `master` /`threads`

Work sharing

1. `omp [parallel] for`
2. `omp [parallel] sections`
3. Only Fortran: `omp [parallel] workshare`
4. `omp single`

loop clauses / options

Can be combined with the *parallel* construct

1. private(...)
2. firstprivate(...)
3. lastprivate(...):
   $x(t = t_f) = x$ last "loop" value
4. reduction($operator|list$)
5. ordered(...):
   if ordered construct inside parallel region!
6. schedule($kind[, chunk\_size]$):
   static,dynamic,guided,runtime,auto
7. nowait:

sections clauses / options

Can be combined with the *parallel* construct
private+firstprivate)+lastprivate)+
reduction+nowait

single clauses / options

1. private+firstprivate)+nowait
2. copyprivate(...): x private in thread to threads

Open Multi Processing: preliminaries

Data environment/Declarations

1. #pragma omp
2. #pragma omp threadprivate
3. shared/$\star$private
4. clauses
5. master /threads

Synchronization

1. omp barrier:
   Look semaphores
2. omp ordered:
   only for *loops*!
3. omp critical $[(name)]$:
   is a block!
   Could use hints!
4. omp atomic: only a statement < 3.1!
5. omp master:
   no barrier at the end!
6. omp flush: enforces data consistency
   relaxed consistency model
7. omp task:
   called from within *single* construct
8. omp taskwait

atomic clauses / options

1. $\mathrm{read}(\ldots)$
2. $\mathrm{write}(\ldots)$
3. $\mathrm{update}(\ldots)$
4. $\mathrm{capture}(\ldots)$

Open Multi Processing: preliminaries

Data environment/Declarations

1. #pragma omp
2. #pragma omp threadprivate
3. shared/⋆private
4. clauses
5. master /threads

Data environment

1. *Routines* (or functions)
2. ENVIRONMENT VARIABLES

useful routines

1. omp_set_max_threads()
2. omp_get_max_threads()
3. omp_get_num_threads()
4. omp_get_num_devices()
5. omp_get_thread_num()
6. omp_get_thread_limit()
7. omp_set_nested()
8. omp_get_nested()
9. omp_get_schedule()
10. omp_get_wtime()
11. omp_in_parallel()
12. omp_init_lock()
13. omp_init_nest_lock()
14. omp_destroy_lock()
15. omp_destroy_nest_lock()
16. omp_test_lock()
17. omp_test_nest_lock()
18. omp_set_lock()
19. omp_set_nest_lock()
20. omp_unset_lock()
21. omp_unset_nest_lock()

Open Multi Processing: preliminaries

Data environment/Declarations

1. `#pragma omp`
2. `#pragma omp threadprivate`
3. `shared/*private`
4. clauses
5. `master` /`threads`

Data environment

1. *Routines* (or functions)
2. ENVIRONMENT VARIABLES

useful *ENV VARS*

1. `OMP_THREAD_LIMIT`
2. `OMP_NUM_THREADS`
3. `OMP_DYNAMIC`
4. `OMP_NESTED`
5. `OMP_SCHEDULE`

Open Multi Processing: preliminaries

Look at the reference guide!

How do we compile?

OpenMP version

"As of GCC 4.2, the compiler implements version 2.5 of the OpenMP specification, as of 4.4 it implements version 3.0 and since GCC 4.7 it supports the OpenMP 3.1 specification. GCC 4.9 supports OpenMP 4.0 for C/C++, GCC 4.9.1 also for Fortran. GCC 5 adds support for Offloading."

GCC:
```
gcc -std=c++11 ex6-loop-reduction.cpp -o exe
-fopenmp
```

Intel Compiler:
```
icc -std=c++11 ex6-loop-reduction.cpp -o exe
-qopenmp
```

Simple example #1: ex1-hostname

Source Code 1: Printing hostname with Master or Single!

```
 1  #define INFO_BUFFER_SIZE 1024
 2  int main(int argc, char *argv[]){
 3  #ifdef _OPENMP
 4      printf("**MESSAGE** OpenMP enabled\n");
 5      (void) omp_set_dynamic(FALSE);
 6      (void) omp_set_num_threads(4);
 7  #else
 8      printf("**MESSAGE** OpenMP disabled\n");
 9  #endif
10      char hostname[INFO_BUFFER_SIZE];
11      char username[INFO_BUFFER_SIZE];
12      #pragma omp parallel
13      {
14      gethostname(hostname, INFO_BUFFER_SIZE);
15      getlogin_r(username, INFO_BUFFER_SIZE);
16      printf("Hostname %s in thread
          ↪   %d\n",hostname,omp_get_thread_num());
17      printf("Username %s in thread
          ↪   %d\n",username,omp_get_thread_num());
18      } /*-- End of parallel region --*/
19      return 0;
20  }
```

Notice

1. The region enclosed in `#ifdef`
2. Manually setting the # of threads
3. Delared the parallel construct (l.12-18)
4. We are using `printf`
5. Prints the thread numbers $[0, 1, 2, 3]$

Simple example #1: ex1-hostname

Source Code 2: Printing hostname with Master or Single!

```
    #pragma omp parallel
    {
    gethostname(hostname, INFO_BUFFER_SIZE);
    getlogin_r(username, INFO_BUFFER_SIZE);
    cout<<"Hostname "<<hostname<<" in thread
  ↪   "<<omp_get_thread_num()<<endl;
    cout<<"Username "<<username<<" in thread
  ↪   "<<omp_get_thread_num()<<endl;
    } /*-- End of parallel region --*/
```

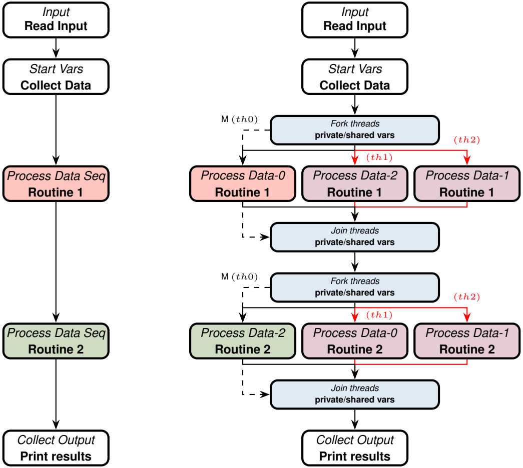
Notice

1. The region enclosed in `#ifdef`
2. Manually setting the # of threads
3. Delared the parallel construct (l.12-18)
4. We are using `printf`
5. Prints the thread numbers $[0, 1, 2, 3]$

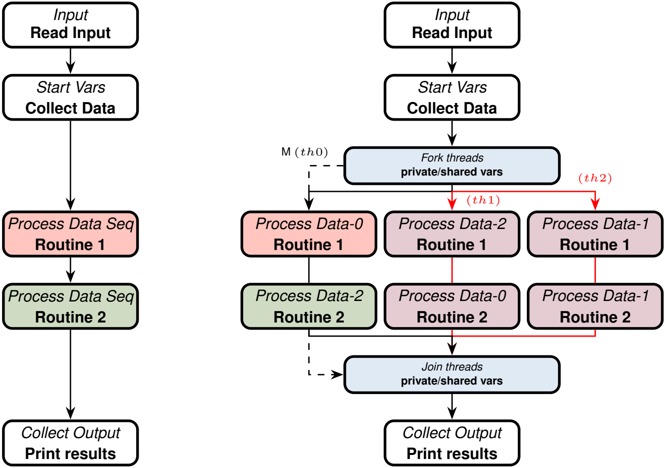
Problem

1. Disordered output! `std::cout`

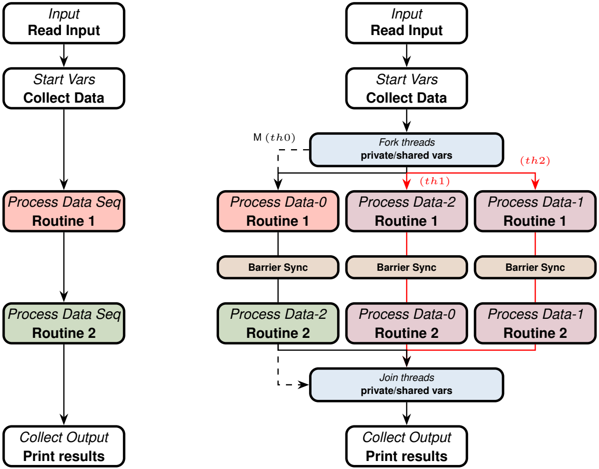
Interesting scenarios!



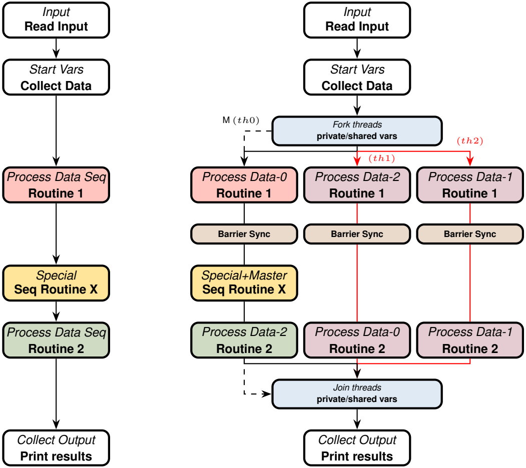
Interesting scenarios!



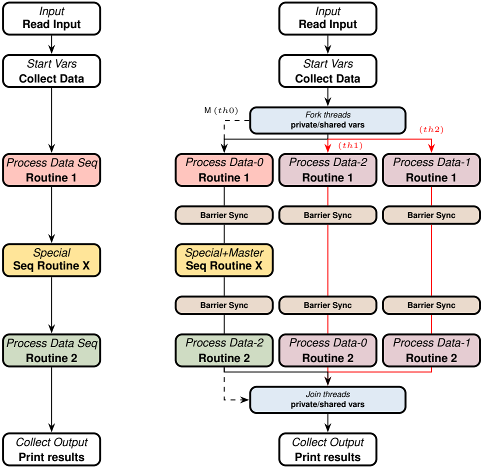
Interesting scenarios!



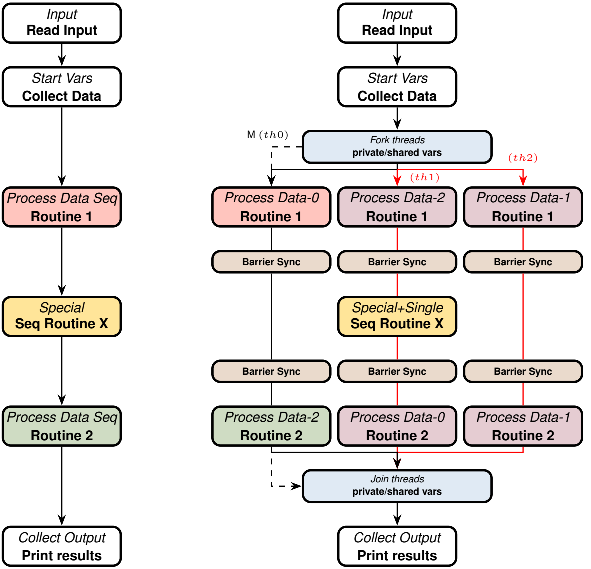
Interesting scenarios!



Interesting scenarios!



Interesting scenarios!



Down to business !

Simple example #2: ex2-master-single

Source Code 3: Printing thread number and Sleep!

```
 1  #define INFO_BUFFER_SIZE 1024
 2  int main(int argc, char *argv[]){
 3  #ifdef _OPENMP
 4      printf("**MESSAGE** OpenMP enabled\n");
 5      (void) omp_set_dynamic(FALSE);
 6      (void) omp_set_num_threads(4);
 7  #else
 8      printf("**MESSAGE** OpenMP disabled\n");
 9  #endif
10      #pragma omp parallel
11      {
12          int TID = omp_get_thread_num();
13          sleep(omp_get_thread_num());
14
15          printf("In parallel region - Thread ID is
              ↪   %d\n",TID);
16      } /*-- End of parallel region --*/
17      return 0;
18  }
```

Notice

1. The region enclosed in `#ifdef`
2. Manually setting the # of threads
3. Delared the parallel construct (l.10-16)
4. Sets the sleep according to thread #

Simple example #2: ex2-master-single

Source Code 4: Printing thread number and Sleep!

```
1      int extra_time = 0;
2      #pragma omp parallel shared(extra_time)
3      {
4          int TID = omp_get_thread_num();
5          #pragma omp master
6          {
7              printf("\tInside Block - Thread ID is
                ↪   %d\n",TID);
8              sleep(1);
9              extra_time = 1;
10         }
11         sleep(TID+extra_time);
12         printf("In parallel region - Thread ID is
            ↪   %d\n",TID);
13     } /*-- End of parallel region --*/
```

Notice

1. The region enclosed in `#ifdef`
2. Manually setting the # of threads
3. Delared the parallel construct (l.10-16)
4. Sets the sleep according to thread #

Problem

1. Cause: value not updated soon enough

Simple example #2: ex2-master-single

Source Code 5: Printing thread number and Sleep!

```
    int extra_time = 0;
    #pragma omp parallel shared(extra_time)
    {
        int TID = omp_get_thread_num();
        #pragma omp single
        {
            printf("\tInside Block - Thread ID is %d\n",TID);
            sleep(1);
            extra_time = 1;
        }
        sleep(TID+extra_time);
        printf("In parallel region - Thread ID is %d\n",TID);
    } /*-- End of parallel region --*/
```

Notice

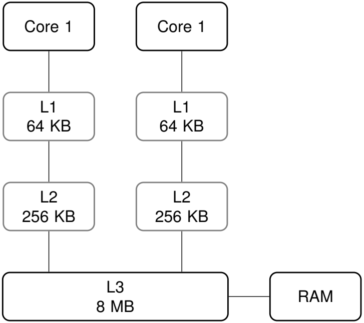
1. The region enclosed in `#ifdef`
2. Manually setting the # of threads
3. Delared the parallel construct (l.10-16)
4. Sets the sleep according to thread #

Problem

1. Corrected

What is the issue?

- Each thread is a copy of the original $th0$
- Private variables have redundant "private" memmory addresses
- Used values are load to cache! Not all cache is shared
- Updates have to be enforced for data coherency!
- How? `flush` or `barrier`.



Simple example #3: ex3-copy

Source Code 6: Private vars and firstprivate!

```
 1  #define INFO_BUFFER_SIZE 1024
 2  int main(int argc, char *argv[]){
 3  #ifdef _OPENMP
 4      printf("**MESSAGE** OpenMP enabled\n");
 5      (void) omp_set_dynamic(FALSE);
 6      (void) omp_set_num_threads(4);
 7  #else
 8      printf("**MESSAGE** OpenMP disabled\n");
 9  #endif
10      int x = 10;
11      printf("thread %d original value:
          ↪   %d\n",omp_get_thread_num(),x);
12      #pragma omp parallel private(x)
13      {
14          int TID = omp_get_thread_num();
15          printf("thread %d value: %d\n",TID,x);
16      } /*-- End of parallel region --*/
17      printf("thread %d after value:
          ↪   %d\n",omp_get_thread_num(),x);
18      return 0;
19  }
```

Notice

1. The region enclosed in `#ifdef`
2. Manually setting the # of threads
3. Delared the parallel construct (l.11-19)
4. Prints zero inside parallel construct!
5. Prints again 10 after parallel construct!

Problem

1. Private value restarts at ZERO!

Simple example #3: ex3-copy

Source Code 7: Private vars and firstprivate!

```
#pragma omp parallel firstprivate(x)
{
    int TID = omp_get_thread_num();
    printf("thread %d value: %d\n",TID,x);
} /*-- End of parallel region --*/
```

Notice

1. The region enclosed in `#ifdef`
2. Manually setting the # of threads
3. Delared the parallel construct (l.11-19)
4. Prints zero inside parallel construct!
5. Prints again 10 after parallel construct!

Problem

1. Corrected, prints $x = 10$.

Special Exercise

Use `copyprivate` to copy private variable from `master` or `single` to other threads.

Simple example #4: ex4-shared

Source Code 8: Updating variable!

```
1  #define INFO_BUFFER_SIZE 1024
2  int main(int argc, char *argv[]){
3  #ifdef _OPENMP
4      printf("**MESSAGE** OpenMP enabled\n");
5      (void) omp_set_dynamic(FALSE);
6      (void) omp_set_num_threads(4);
7  #else
8      printf("**MESSAGE** OpenMP disabled\n");
9  #endif
10     int x = 10;
11     #pragma omp parallel
12     {
13         int TID = omp_get_thread_num();
14         x += 5;
15         printf("thread %d value: %d\n",TID,x);
16     } /*-- End of parallel region --*/
17     printf("Value: %d\n",x);
18     return 0;
19 }
```

Notice

1. The region enclosed in `#ifdef`
2. Manually setting the # of threads
3. Delared the parallel construct (l.11-16)
4. We are using `printf`
5. All threads add to x
6. x is shared

Problem

1. Every execution prints different values
2. RACE CONDITION

Simple example #5: ex5-loop

Source Code 9: Adding arrays!

```
#define INFO_BUFFER_SIZE 1024
int main(int argc, char *argv[]){
#ifdef _OPENMP
    printf("**MESSAGE** OpenMP enabled\n");
    (void) omp_set_dynamic(FALSE);
    (void) omp_set_num_threads(4);
#else
    printf("**MESSAGE** OpenMP disabled\n");
#endif
    Crandom r(10);
    double *v1,*v2,*res;
    CREATEARRAY(v1,double,N,r.gaussian(1,2));
    CREATEARRAY(v2,double,N,-1);
    CREATEARRAY(res,double,N,0);
    PRINTARRAY(v1,N);
    PRINTARRAY(v2,N);
    for(int j = 0; j < N; j++)
        res[j] = v1[j] + v2[j];
    PRINTARRAY(res,N);
    return 0;
}
```

Notice

1. The region enclosed in `#ifdef`
2. Manually setting the # of threads
3. Notice the strange definitions! defs.h
4. No parallel construct
5. Prints the arrays $v1$, $v2$ and $v1 + v2$

Simple example #5: ex5-loop

Source Code 10: Adding arrays!

```c
    SimpleTimer_t t1;
    PRINTARRAY(v1,N);
    PRINTARRAY(v2,N);
    SimpleTimer_start( &t1 );
    #pragma omp parallel shared(j,N,v1,v2,res)
    {
        #pragma omp for
        for(int j = 0; j < N; j++)
            res[j] = v1[j] + v2[j];
    } /*-- End of parallel region --*/
    SimpleTimer_stop( &t1 );
    SimpleTimer_print( &t1 );
    PRINTARRAY(res,N);
```

Notice

1. The region enclosed in `#ifdef`
2. Manually setting the # of threads
3. Notice the strange definitions! defs.h
4. No parallel construct
5. Prints the arrays $v1$, $v2$ and $v1 + v2$

Problem

1. Any comments?

Simple example #6: ex6-loop-reduction

Source Code 11: Norm of a vector!

```c
#define INFO_BUFFER_SIZE 1024
int main(int argc, char *argv[]){
#ifdef _OPENMP
    printf("**MESSAGE** OpenMP enabled\n");
    (void) omp_set_dynamic(FALSE);
    (void) omp_set_num_threads(4);
#else
    printf("**MESSAGE** OpenMP disabled\n");
#endif
    int j,N = 200;
    Crandom r(10);
    double *v1;
    CREATEARRAY(v1,double,N,r.gaussian(0,0.5));
    PRINTARRAY(v1,N);
    double norm_v1 = 0;
    for(j = 0; j < N; j++)
        norm_v1 += v1[j]*v1[j];
    printf("v1 norm = %lf\n",norm_v1);
    return 0;
}
```

Notice

1. The region enclosed in `#ifdef`
2. Manually setting the # of threads
3. Notice the strange definitions! defs.h
4. No parallel construct
5. Prints the quadratic norm of array $|v1| = 59.725\ldots$

Simple example #6: ex6-loop-reduction

Source Code 12: Norm of a vector!

```
    #pragma omp parallel shared(j,N,norm_v1,v1)
    {
        #pragma omp for
        for(j = 0; j < N; j++)
            norm_v1 += v1[j]*v1[j];
    } /*-- End of parallel region --*/
    // OR...
    #pragma omp parallel for shared(j,N,norm_v1,v1)
    for(j = 0; j < N; j++)
        norm_v1 += v1[j]*v1[j];
```

Notice

1. The region enclosed in `#ifdef`
2. Manually setting the # of threads
3. Notice the strange definitions! defs.h
4. Parallel workshare construct
5. Prints the quadratic norm of array $|v1| = 59.725\ldots$

Problem

1. Norm doesn't provide reliable results!

Simple example #6: ex6-loop-reduction

Source Code 13: Norm of a vector!

```
#pragma omp parallel shared(N,norm_v1,v1)
{
    double tmp = 0;
    #pragma omp for
    for(j = 0; j < N; j++){
        tmp = v1[j]*v1[j];
        #pragma omp critical
        {
            norm_v1 += tmp;
        }
    }
} /*-- End of parallel region --*/
// OR...
#pragma omp parallel for shared(N,norm_v1,v1)
for(j = 0; j < N; j++)
    #pragma omp critical
    norm_v1 += v1[j]*v1[j];
```

Notice

1. The region enclosed in `#ifdef`
2. Manually setting the # of threads
3. Notice the strange definitions! defs.h
4. Parallel workshare construct
5. Prints the quadratic norm of array $|v1| = 59.725\ldots$

Problem

1. Solved with: critical

Simple example #6: ex6-loop-reduction

Source Code 14: Norm of a vector!

```
#pragma omp parallel shared(N,norm_v1,v1)
{
    double tmp = 0;
    #pragma omp for
    for(j = 0; j < N; j++){
        tmp = v1[j]*v1[j];
        #pragma omp atomic
        norm_v1 += tmp;
    }
} /*-- End of parallel region --*/
// OR...
#pragma omp parallel for shared(N,norm_v1,v1)
for(j = 0; j < N; j++)
    #pragma omp atomic
    norm_v1 += v1[j]*v1[j];
```

Notice

1. The region enclosed in `#ifdef`
2. Manually setting the # of threads
3. Notice the strange definitions! defs.h
4. Parallel workshare construct
5. Prints the quadratic norm of array $|v1| = 59.725\ldots$

Problem

1. Solved with: atomic

Simple example #6: ex6-loop-reduction

Source Code 15: Norm of a vector!

```
#pragma omp parallel shared(N,norm_v1,v1)
{
    #pragma omp for reduction(+:norm_v1)
    for(j = 0; j < N; j++)
        norm_v1 += v1[j]*v1[j];
} /*-- End of parallel region --*/
// OR...
#pragma omp parallel for shared(N,norm_v1,v1) reduction(+:norm_v1)
for(j = 0; j < N; j++)
    norm_v1 += v1[j]*v1[j];
```

Notice

1. The region enclosed in `#ifdef`
2. Manually setting the # of threads
3. Notice the strange definitions! defs.h
4. Parallel workshare construct
5. Prints the quadratic norm of array $|v1| = 59.725\ldots$

Problem

1. Solved with: reduction

Simple example #6: ex6-loop-reduction

Source Code 16: Norm of a vector!

```
#pragma omp parallel shared(N,norm_v1,v1)
{
    double tmp = 0;
    #pragma omp for ordered
    for(j = 0; j < N; j++){
        tmp = v1[j]*v1[j];
        #pragma omp ordered
        {
            norm_v1 += tmp;
        }
    }
} /*-- End of parallel region --*/
// OR...
#pragma omp parallel for shared(N,norm_v1,v1) ordered
for(j = 0; j < N; j++)
    #pragma omp ordered
    norm_v1 += v1[j]*v1[j];
```

Notice

1. The region enclosed in `#ifdef`
2. Manually setting the # of threads
3. Notice the strange definitions! defs.h
4. Parallel workshare construct
5. Prints the quadratic norm of array $|v1| = 59.725\ldots$

Problem

1. Solved with: ordered

Exercise 1

Problem statement: Normalize array $v1/\sqrt{|v1|}$ into $v1$

Exercise 2

Problem statement: Find the determinant and inverse of a matrix

Exercise 3

Problem statement: Integrate an arbitrary real 1D function

Exercise 4

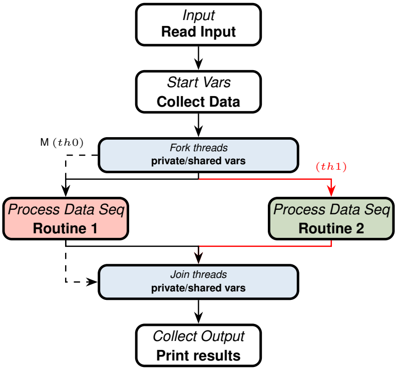
Problem statement: Evaluate the Fourier Transform

$$c_k = \frac{1}{2\pi} \int f(x) e^{-i\ k\ x} \mathrm{d}x \tag{1}$$

Exercise 5

Sort an array of floats or integers! Create a `swap` function that takes two parameters (pass by reference) and does the swapping. Then write the Sequential code. Finally, parallelize it! What is the order of the problem? *Hint:* N^p.

Back to the parallelized Task idea: ex7-task-palindrome/ex8-fibonacci



Input
Read Input

Start Vars
Collect Data

M $(th0)$

Fork threads
private/shared vars

$(th1)$

Process Data Seq
Routine 1

Process Data Seq
Routine 2

Join threads
private/shared vars

Collect Output
Print results

Source Code 17: The word palindrome!

```
printf("A ");
printf("race ");
printf("car ");
printf("is fun to watch.\n");

#pragma omp parallel
{
    #pragma omp single
    {
        printf("A ");
        #pragma omp task
        { printf("race "); }
        #pragma omp task
        { printf("car "); }
        #pragma omp taskwait
        printf("is fun to
          ↪   watch.\n");
    }
}
```

Exercise 6

Problem statement: Find random numbers using all threads

1. `nesting`: delicate!
2. `cancel`: The analog of break or exception handling
3. `simd`
4. `device`: For GPU's and GP devices

Bibliography

Special thanks to ALL of you!

- **USING OpenMP - The Next Step**, *Ruud van der Pas, et. al.*, MIT Press (2017)
- **Introduction to Parallel Programming**, *Peter S. Pacheco*, Elsevier (2011)
- **Deep Learning**, *Ian Goodfellow, et. al.*, MIT Press (2016)
- **Structured Parallel Programming**, *Michael McCool, et. al.*, Elsevier (2012)

(Bonus-Challenge): Machine Learning RBM's

Problem statement: Evaluate $\langle E \rangle$ for a distribution $\rho = \frac{1}{Z} e^{-E}$ with

$$E(\mathbf{v}, \mathbf{h}) = -\mathbf{b}^{\mathsf{T}}\mathbf{v} - \mathbf{c}^{\mathsf{T}}\mathbf{h} - \mathbf{v}^{\mathsf{T}}\mathbf{W}\mathbf{h}.$$