

Algorithms and Programming IV

OpenMP

Summer Term 2023 | 22.05.2023
Barry Linnert

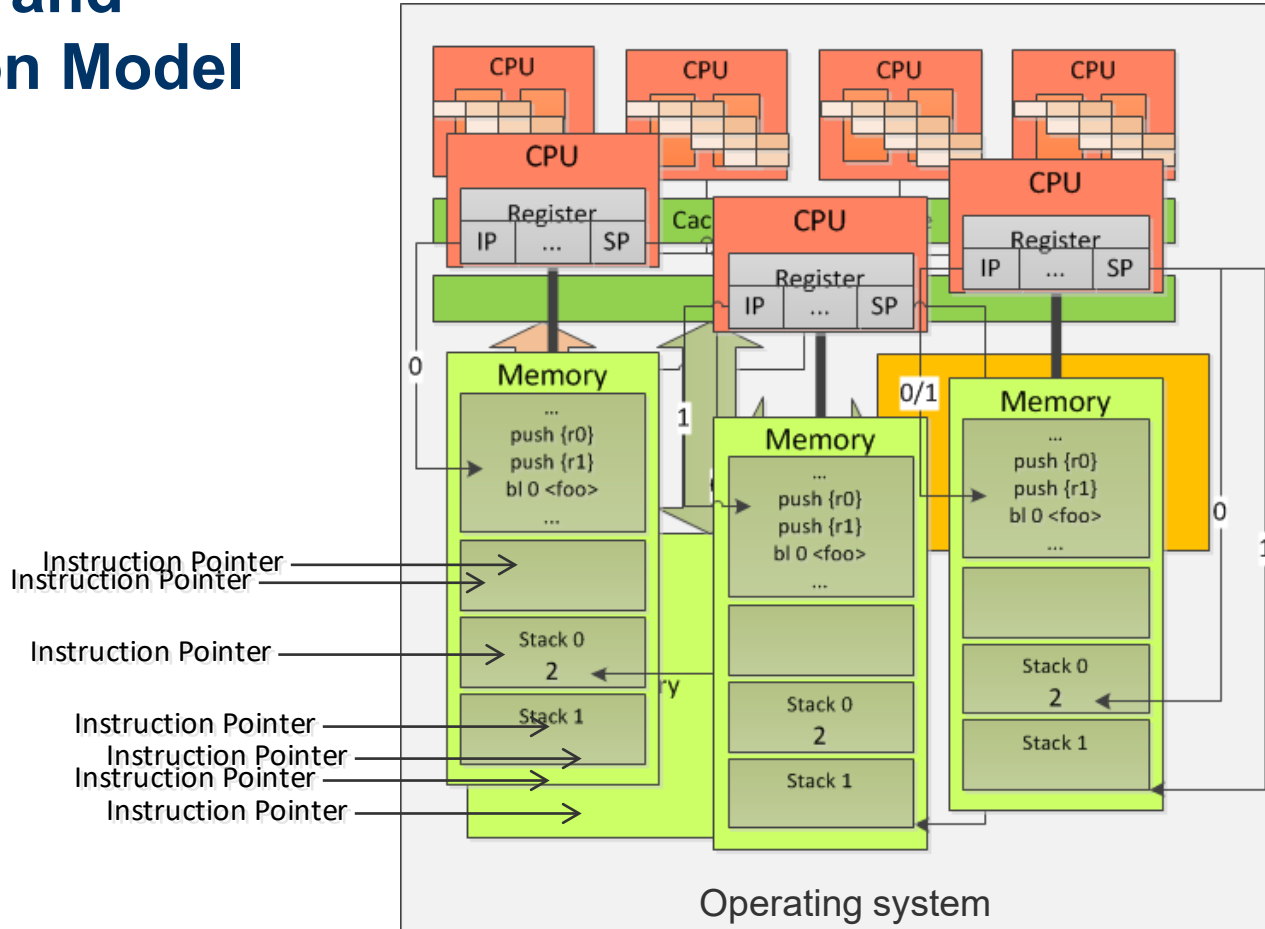
Objectives of Today's Lecture

- Introduction to OpenMP



Concepts of Non-sequential and Distributed Programming

OPENMP

Machine and Execution Model



Requirements for programs

- The program should do what it has been programmed to do!
 - Functional properties
 - Scope of functions
 - Correctness
- The program should follow certain rules for this purpose!
 - Non-functional properties
 - Performance
 - Security
 - ...

OpenMP

- OpenMP was designed to support parallel programming.
- Wide range of parallel algorithms and parallel programs implements processing of parts of data out of a common problem space.
- Thus, the same instruction are used on different parts of the problem represented by shared data.
- Modern hardware architectures provide more than one execution unit that can execute a thread by its own.
- OpenMP represents an application programming interface (API), compiler extensions and operating system support to support multi-threaded shared-memory programming on different hardware architectures.

OpenMP II

- The application programming interface implements support for parallel programming, such as single program multiple data (SPMD) with one operation is performed for different parts of the data.
- Thus, the effort for programming in parallel is reduced in comparison to implementing a multi-threaded application using the operating system interface or thread libraries.
- The API provides a higher abstraction level than POSIX threads (pthreads).

OpenMP III

- OpenMP is architecture and operating system independent.
- The API provides support for programming languages C/C++ and Fortran.
- The compiler has to support the API and maps the operations onto the specific thread libraries of the operating system or the execution environment.
- In case the compiler does not support the OpenMP standard the OpenMP directives (operations) are ignored and the program can be compiled without usage of the OpenMP API.

- www.openmp.org

Example: Matrix Multiplication

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \\ B_{31} & B_{32} \end{pmatrix}$$

$$= \begin{pmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} + A_{13} \cdot B_{31} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} + A_{13} \cdot B_{32} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} + A_{23} \cdot B_{31} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} + A_{23} \cdot B_{32} \end{pmatrix}$$

Example: Matrix Multiplication

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \\ B_{31} & B_{32} \end{pmatrix}$$

$$= \begin{pmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} + A_{13} \cdot B_{31} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} + A_{13} \cdot B_{32} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} + A_{23} \cdot B_{31} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} + A_{23} \cdot B_{32} \end{pmatrix}$$

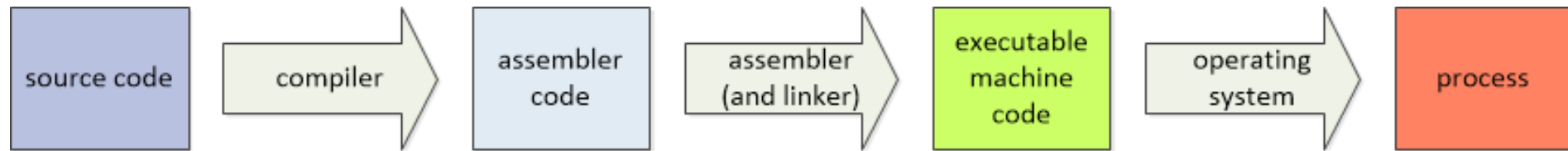
Example: Matrix Multiplication

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \\ B_{31} & B_{32} \end{pmatrix}$$

$$= \begin{pmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} + A_{13} \cdot B_{31} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} + A_{13} \cdot B_{32} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} + A_{23} \cdot B_{31} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} + A_{23} \cdot B_{32} \end{pmatrix}$$

OpenMP – Compiling

- From program to process with threads:



- Activating OpenMP while using the Gnu C-compiler:
 - `gcc -o test -fopenmp test.c`
- Setting the amount of threads to be generated by using environment variable:
 - `export OMP_NUM_THREADS=4`

OpenMP – API

- Using the Fortran programming language:

```
!$omp . . .
```

- Using C/C++:

```
#pragma omp . . .
```

- Applies to the following statement (operation) or a block of statements ({...} in curly brace) only.
- Generates team of threads including the main thread.

Parallel Processing

```
#pragma omp parallel [clause [[,]clause] ...]
```

- Executes the following statement or statement block as team of separated threads.
- The number of threads is determined by the environment variable `OMP_NUM_THREADS`.
- Parameters (clauses) determine the behavior of the team of threads or the usage of data. Clauses can be passed in any order.
- The main thread generates all other threads (team) and executes the instructions (parallel region) as well.
- The (additionally) created threads of the team end with the completion of the last statement. The main thread continues with the following statement.

Parallel Processing – Example

```
// simple program with openmp

#include <stdio.h>

int main ()
{
    #pragma omp parallel
    printf ("I'm a thread. \n");

    return 0;
}
```

How many threads will be created?
How often the sentence “I’m a thread.” will be printed?

Parallel Processing – Variables in Parallel Region

- The usage of the variables within a parallel executed statement block is controlled by the clauses (parameters):

```
private (list_of_variables)
```

- Indicates these are private variables whose values are only available to the corresponding thread.

```
shared (list_of_variables)
```

- Deliver a list of variables holding data shared between all of the threads (shared variables).

Parallel Processing – Variables in Parallel Region II

`default (shared)`

- Indicates that all variables, except those specified separately, are treated as shared variables.

`default (none)`

- For all variables the usage must be explicitly specified.

- **Example:**

```
#pragma omp parallel shared (a, b) private (i)
```

Control of the Threads of a Team

- Different functions can be used to control the team of threads working in parallel:

```
#include <omp.h>
```

```
int omp_get_num_threads ();
```

- Returns the amount of threads executing the current parallel region.

```
int omp_get_thread_num ();
```

- Returns the number of the current thread of the team of threads.

```
// simple program with openmp
// Rauber, Ruenger: Parallele und vert. Prg.
```

```
#include <stdio.h>
```

```
int npoints, iam, np, mypoints;
```

```
double *x;
```

```
int main () {
```

```
    scanf ("%d", &npoints);
```

```
    x = (double *) malloc (npoints * sizeof (double));
```

```
    initialize ();
```

```
    #pragma omp parallel shared (x, npoints) private (iam, np, mypoints)
```

```
    {
```

```
        mypoints = npoints / np;
```

```
        compute_subdomain (x, iam, mypoints);
```

```
    }
```

```
    return 0;
```

```
}
```

source code: 09-01.c

```
// simple program with openmp
// Rauber, Ruenger: Parallele und vert. Prg.
```

```
#include <stdio.h>
#include <omp.h>

int npoints, iam, np, mypoints;
double *x;

int main () {
    scanf ("%d", &npoints);
    x = (double *) malloc (npoints * sizeof (double));
    initialize ();

    #pragma omp parallel shared (x, npoints) private (iam, np, mypoints)
    {
        np = omp_get_num_threads ();
        iam = omp_get_thread_num ();
        mypoints = npoints / np;
        compute_subdomain (x, iam, mypoints);
    }
    return 0;
}
```

source code: 09-01.c

Nested Parallel Regions

- Parallel regions can be nested.
- The inner parallel region is then executed (by default) by a team containing one thread only.
- If the inner region is also to be executed in parallel, this can be changed:

```
#include <omp.h>
```

```
int omp_set_nested (int nested);
```

- Enables or disables nested parallelism. To enable nested parallel execution the value `nested != 0`

- The amount of threads of the team used for the inner parallel region depends on the specific implementation of the OpenMP environment.

Parallel Loop

```
#pragma omp for [clause [[,] clause] ...]  
for (i = lower_bound; i op upper_bound; incr_expr)  
{  
    ...  
}
```

- For processing one or more loops in parallel.
- Loop has to be embedded in a parallel region.
- The loop must be of fixed length.
- `incr_expr` must be a loop independent integer expression and the loop must not be terminated prematurely (`break`).
- The loop count variable is always a private variable.

Parallel Loop – Shortcut

```
#pragma omp parallel for [clause [[,] clause] ...]  
for (i = lower_bound; i op upper_bound; incr_expr)  
{  
    ...  
}
```

- Is a short notation of the parallel loop OpenMP directive. It can be used if only a single loop is to be executed in parallel.
- The loop statement (operation) has to follow the OpenMP directive.

Distribution of Iterations of the Loop

- Usually, the distribution of the loop iterations (chunk) among the threads of the team is done automatically but can be controlled by clauses (parameters):

```
schedule (static, chunk_size)
```

- A static partitioning is performed so that the iterations are divided into chunks of the size `chunk_size` and assigned to the threads in turn (round-robin).

```
schedule (dynamic, chunk_size)
```

- Threads whose processing is complete are assigned the next chunk (bag-of-tasks).

```
schedule (guided, chunk_size)
```

- The size of the newly assigned chunk is reduced with every new assignment.

Distribution of Iterations of the Loop II

- The settings for the distribution of the loop iterations can also be controlled by environment variables:

```
schedule (runtime)
```

- The schedule and chunk size are taken from the environment variables:

```
export OMP_SCHEDULE "dynamic, 4"
```

```
export OMP_SCHEDULE "guided"
```

Example: Matrix Multiplication

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \\ B_{31} & B_{32} \end{pmatrix}$$

$$= \begin{pmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} + A_{13} \cdot B_{31} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} + A_{13} \cdot B_{32} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} + A_{23} \cdot B_{31} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} + A_{23} \cdot B_{32} \end{pmatrix}$$

```
// simple program with openmp
// Rauber, Ruenger: Parallele und vert. Prg.
```

```
#include <stdio.h>
#include <omp.h>

double MA[100][100], MB[100][100];
double MC[100][100];
int i, row, col, size = 100;

void read_input ()
{
    int j;

    for (i = 0; i < 100; i++)
        for (j = 0; j < 100; j++)
            MA[i][j] = (double)(i + j) + 1.0;

    for (i = 0; i < 100; i++)
        for (j = 0; j < 100; j++)
            MB[i][j] = (double)(i + j) + 1.0;
}
```

```
void write_output ()
{
    int j;

    for (i = 0; i < 100; i++)
        for (j = 0; j < 100; j++)
            printf ("%f ", MC[i][j]);
        printf ("\n");
}

int main () {
    read_input (); // MA, MB
    ...

    write_output (); // MC

    return 0;
}
```

```
// simple program with openmp
// Rauber, Ruenger: Parallele und vert. Prg.
```

```
#include <stdio.h>
#include <omp.h>

double MA[100][100], MB[100][100];
double MC[100][100];
int i, row, col, size = 100;
```

```
void read_input ()
{
    ...
}
```

```
void write_output ()
{
    ...
}
```

```
int main () {
    read_input (); // MA, MB
    #pragma omp parallel
        shared (MA, MB, MC, size)
        private (row, col, i)
    {
        for (row = 0; row < size; row++) {
            for (col = 0; col < size; col++)
                MC[row][col] = 0.0;
        }

        for (row = 0; row < size; row++) {
            for (col = 0; col < size; col++)
                for (i = 0; i < size; i++)
                    MC[row][col] += MA[row][i] *
                        MB[i][col];
        }
    }
    write_output (); // MC
    return 0;
}
```

source code: 09-02.c

```
// simple program with openmp
// Rauber, Ruenger: Parallele und vert. Prg.
```

```
#include <stdio.h>
#include <omp.h>

double MA[100][100], MB[100][100];
double MC[100][100];
int i, row, col, size = 100;

void read_input ()
{
    ...
}

void write_output ()
{
    ...
}
```

```
int main () {
    read_input (); // MA, MB
    #pragma omp parallel
        shared (MA, MB, MC, size)
        private (row, col, i)
    {
        #pragma omp for schedule (static)
        for (row = 0; row < size; row++) {
            for (col = 0; col < size; col++)
                MC[row][col] = 0.0;
        }
        #pragma omp for schedule (static)
        for (row = 0; row < size; row++) {
            for (col = 0; col < size; col++)
                for (i = 0; i < size; i++)
                    MC[row][col] += MA[row][i] *
                        MB[i][col];
        }
    }
    write_output (); // MC
    return 0;
}
```

source code: 09-02.c

```
// simple program with openmp
// Rauber, Ruenger: Parallele und vert. Prg.
```

```
#include <stdio.h>
#include <omp.h>

double MA[100][100], MB[100][100];
double MC[100][100];
int i, row, col, size = 100;

void read_input ()
{
    ...
}

void write_output ()
{
    ...
}
```

```
int main () {
    read_input (); // MA, MB
    #pragma omp parallel
        private (row, col, i)
    {
        #pragma omp for schedule (static)
        for (row = 0; row < size; row++) {
            #pragma omp parallel
                shared (MA, MB, MC, size)
            {
                #pragma omp for schedule (static)
                for (col = 0; col < size; col++) {
                    MC[row][col] = 0.0;
                    for (i = 0; i < size; i++)
                        MC[row][col] += MA[row][i] *
                            MB[i][col];
                }
            }
        }
    }
    write_output (); // MC
    return 0; }
```

Functional Parallelism

```
#pragma omp sections [clause [[,]clause] ...]
```

- Executes the parts of the following statement block by separated threads as team of threads. The OpenMP directive does not apply to iterative parallel regions (of loops).
- Each of the sub-regions is delimited as a section:

```
#pragma omp section
```

- An implicit synchronization of the thread with the main thread takes place at the end of a `section`. The synchronization can be prevented with the clause (parameter) `nowait`.

```
// simple program with openmp
```

```
// Quinn: parallel prog. in c with mpi and openmp
```

```
#include <stdio.h>
```

```
#include <omp.h>
```

```
double alpha () {  
    printf ("alpha \n");  
    return 1.1;  
}
```

```
double beta () {  
    printf ("beta \n");  
    return 2.2;  
}
```

```
double delta () {  
    printf ("delta \n");  
    return 3.3;  
}
```

```
double my_gamma (double v, double w) {  
    printf ("gamma \n");  
    return v + w;  
}
```

```
double epsilon (double x, double y) {  
    printf ("epsilon \n");  
    return x + y;  
}
```

```
int main () {  
    double v, w, x, y;
```

```
    v = alpha ();
```

```
    w = beta ();
```

```
    x = my_gamma (v, w);
```

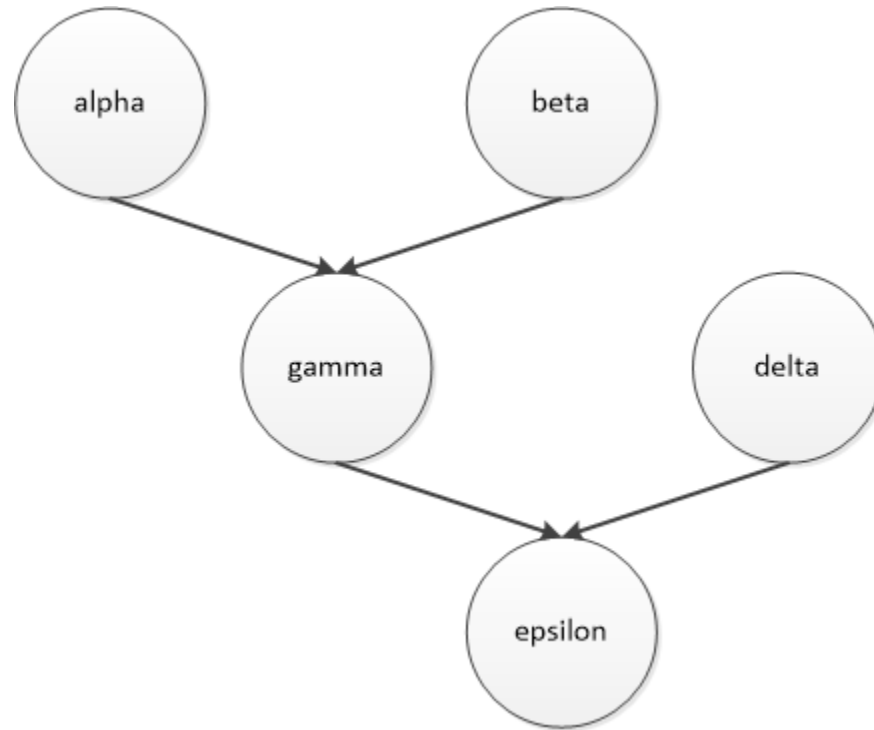
```
    y = delta ();
```

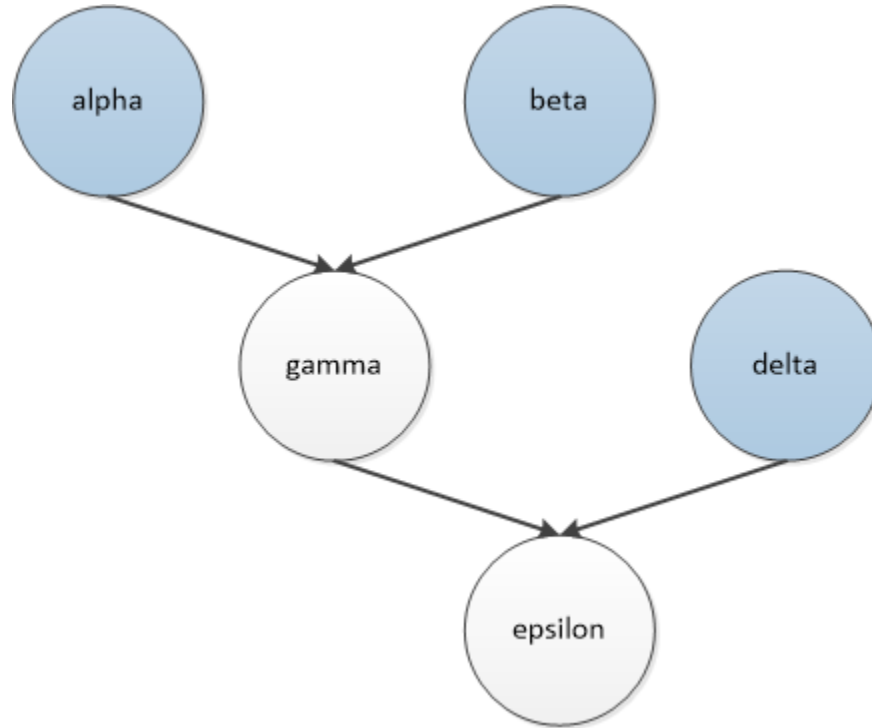
```
    printf ("%6.2f \n", epsilon (x, y));
```

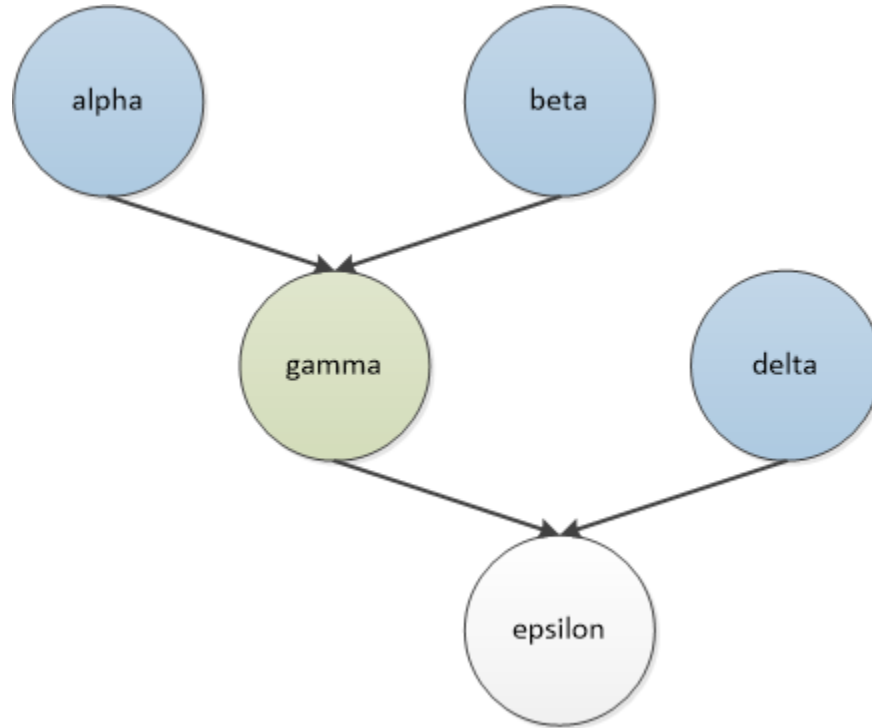
```
    return 0;
```

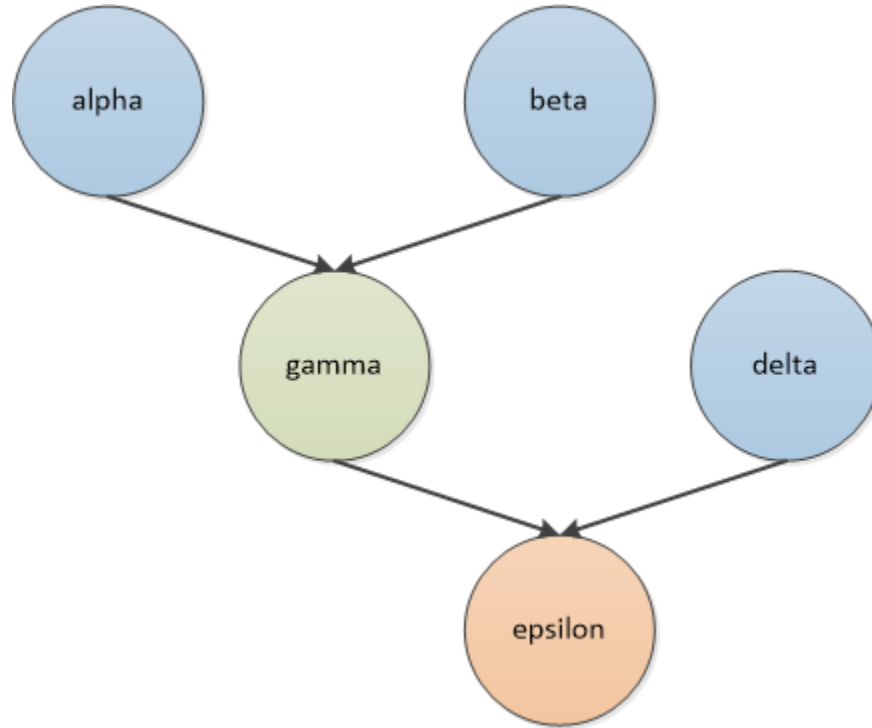
```
source code: 09-04.c
```











```
// simple program with openmp
```

```
// Quinn: parallel prog. in c with mpi and openmp
```

```
#include <stdio.h>
```

```
#include <omp.h>
```

```
double alpha () {  
    printf ("alpha \n");  
    return 1.1;  
}
```

```
double beta () {  
    printf ("beta \n");  
    return 2.2;  
}
```

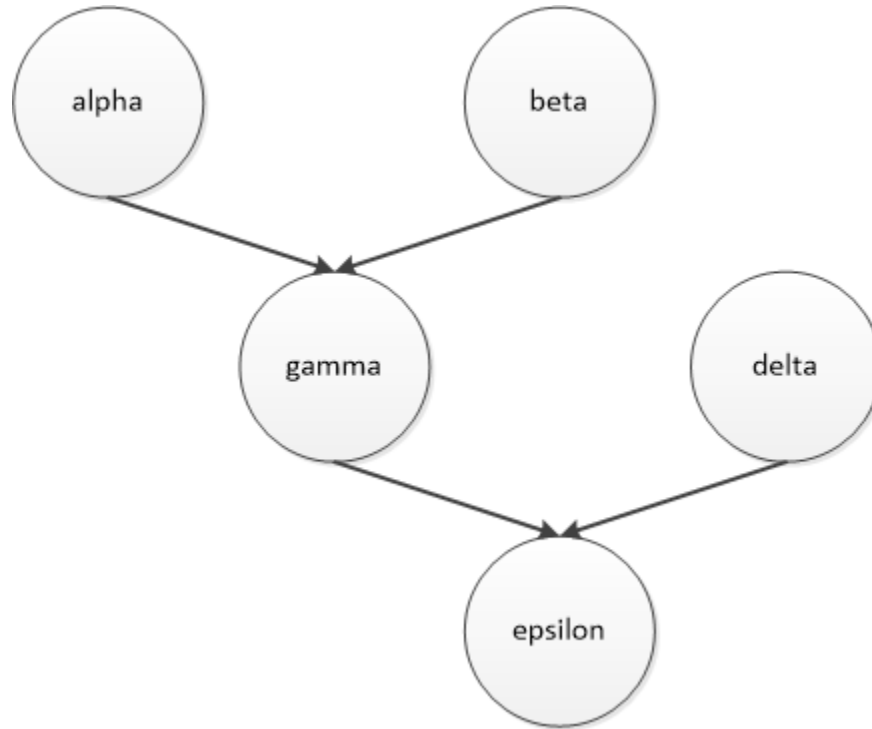
```
double delta () {  
    printf ("delta \n");  
    return 3.3;  
}
```

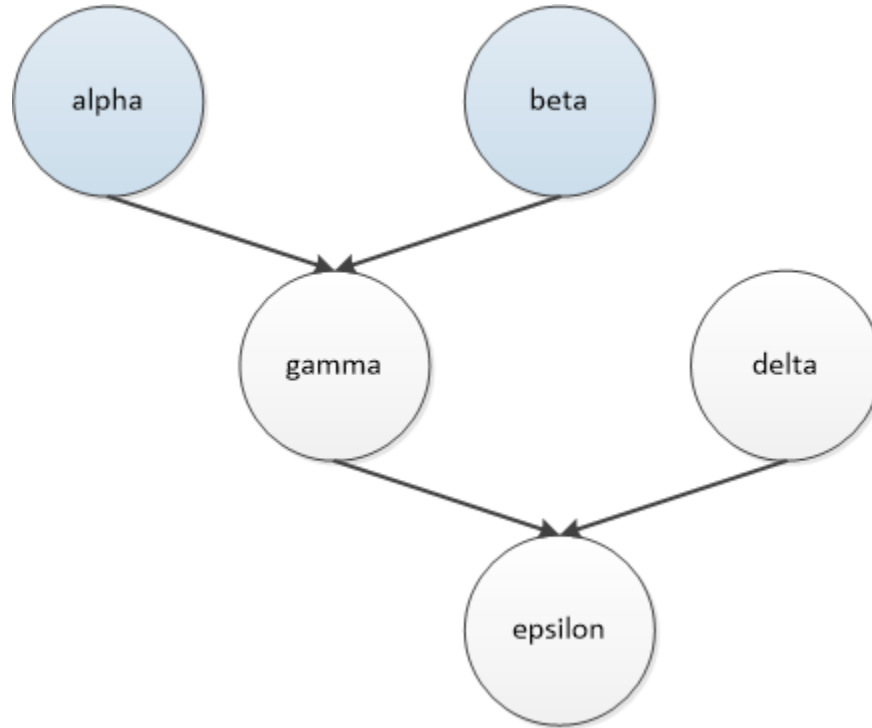
```
double my_gamma (double v, double w) {  
    printf ("gamma \n");  
    return v + w;  
}
```

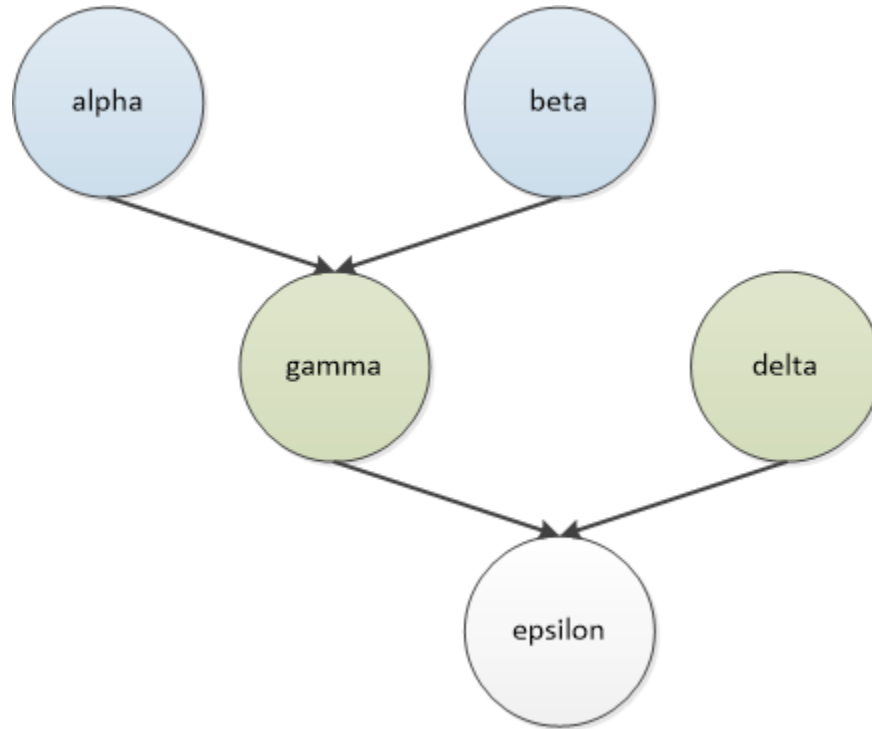


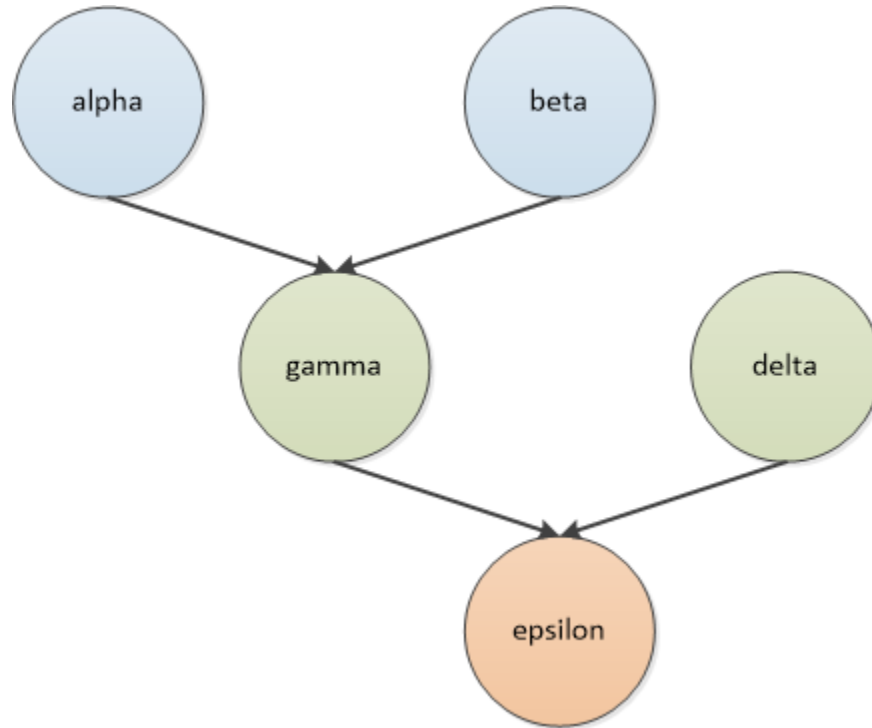
```
double epsilon (double x, double y) {  
    printf ("epsilon \n");  
    return x + y;  
}
```

```
int main () {  
    double v, w, x, y;  
  
    #pragma omp parallel sections  
    {  
        #pragma omp section  
        v = alpha ();  
        #pragma omp section  
        w = beta ();  
        #pragma omp section  
        y = delta ();  
    }  
  
    x = my_gamma (v, w);  
    printf ("%6.2f \n", epsilon (x, y));  
  
    return 0;
```









```
// simple program with openmp
// Quinn: p. prg. in c w. mpi + openmp
```

```
#include <stdio.h>
#include <omp.h>
```

```
double alpha () {
    return 1.1;
}
double beta () {
    return 2.2;
}
double delta () {
    return 3.3;
}
double my_gamma (double v, double w) {
    return v + w;
}

double epsilon (double x, double y) {
    return x + y;
}
```

```
int main () {
    double v, w, x, y;
    #pragma omp parallel
    {
        #pragma omp sections
        {
            #pragma omp section
            v = alpha ();
            #pragma omp section
            w = beta ();
        }
        #pragma omp sections
        {
            #pragma omp section
            x = my_gamma (v, w);
            #pragma omp section
            y = delta ();
        }
    }
    printf ("%6.2f \n", epsilon (x, y));

    return 0;
}
```

source code: 09-05.c



Reduction

- After completion of a parallel statement block defined by `parallel`, `sections` and `for` directives, the intermediate results can be summarized with the clause:

```
reduction (op: list)
```

- The clause `op` denotes a specific reduction operator to be applied (+, -, *, /, &, ^, |, &&, ||) and `list` denotes the variables to be used for reduction.
- The intermediate results are processed by the threads in separate, private variables and are only merged when the parallel region is completed.

```
// simple program with openmp to calc pi
// Quinn: parallel prog. in c with mpi and openmp
```

```
#include <stdio.h>
#include <omp.h>
```

```
int main () {
    double area, pi, x;
    int i, n;

    n = 1000000000;
    area = 0.0;

    #pragma omp parallel for private (x) reduction (+:area)
    for (i = 0; i < n; i++) {
        x = ((double) i + 0.5) / n;
        area += 4.0 / (1.0 + x*x);
    }
    pi = area / n;

    printf ("Pi: %f \n", pi);
    return 0;
}
```

source code: 09-06.c

Single Thread Execution

```
#pragma omp single [clause [[,]clause] ...]
```

- The following (sub)region is processed by one thread of the team only.
- This is particularly useful to process a (cumulated) output.
- At the end of the (partial) region an implicit synchronization takes place. The synchronization can be prevented with the parameter `nowait`.

Execution by the Main Thread

```
#pragma omp master
```

- The following (sub)region is processed by the main thread of the team only. All other threads ignore the following statement or statements. They do not even need to reach the previous statements.
- This is particularly useful for initialization, work distribution and processing of results.
- There is no implicit synchronization.

Thread Control

- Further functions to control the number of threads are:

```
#include <omp.h>
```

```
int omp_set_dynamic (int dynamic_threads);
```

- With submitting `dynamic_threads != 0` the runtime system is allowed to determine the number of threads of the team dynamically.
- The function must be called outside of a parallel region, and the number of threads for each parallel region remains constant.

```
int omp_get_dynamic ();
```

- Returns the current setting of the dynamic assignment of the number of threads.

Thread Control II

- As there are functions to get the current number of threads of a team the amount of threads can be set by:

```
#include <omp.h>
```

```
int omp_set_num_threads (int num_threads);
```

- Defines the number of threads for the subsequent parallel region.
- The function must be called outside of a parallel region.
- The effect depends on a dynamic or static adjustment of the number of threads.

Critical Section

```
#pragma omp critical [(name)]
```

- Protects a critical section generated by parallel processing threads.
- Emphasize the following parallel (sub)region as a critical section and protect it by mutual exclusion. All threads wait as long as the critical section is processed by another thread.
- A name `name` can be assigned to the specific critical section. If the critical sections are named individually, a mutual exclusion will only occur for the specific named critical section.

Barrier

```
#pragma omp barrier
```

- All threads will synchronize at the barrier and wait until all other threads have reached this point of processing.

Lock

- In order to build individual solutions to protect a critical section different functions to use a lock are provided:

```
#include <omp.h>
```

```
int omp_set_lock (omp_lock_t *lock);
```

- Tries to set a lock and block the thread in case the lock is set by another thread.

```
int omp_unset_lock (omp_lock_t *lock);
```

- Release (unset) the lock.

Lock II

- Critical section may be nested too. So there is another set of functions dealing with nested critical sections:

```
int omp_set_nest_lock (omp_nest_lock_t *lock);
```

- If the lock is free, set it for the current thread. If the lock was set by another thread, block the current requesting thread. If the lock was set by the same thread increments the lock, so the thread is able to access the critical section (again).
- All other threads attempting to set a set lock variable will be blocked until the lock is completely reset.

```
int omp_unset_nest_lock (omp_nest_lock_t *lock);
```

- Decrement the lock and if the value is 0 deblock all blocked threads.

Lock III

```
int omp_init_lock (omp_lock_t *lock);
```

```
int omp_init_nest_lock (omp_nest_lock_t *lock);
```

– Initialization of the lock.

```
int omp_destroy_lock (omp_lock_t *lock);
```

```
int omp_destroy_nest_lock (omp_nest_lock_t *lock);
```

– Remove the lock.

Lock IV

```
int omp_test_lock (omp_lock_t *lock);
```

```
int omp_test_nest_lock (omp_nest_lock_t *lock);
```

- Tries to acquire the lock, but does not block the thread in case the attempt was not successful. The return value indicates the success or failure.

Consistent Content of the Shared Data

```
#pragma omp atomic
```

- Ensures that the following statement accessing the memory is executed atomically.
- Possible statements are: $x \text{ binop} = E$, $x++$, $++x$, $x--$, $--x$.

With $\text{binop} = \{+, -, *, /, \&, \wedge, |, \ll, \gg\}$.

```
#pragma omp flush [(list)]
```

- All threads get a consistent view on the content of the variables of the set `list`.
- If the variables are pointers, only the consistency of the pointers - not of the data to which the pointer points - is ensured.

Concepts of Non-sequential and Distributed Programming

NEXT LECTURE

Parallel Programming with Message Passing

APL IV: Concepts of Non-sequential and Distributed
Programming (Summer Term 2023)