



Multicore & GPU Programming : An Integrated Approach

# Shared-Memory Programming : OpenMP

By G. Barlas

# Objectives

- Learn how to use OpenMP compiler directives to introduce concurrency in a sequential program.
- Learn the most important OpenMP `#pragma` directives and associated clauses, for controlling the concurrent constructs generated by the compiler.
- Understand which loops can be parallelized with OpenMP directives.
- Address the dependency issues that OpenMP-generated threads face, using synchronization constructs.
- Learn how to use OpenMP to create function-parallel programs.
- Learn how to write thread-safe functions.
- Understand the issue of cache false-sharing and learn how to eliminate it.
- Learn how to use OpenMP to offload work to GPUs and/or take advantage of SIMD CPU extensions.

# Introduction

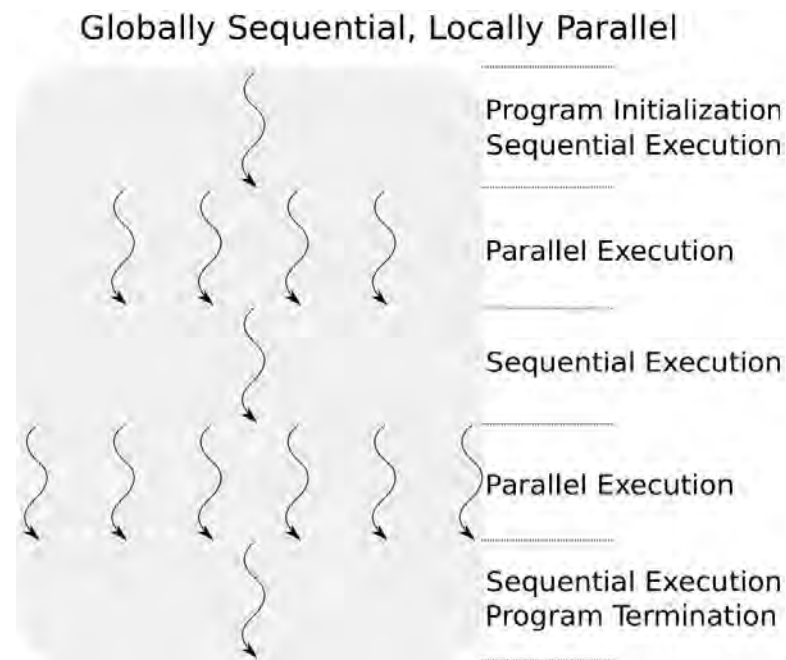
- The decomposition of a sequential program into components that can execute in parallel is a tedious enterprise.
- OpenMP has been designed to alleviate much of the effort involved, by accommodating the incremental conversion of sequential programs into parallel ones, with the assistance of the compiler.
- OpenMP relies on compiler directives for decorating portions of the code that the compiler will attempt to parallelize.

# OpenMP history

- OpenMP : Open Multi-Processing is an API for shared-memory programming.
- OpenMP was specifically designed for parallelizing existing sequential programs.
- Uses compiler directives and a library of functions to support its operation.
- OpenMP v.1 was published in 1998.
- OpenMP v.5.2 was published in November 2021. Now OpenMP supports GPU accelerators.
- Standard controlled by the OpenMP Architecture Review Board (ARB).
- GNU Compiler support:
  - GCC 9 supports OpenMP 5.0
  - GCC 12 supports OpenMP 5.1

# OpenMP paradigm

- OpenMP programs are Globally Sequential, Locally Parallel.
- Programs follow the fork-join paradigm:



# OpenMP essential definitions

- **Structured block** : an executable statement or a compound block, with a single point of entry and a single point of exit.
- **Construct** : an OpenMP directive and the associated statement, for-loop or structured block that it controls.
- **Region** : all code encountered during the execution of a *construct*, including any called functions.
- **Parallel region** : a region executed simultaneously by multiple threads.
- A **region** is dynamic but a **construct** is static.
- **Master thread** : the thread executing the sequential part of the program and spawning the **child threads**.
- **Thread team** : a set of threads that execute a parallel region.

# „Hello World“ in OpenMP

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

using namespace std;

int main (int argc, char **argv)
{
    int numThr = atoi (argv[1]);

#pragma omp parallel num_threads(numThr)
    cout << "Hello from thread " << omp_get_thread_num () << endl;

    return 0;
}
```

- Can you match some of the previous definitions with parts of this program?

# „Hello World“ sequence diagram

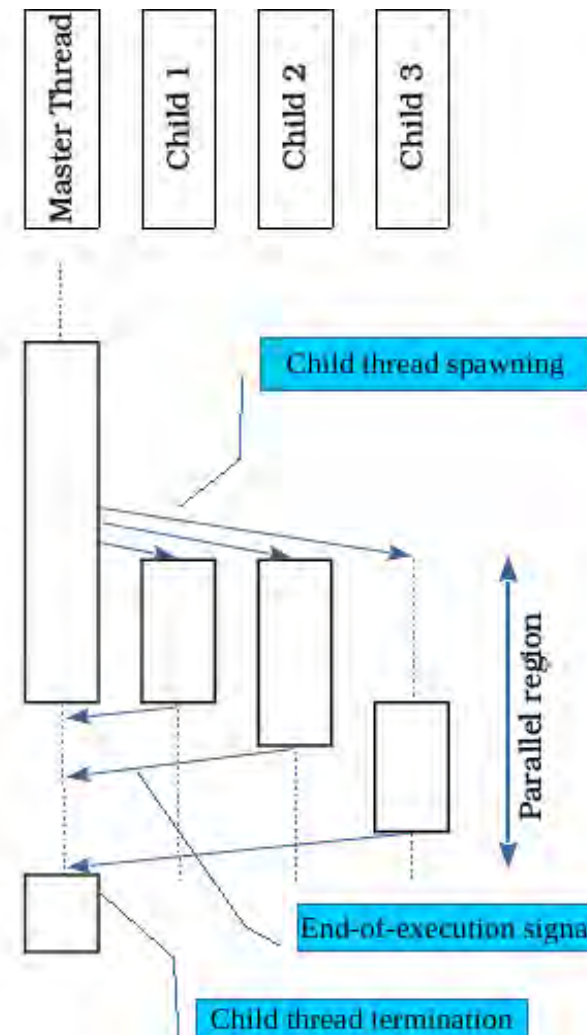
- One of the possible execution sequences:

```
int main (int argc, char **argv)
{
    int numThr = atoi (argv[1]);

    #pragma omp parallel num_threads(numThr)

    cout << "Hello from thread " <<
        Omp_get_thread_num () << endl;

    return 0;
}
```





# #pragma directives

- Pragma directives allow a programmer to access compiler-specific preprocessor extensions.
- For example, a common use of pragmas, is in the management of include files. E.g.

```
#pragma once
```

- Pragma directives in OpenMP can have a number of optional **clauses**, that modify their behavior.
- In the previous example the clause is `num_threads(numThr)`
- Compilers that do not support certain pragma directives, ignore them.

# Thread team size control

- **Universally:** via the `OMP_NUM_THREADS` environment variable:

```
$ echo ${OMP_NUM_THREADS} # to query the value
```

```
$ export OMP_NUM_THREADS=4 # to set it in BASH
```

- **Program level :** via the `omp_set_num_threads` function, outside an OpenMP construct.
- **Pragma level :** via the `num_threads` clause.
- The `omp_get_num_threads` call returns the active threads in a parallel region. If it is called in a sequential part it returns 1.

# Variable scope

- Outside the parallel regions, normal scope rules apply.
- OpenMP specifies the following types of variables:
  - **Shared** : all variables declared outside a parallel region are **by default** shared. That does not mean that they are in anyway "protected" from race conditions.
  - **Private** : all variables declared inside a parallel region are allocated in the run-time stack of each thread. So we have as many copies of these variables as the size of the thread team. Private variables are destroyed upon the termination of a parallel region.
  - **Reduction** : a reduction variable gets individual copies for each thread running the corresponding parallel region. Upon the termination of the parallel region, an operation is applied to the individual copies (e.g. summation) to produce the value that will be stored in the shared variable.
- The default scope of variables can be modified by clauses in the pragma lines.

# Example : function integration

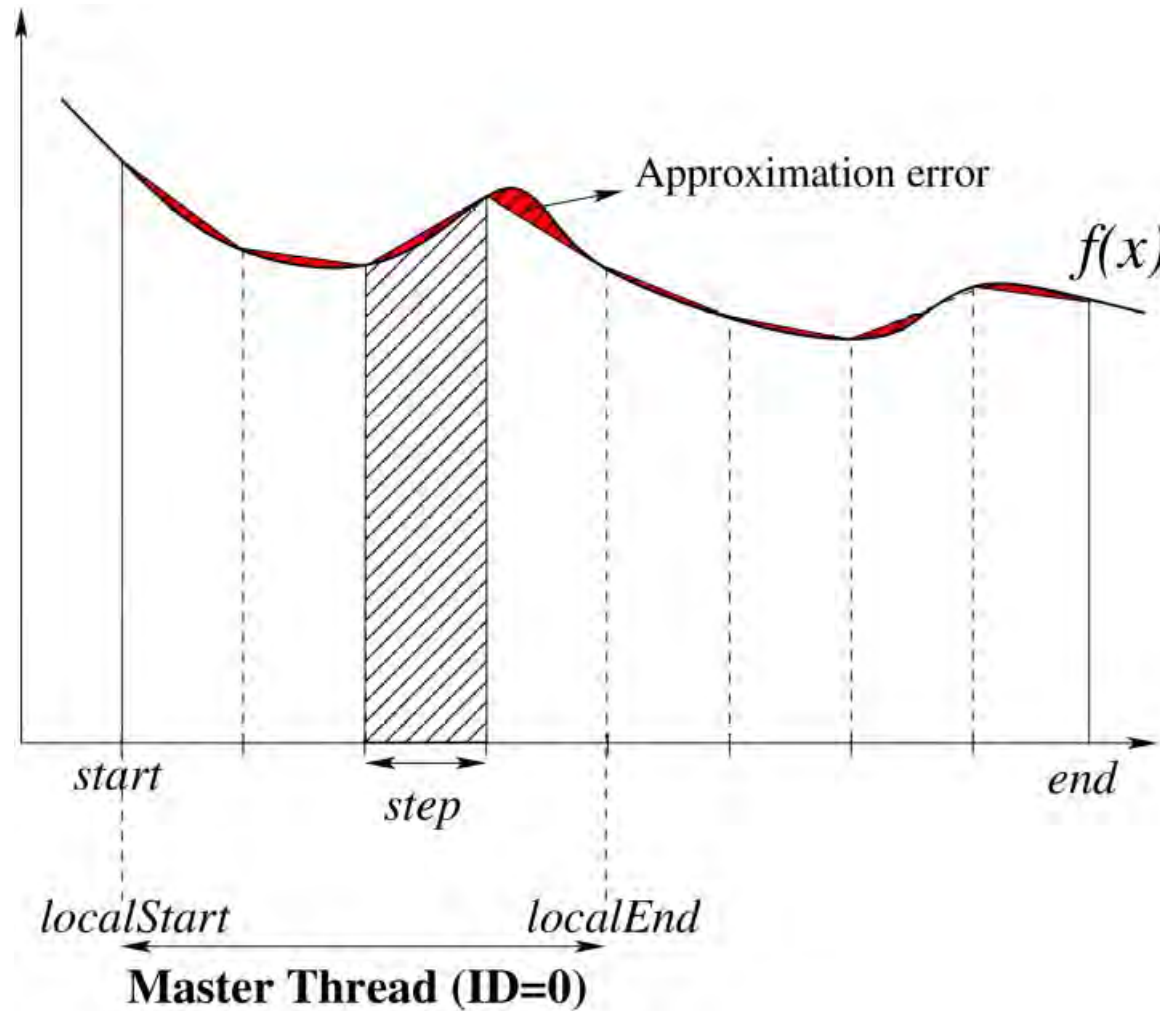
- The sequential implementation:

```
double integrate (double st, double en, int div, double (*f) (double))
{
    double localRes = 0;
    double step = (en - st) / div;
    double x;
    x = st;
    localRes = f (st) + f (en);
    localRes /= 2;
    for (int i = 1; i < div; i++)
    {
        x += step;
        localRes += f (x);
    }
    localRes *= step;

    return localRes;
}
//-----
int main (int argc, char *argv[])
{
    . . .
    double finalRes = integrate (start, end, divisions, testf);

    cout << finalRes << endl;
```

# Parallel function integration



# OpenMP V.0 : manual partitioning

- Given the ID of each thread, we can calculate:

$$localStart = start + ID \cdot localDiv \cdot step$$

$$localEnd = localStart + localDiv \cdot step$$

```
// get the number of threads for next parallel region
int N = omp_get_max_threads ();
divisions = (divisions / N) * N;    // make sure divisions is a ←
    multiple of N
double step = (end - start) / divisions;

double finalRes = 0;
#pragma omp parallel
{
    int localDiv = divisions / N;
    int ID = omp_get_thread_num ();
    double localStart = start + ID * localDiv * step;
    double localEnd = localStart + localDiv * step;
    finalRes += integrate (localStart, localEnd, localDiv, testf);
}

cout << finalRes << endl;
```

Data race!



# OpenMP V.1 : removing the data race

- Giving each thread, its own private storage. Sequential reduction is required afterwards.

```
// allocate memory for the partial results
double *partial = new double[N];
#pragma omp parallel
{
    int localDiv = divisions / N;
    int ID = omp_get_thread_num ();
    double localStart = start + ID * localDiv * step;
    double localEnd = localStart + localDiv * step;
    partial[ID] = integrate (localStart, localEnd, localDiv, testf);
}

// reduction step
double finalRes = partial[0];
for (int i = 1; i < N; i++)
    finalRes += partial[i];
```

# OpenMP V.2 : implicit partitioning with locking

- Moving the parallel construct inside the `integrate()` function. The `main` remains the same as the sequential program.

```
double integrate (double st, double en, int div, double (*f) (double))
{
    double localRes = 0;
    double step = (en - st) / div;
    double x;
    x = st;
    localRes = f (st) + f (en);
    localRes /= 2;

    #pragma omp parallel for private(x)
    for (int i = 1; i < div; i++)
    {
        x = st + i * step;
        double temp = f (x);
        #pragma omp critical
        localRes += temp;
    }

    localRes *= step;

    return localRes;
}
```

Can we eliminate x from here?

This statement is also different from the sequential version.



# OpenMP V.3 : implicit partitioning with reduction

- Most efficient way to consolidate results.

```
double integrate (double st, double en, int div, double (*f) (double))
{
    double localRes = 0;
    double step = (en - st) / div;
    double x;
    x = st;
    localRes = f (st) + f (en);
    localRes /= 2;

#pragma omp parallel for private(x) reduction(+: localRes)
    for (int i = 1; i < div; i++)
    {
        x = st + i * step;
        localRes += f (x);
    }

    localRes *= step;

    return localRes;
}
```

# Reduction clause

- The reduction clause syntax:

`reduction( reduction_id : variable_list)`

where `variable_list` is a comma-separated list of variable identifiers, and `reduction_id` is one of the following binary arithmetic and boolean operators :

`+, *, -, & , &&, |, || , ^, max, min`

- Example:

```
int maxElem = data[0];
#pragma omp parallel for reduction(max : maxElem)
for (int i = 1; i < sizeof (data) / sizeof (int); i++)
    if (maxElem < data[i])
        maxElem = data[i];
```

# Reduction clause (2)

- The initial values of a reduction variable's private copies depend on the operator used:

| Operator                     | Private Copy Initial Value   |
|------------------------------|--|
| $+$ , $-$ , $ $ , $  $ , $^$ | 0  |
| $*$ , $\&\&$                 | 1  |
| $\&$                         | 0xFFFF...FFF, i.e. all bits set to 1   |
| max                          | Smallest possible number that can be represented by the type of the reduction variable |
| min                          | Largest possible number that can be represented by the type of the reduction variable  |

# Scope modifying clauses

- `shared` : the default behavior for variables declared outside of a parallel block. It needs to be used only if `default(none)` is also specified.
- `reduction` : a reduction operation is performed between the private copies and the „outside“ object. The final value is stored in the „outside“ object.
- `private` : creates a separate copy of a variable for each thread in the team. Private variables are not initialized, so one should not expect to get the value of the variable declared outside the parallel construct.
- `firstprivate` : behaves the same way as the `private` clause, but the private variable copies are initialized to the value of the „outside“ object.
- `lastprivate` : behaves the same way as the `private` clause, but the thread finishing the last iteration of the sequential block (for the final value of the loop control variable that produces an iteration), copies the value of its object to the „outside“ object.
- `threadprivate` : creates thread-specific, *persistent* storage (i.e. for the duration of the program) for global data.

# Data copying clauses

- `copyin` : Used in conjunction with the `threadprivate` clause, to initialize the thread private copies of a team of threads, from the master thread's variables.
- `copyprivate` : Used to broadcast the value of a private variable of a task to the copies of other tasks that belong to the same parallel region. This clause is typically attached to a `single` construct, and the copy is performed at the end of the single's structured block and before the parallel region is complete.
- The difference between `copyin` and `copyprivate` is that the former is used to make the copy at the very beginning of a construct, while the latter can make the copy at any point of a parallel region's execution.
- Example:

```
float x, y;
#pragma omp threadprivate(x, y)

void init() {
#pragma omp single copyprivate(x,y)
{
    scanf("%f %f", &x, &y);
}
}
```

One thread would read `x` and `y` from the user and copy these values to the other threads upon exit from the `single` block



# Loop Level Parallelism

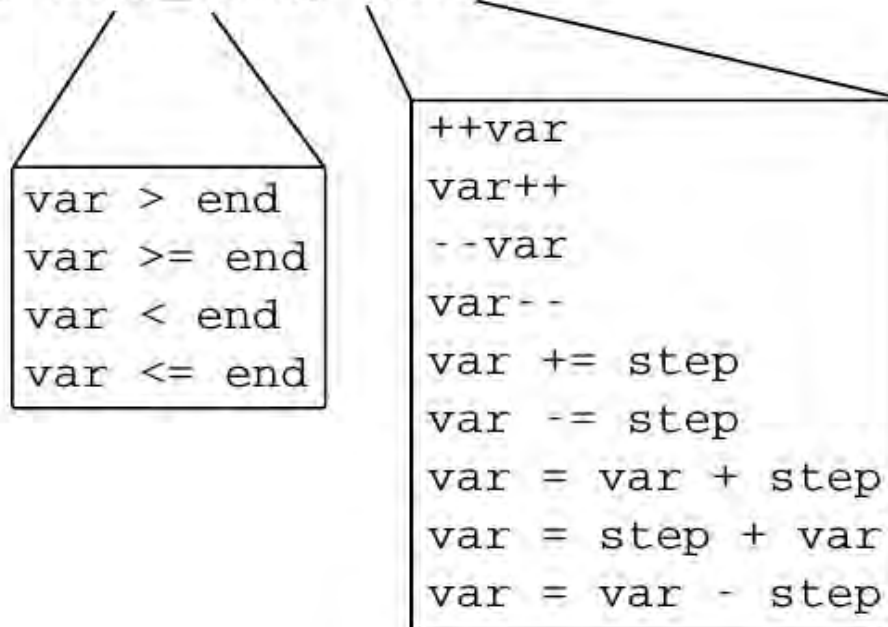
- A for-loop has to satisfy certain conditions for the application of the `pragma omp for` directive, which are called the **canonical form**:
  - The loop control variable has to be an integer type (signed or unsigned), a pointer type (e.g. base address of an array), or a random access iterator (for C++). The loop control variable is made private by default, even if it is declared outside the loop.
  - The loop control variable must not be modified in the body of the loop.
  - The limit against which the loop control variable is compared against, to determine the truth of the termination condition, must be loop invariant.
- Counter-example of a filtering data loop:

```
for (int i = 0; i < M; i++)  
{  
    if (data[i] % 2 == 0)  
    {  
        data[i] = data[M-1]; // copy over  
        M--; // limit modified  
        i--; // loop control var change  
    }  
}
```

# Canonical form

- Loop control variable operations are also limited:

```
for(var = start; term_cond; incr)
```



- break, goto and throw are not allowed to transfer control outside the loop.
- Exiting the program from within the loop is allowed.

# The "parallel for" directive

- The `#pragma omp parallel for` directive is actually a shortcut for:

```
#pragma omp parallel
{
    #pragma omp for
    for(....
}
}
```

- This has implications about what exactly `#pragma omp parallel` actually does.
- The same parallel construct can be populated by other **work sharing constructs**, such as sections and tasks.



# Data dependencies

- Assuming we have a loop of the form:

```
for (i = ...  
{  
    S1 : operate on a memory location x  
    ...  
    S2 : operate on a memory location x  
}
```

- There are four different ways that S1 and S2 are connected, based on whether they are reading or writing to x.
- A problem exists if the dependence crosses loop iterations : **loop-carried dependence**.

# Dependence types

- Flow dependence : RAW  $S1 \delta^f S2$

```
x = 10; // S1  
y = 2 * x + 5; // S2
```

- Anti-flow dependence : WAR  $S1 \delta^a S2$

```
y = x + 3; // S1  
x ++ ; // S2
```

# Dependence types (cont.)

- Output dependence : WAW  $S1 \delta^o S2$

```
x = 10; // S1  
x = x + c; // S2
```

- Input dependence : RAR  $S1 \delta^i S2$

```
y = x + c; // S1  
z = 2 * x + 1; // S2
```

# Flow dependence : reduction, induction variables

- Example:

```
double v = start;  
double sum=0;  
for (int i = 0; i < N; i++)  
{  
    sum = sum + f(v);    // S1  
    v = v + step;        // S2  
}
```

- $S1 \delta^f S1$  caused by reduction variable `sum`.
- $S2 \delta^f S2$  caused by induction variable `v`.
- $S2 \delta^f S1$  caused by induction variable `v`.
- Induction variable : affine function of the loop variable.

# Reduction, induction variables fix

- Reduction variables : use a `reduction` clause.
- Induction variables : use affine function directly.

```
double v = start; // now irrelevant. v can be an automatic variable
                  declared inside the loop
double sum=0;

#pragma omp parallel for reduction(+ : sum) private(v) shared(step)
for(int i = 0; i < N; i++)
{
    v = i * step + start;
    sum = sum + f(v);
}
```

# Flow dependence : loop skewing

- Another technique involves the rearrangement of the loop body statements. Example with :  $S2 \delta^f S1$

```
for (int i = 1; i < N; i++)  
{  
    y[i] = f(x[i-1]); // S1  
    x[i] = x[i] + c[i]; // S2  
}
```

- Solution: make sure the statements that consume the calculated values that cause the dependence, use values generated during the same iteration.

# Flow dependence : loop skewing (2)

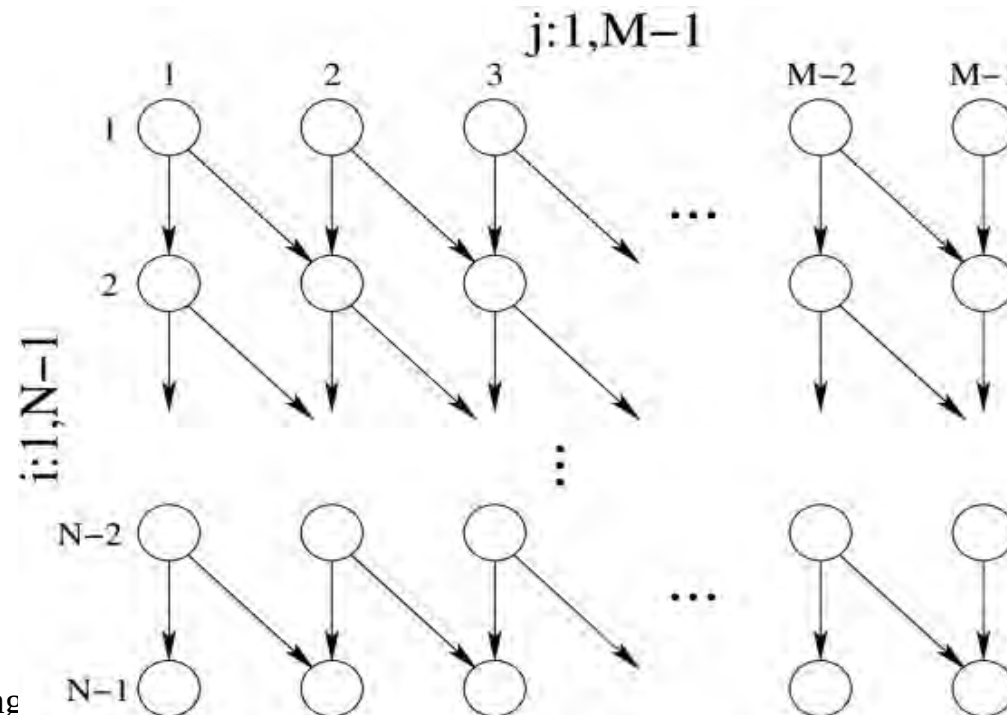
```
y[ 1 ] = f( x[ 0 ] );  
for( int i = 1; i < N - 1; i++)  
{  
    x[ i ] = x[ i ] + c[ i ];  
    y[ i + 1 ] = f( x[ i ] );  
}  
x[ N - 1 ] = x[ N - 1 ] + c[ N - 1 ];
```



# Iteration space dependency graph

- **ISDG** is made up of nodes that represent a single execution of the loop body, and edges that represent dependencies.
- Example:

```
for (int i = 1; i < N; i++)  
  for (int j = 1; j < M; j++)  
    data[i][j] = data[i - 1][j] + data[i - 1][j - 1];
```





# Flow dependencies : partial parallelization

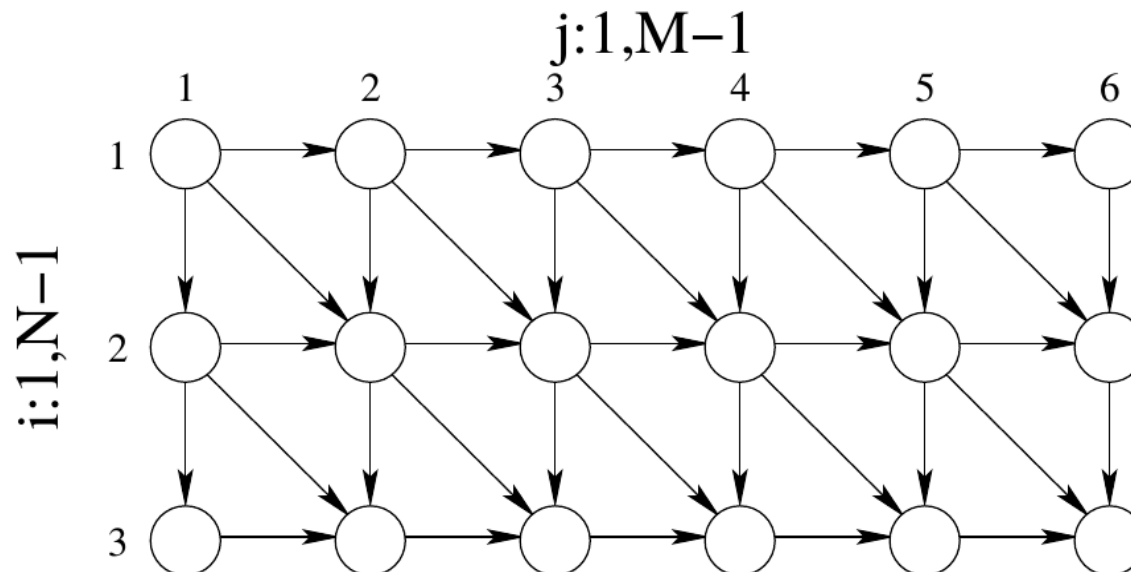
- In the previous example, the j-loop can be parallelized, but the i-loop has to be run sequentially.

```
for (int i = 1; i < N; i++)  
#pragma omp parallel for  
    for (int j = 1; j < M; j++)  
        data[i][j] = data[i - 1][j] + data[i - 1][j - 1];
```

# Flow dependencies : refactoring

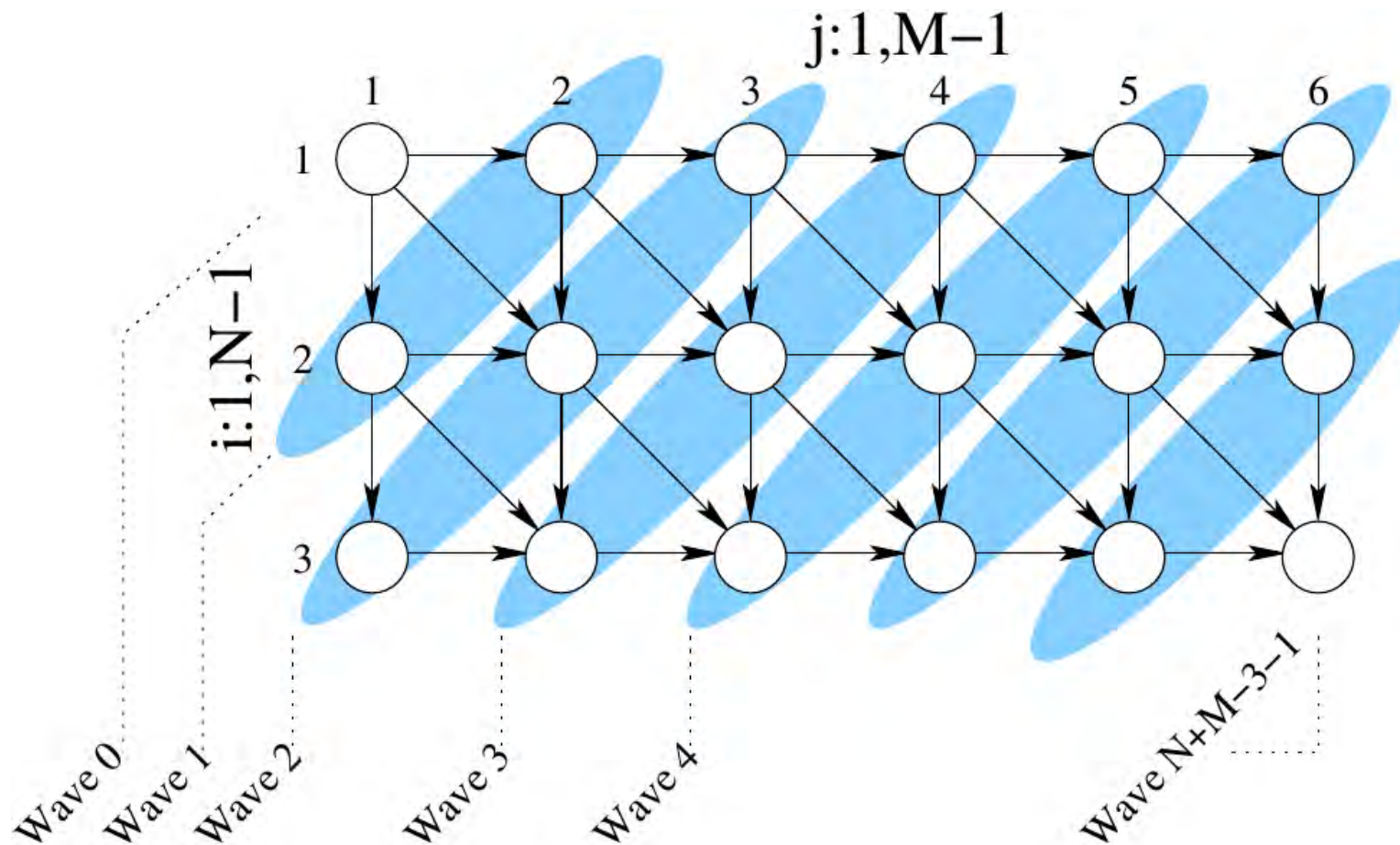
- Refactoring refers to rewriting of the loop(s) so that parallelism can be exposed.
- The ISDG for the following example:

```
for (int i = 1; i < N; i++)  
  for (int j = 1; j < M; j++)  
    data[i][j] = data[i - 1][j] + data[i][j - 1] + data[i - 1][j - 1]; // S1
```



# Flow dependencies : refactoring (2)

- Diagonal sets can be executed in parallel:



# Flow dependencies : fissioning

- Fissioning means breaking the loop apart into a sequential and a parallelizable part. Example:

```
s = b[ 0 ];  
for (int i = 1; i < N; i++)  
{  
    a[ i ] = a[ i ] + a[ i - 1 ];    // S1  
    s = s + b[ i ];  
}
```



```
// sequential part
```

```
for (int i = 1; i < N; i++)  
    a[ i ] = a[ i ] + a[ i - 1 ];
```

```
// parallel part
```

```
s = b[ 0 ];  
#pragma omp parallel for reduction(+ : s)  
for (int i = 1; i < N; i++)  
    s = s + b[ i ];
```

Actually a case of  
prefix-sum

# Flow dependencies : algorithm change

- If everything else fails, switching the algorithm maybe the answer.
- For example, the Fibonacci sequence:

```
for (int i = 2 ; i < N ; i++)  
{  
    int x = F[i - 2]; // S1  
    int y = F[i - 1]; // S2  
    F[i] = x + y;      // S3  
}
```

can be parallelized via Binet's formula:

$$F_n = \frac{\varphi^n - (1 - \varphi)^n}{\sqrt{5}}$$



# Antidependencies

- Example:

```
for (int i = 0; i < N-1; i++)  
{  
    a[ i ] = a[ i + 1 ] + c;  
}
```

- The problem can be solved if we can prevent the „corruption“ of the  $a[i+1]$  values prior to the calculation of  $a[i]$ .
- Solution : save them! **Q.:** *Is this a good idea every time?*

```
for (int i = 0; i < N-1; i++)  
{  
    a2[ i ] = a[ i + 1 ];  
}  
  
#pragma omp parallel for  
for (int i = 0; i < N-1; i++)  
{  
    a[ i ] = a2[ i ] + c;  
}
```

# Nested loops

- As of OpenMP 3.0, perfectly nested loops can be parallelized in unison.
- The **collapse** clause instructs OpenMP how many loops to convert to a single parallel one.
- Example, matrix multiplication:

```
#pragma omp parallel for collapse(2)
  for (int i = 0; i < K; i++)
    for (int j = 0; j < M; j++)
    {
      C[i][j] = 0;
      for (int k = 0; k < L; k++)
        C[i][j] += A[i][k] * B[k][j];
    }
```

- **Q.** : could we modify the above code so that `collapse(3)` would be allowed?

# Loop scheduling

- The way a for loop is partitioned between a team of threads can be controlled.
- These are the available scheduling options:
  - `static`
  - `dynamic`
  - `guided`
  - `auto` : any of the above
- Each option can be accompanied by an optional `chunk_size` parameter, that controls the granularity of the schedule.
- Controlling the schedule can be critical if the iterations are not identical in execution cost.

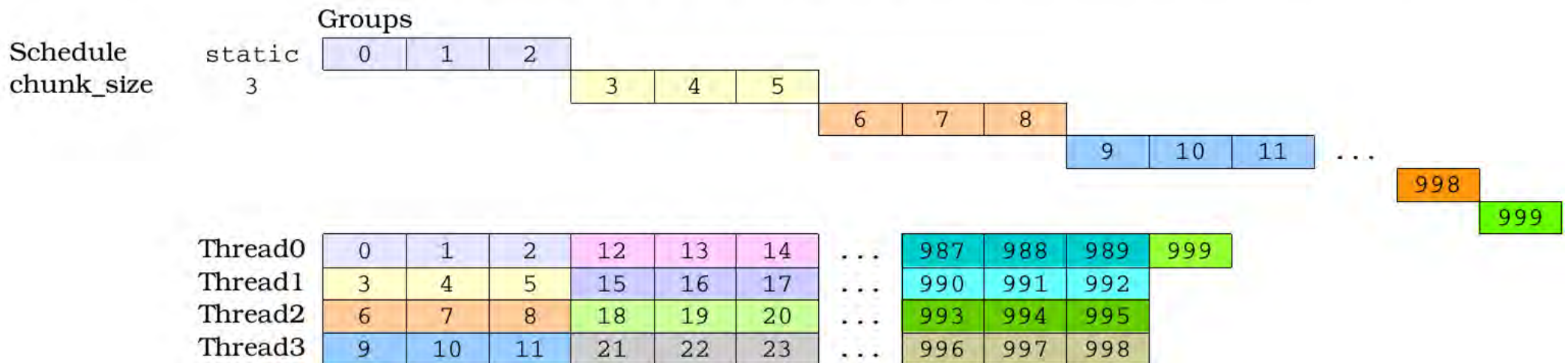


# static schedule

- $N$  iterations are broken up into equal pieces of `chunk_size`, and assigned in a round-robin fashion to the  $p$  threads.
- `chunk_size` defaults to  $\lceil \frac{N}{p} \rceil$

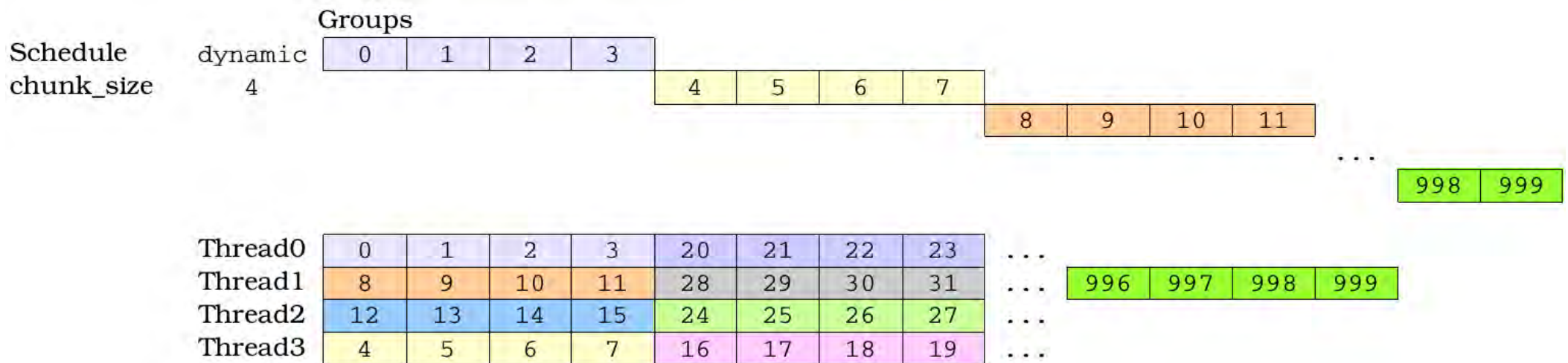
```
for(int i=0;i<1000;i++) {...
```

|   |   |   |   |   |   |   |   |   |   |    |    |     |     |     |
|---|---|---|---|---|---|---|---|---|---|----|----|-----|-----|-----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | ... | 998 | 999 |
|---|---|---|---|---|---|---|---|---|---|----|----|-----|-----|-----|



# dynamic schedule

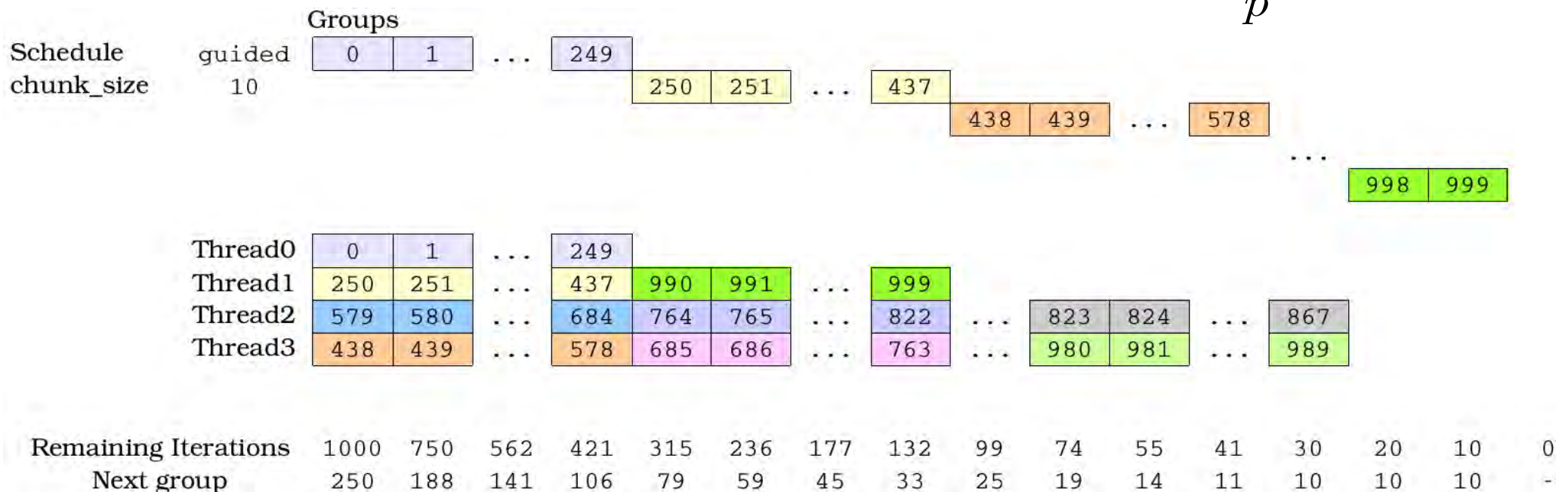
- $N$  iterations are broken up into equal pieces of `chunk_size`, and assigned in a **first-come-first-served** basis to the  $p$  threads.
- Very good candidate for load balancing.
- But, it has a high coordination cost.



# guided schedule

- First-come-first-served assignment of iterations, but the partitioning is uneven.
- Each time a group is to be assigned, its size is calculated by the formula:

$$groupSize = \min(remainingIter, \max(chunk\_size, \lceil \frac{remainingIter}{p} \rceil))$$



# Controlling the schedule

- By setting the `OMP_SCHEDULE` environmental variable. Setting affects all OpenMP programs that will run afterwards. Examples:

```
export OMP_SCHEDULE="static,1"
```

```
export OMP_SCHEDULE="guided"
```

- By using the `omp_set_schedule` function. Syntax:

```
void omp_set_schedule(omp_sched_t kind,  
                     int chunk_size);
```

- Where `kind` is one of:

- `omp_sched_static`
- `omp_sched_dynamic`
- `omp_sched_guided`
- `omp_sched_auto`

# Controlling the schedule (cont.)

- By the `schedule` clause `schedule`. Syntax:

```
#pragma omp parallel for schedule (  
    static | dynamic |  
    guided | auto | runtime  
    [, chunk_size ] )
```

- The `runtime` option delegates the scheduling decision for the execution of the program, where a previous setting (e.g. via `OMP_SCHEDULE`) can be inspected for suggestions. This is exclusive to the `schedule` clause only.

# How to select a schedule option

- `static` : If iterations are „homogeneous“
- `dynamic` : If execution cost varies
- `guided` : if execution cost varies and the number of iteration groups is too high, or the per-iteration cost increases as the loop progresses.
- If in doubt, set:

```
#pragma omp parallel for schedule( runtime )  
for ( . . .
```

# How to select a schedule option (cont.)

- And use a script similar to:

```
#!/bin/bash
# File : schedule_script.sh

for scheme in static dynamic guided
do
    for chunk in 1 2 4 8 16 32
    do
        export OMP_SCHEDULE="${scheme},${chunk}"
        echo $OMP_SCHEDULE `/usr/bin/time -o tmp.log -p $1
>/dev/null ; head -n 1 tmp.log | gawk '{print $2}' ` >> $2
    done
done

Done
```

- The above can be executed as:

```
$ ./schedule_script.sh program_to_test file_to_save_times
```



# Task parallelism

- The `sections` directive can be used to setup individual work items that will be executed by threads. Their relative order of execution (or by which thread it is done) is unknown.

```
#pragma omp parallel
{
    ...
    #pragma omp sections
    ...
}

// OR

#pragma omp parallel sections
{
    ...
}
```

# The section/sections directives

- The individual work items are contained in blocks decorated by section directives:

```
#pragma omp parallel sections
{
    #pragma omp section
    {
        // concurrent block 0
    }
    ...
    #pragma omp section
    {
        // concurrent block M-1
    }
}
```

# Example: producers-consumers in OpenMP

- OpenMP provides a binary mutex type, but using C++ classes is more convenient.
- To combine C++11/17 and OpenMP (and Qt if desired), one just has to add the following lines in a .pro file, before calling the `qmake` utility:

```
QMAKE_CXXFLAGS += -fopenmp -std=c++17
```

```
QMAKE_LFLAGS += -fopenmp -std=c++17
```

- The producers-consumers pattern can be implemented by placing each producer and consumer part in a `section` block.

# Integration using producers-consumers

```
Slice *buffer = new Slice[BUFSIZE];
int in = 0, out = 0;
semaphore avail, buffSlots (BUFSIZE);
mutex l, integLock;
double integral = 0;
#pragma omp parallel sections default(none) \
    shared(buffer, in, out, \
    avail, buffSlots, l, \
    integLock, integral, J)
{
    // producer part
    #pragma omp section
    {
        // producer thread, responsible for handing out 'jobs'
        double divLen = (UPPERLIMIT - LOWERLIMIT) / J;
        double st, end = LOWERLIMIT;
        for (int i = 0; i < J; i++)
        {
            st = end;
            end += divLen;
            if (i == J - 1)
                end = UPPERLIMIT;

            buffSlots.acquire ();

```

main() function  
automatic variables

Fine-tuning  
variable access.  
Not everything  
should be shared.

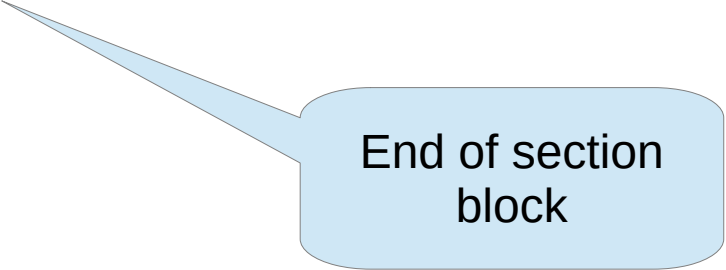
J is the  
number of  
slices to use

Using semaphores  
to manage the buffer

# Consumers part

```
// 1st consumer part
#pragma omp section
{
    integrCalc (buffer, buffSlots, avail, 1, out, integLock, integral);
}

// 2nd consumer part
#pragma omp section
{
    integrCalc (buffer, buffSlots, avail, 1, out, integLock, integral);
}
}
```



End of section  
block



# Consumer code

```
void integrCalc (Slice * buffer, semaphore &buffSlots, semaphore &avail,  
                | mutex &l, int &out, mutex &resLock, double &res)  
{  
    while (1)  
    {  
        avail.acquire ();           // wait for an available item  
        l.lock ();  
        // take the item out  
        double st = buffer[out].start;  
        double en = buffer[out].end;  
        double div = buffer[out].divisions;  
        out = (out + 1) % BUFFSIZE; // update the out index  
        l.unlock ();  
  
        buffSlots.release ();       // signal for a new empty slot  
  
        if (div == 0)  
            break;                 // exit  
  
        //calculate area  
        double localRes = 0;
```

All critical variables passed  
by reference

Typical consumer  
Sequence, using  
semaphores

- The complete code is available online.

# The `task` directive

- Tasks in OpenMP are entities consisting of:
  - **Code** : a block of statements designated to be executed concurrently.
  - **Data** : a set of variables/data owned by the task (e.g. local variables)
  - **Thread Reference** : references the thread (if any) executing the task
- OpenMP performs two activities related to tasks:
  - **Packaging** : creating a structure to describe a task entity
  - **Execution** : assigning a task to a thread
- The task directive decouples the two activities which are joint together in the case of the `section` directive.
- This way, tasks can be dynamically created and executed asynchronously.



# Example

- Traversing a linked list using multiple threads:

```
template <class T>
struct Node
{
    T info;
    Node *next;
};
```

Only one of the team threads executes the following statement/block.

```
#pragma omp parallel
{
    #pragma omp single
    {
        Node<int> *tmp = head;
        while (tmp != NULL)
        {
            #pragma omp task
            process (tmp);
            tmp = tmp->next;
        }
    }
}
```

# The `task` directive clauses

- The task directive can lead to the creation of too many tasks. A set of clauses are provided to limit or control this process:
- **`if (scalar-expression)`** : if the expression evaluates to 0, the generated task becomes **undelayed**, i.e. the current task is suspended, until the generated task completes execution. The generated task may be executed by a different thread. An undelayed task that is executed immediately by the thread that generated it, is called an **included task**.
- **`final (scalar-expression)`** : when the expression evaluates to true, the task and all its child tasks (i.e. other tasks that can be generated by its execution), become **final** and **included**. This means that a task and all its descendants, will be executed by a single thread.
- **`untied`** : a task is by-default tied to a thread : if it gets suspended, it will wait for the particular thread to run it again, even if there are other idle threads. This, in principle, creates better CPU cache utilization. If the untied clause is given, a task may resume execution on any free thread.
- **`mergeable`** : a **merged task** is a task that shares the data environment of the task that generated it. This clause may cause OpenMP to generate a merged task out of an undelayed task.

# task "running wild" example

```
int fib (int i)
{
    int t1, t2;
    if (i == 0 || i == 1)
        return 1;
    else
    {
        #pragma omp task shared(t1) if(i > 25) mergeable
            t1 = fib (i - 1);
        #pragma omp task shared(t2) if(i > 25) mergeable
            t2 = fib (i - 2);
        #pragma omp taskwait
        return t1 + t2;
    }
}
```

Arbitrary threshold.

Second task pragma  
can be removed  
to avoid leaving  
the parent task idle

fib(40) takes 1 sec with the `if` clause, and 108 sec without!

# Task dependencies

- Tasks can have complex dependencies that are declared to the OpenMP runtime with the `depend` clause.

```
#pragma omp task [ depend( dependence-type : list ) ]  
{  
    ...  
}
```

- The `list` contains one or more variable identifiers and/or array sections. These should be either input to the task or be the result of the task computation.
- The `dependence-type` describes the nature of the dependency:
  - `in`: The task is dependent on all previously generated sibling tasks that reference at least one of the list items in an `out` or `inout` dependency type.
  - `out, inout`: The task is dependent on all previously generated sibling tasks that reference at least one of the list items in an `in`, `out`, or `inout` dependency type.
  - `mutexinoutset`: same as `inout`, but two tasks that share the same variable with this type, cannot run in parallel. They will have to execute in sequence.

# Task dependencies example

- In the following, T2 and T3 tasks, depend on T1:

```
#pragma omp parallel
#pragma omp single
{
    for (int i = 0; i < 3; ++i)
    {
        #pragma omp task shared(x) depend(out: x)
        // T1
        printf ("T1 %i\n", i);
        #pragma omp task shared(x) depend(in: x)
        // T2
        printf ("T2 %i\n", i);
        #pragma omp task shared(x) depend(in: x)
        // T3
        printf ("T3 %i\n", i);
    }
}
```

- Output:

```
$ ./depend_test
T1 0
T3 0
T2 0
T1 1
T2 1
T3 1
T1 2
T3 2
T2 2
```

# The `taskloop` directive

- The `taskloop` is a direct replacement for the `parallel for` directive, where the loop iterations are partitioned amongst tasks instead of a team of threads.
- The `reduction`, `shared` and `private` clauses can be used.
- There is no `schedule` clause though. The programmer can offer a hint about the partitioning by using the `grainsize` clause:

```
#pragma omp taskloop grainsize(GS)
```

- OpenMP will break the loop by assigning between `GS` and  $2 * GS - 1$  iterations to each task.



# Synchronization constructs

- **critical** : allows only one thread at a time, to enter the structured block that follows. The syntax involves an optional identifier:

```
#pragma omp critical [ ( identifier ) ]  
{  
    // structured block  
}
```

- The identifier allows the establishment of **named critical sections**. All critical directives without an identifier are assumed to have the same name, and use the same mutex.
- **atomic** : this is a lightweight version of the `critical` construct. Only a single statement (not a block) can follow.

# Synchronization constructs (cont.)

- Allowed statements for `atomic`:

```
x++;
```

```
x--;
```

```
++x;
```

```
--x;
```

```
x binop= expr;
```

```
x = x binop expr;
```

```
x = expr binop x;
```

where `x` has to be a variable of scalar type and `binop` can be one of

`+, *, -, /, &, ^, |, <<, >>`

and `expr` is a scalar expression.

- Caution should be used in the calculation of the `expr` above. In the following example:

```
#pragma omp atomic
```

```
x += y++;
```

the update to `y` is not atomic.

# Synchronization constructs (cont.)

- **master**, **single** : both force the execution of the following structured block by a single thread. There is a significant difference : `single` implies a barrier on exit from the block.
- The `master` can be used for I/O operations.
- **barrier** : blocks until all team threads reach that point.
- **taskwait** : applies to a team of tasks. Blocks until all child tasks terminate.
- **ordered** : used inside a `parallel for`, to ensure that a block will be executed as if in sequential order.

# master example

```
int examined = 0;
int prevReported = 0;
#pragma omp for shared( examined, prevReported )
    for( int i = 0 ; i < N ; i++ )
    {
        // some processing

        // update the counter
#pragma omp atomic
        examined++;

        // use the master to output an update every 1000 newly ←
        finished iterations
#pragma omp master
    {
        int temp = examined;
        if( temp - prevReported >= 1000)
        {
            prevReported = temp;
            printf("Examined %.2lf%%\n", temp * 1.0 / N );
        }
    }
}
```

# taskwait example

- Post-order tree traversal:

```
template < class T > struct Node
{
    T info;
    Node *left, *right;

    Node (int i, Node < T > *l, Node < T > *r) :
        info (i), left (l), right (r) { }
};
```

```
template < class T > void postOrder (Node < T > *n)
{
    if (n == NULL)
        return;

    #pragma omp task
    postOrder (n->left);
    #pragma omp task
    postOrder (n->right);
    #pragma omp taskwait

    process (n->info);
}
```

# ordered example

```
double data[ N ];  
#pragma omp parallel shared( data, N )  
{  
  
#pragma omp for ordered schedule( static, 1 )  
    for(int i = 0; i < N; i++)  
    {  
        // process the data  
  
        // print the results in order  
#pragma omp ordered  
        cout << data[i];  
    }  
}
```

ordered clause  
is required



# The `flush` directive

- The `flush` directive is used as a *memory barrier*. It makes a thread's view of certain variables, consistent with main memory.
- All memory operations, initiated before the `flush`, must complete before the `flush` can complete, i.e. the modifications have to propagate from the cache/registers to main memory.
- All operations that follow the `flush` directive cannot commence before the `flush` is complete. Access to shared variables after the `flush`, requires fresh access to main memory.
- The benefit of using `flush` is that we do not have to rely on the execution platform for proper memory consistency.

# flush example

```
bool flag = false;
#pragma omp parallel sections default( none ) shared( flag , cout ↵
{

#pragma omp section
{
    // wait for signal
    while (flag == false)
    {
#pragma omp flush ( flag )
    }
    // do something
    cout << "First section\n";
}

#pragma omp section
{
    // do something first
    cout << "Second section\n";
    sleep (1);
    // signal other section
    flag = true;

#pragma omp flush ( flag )

}

}
```

# Cancellation constructs

- A cancellation construct allows the termination of the innermost region that encloses it. It is a convenient way for controlling the flow of parallel execution when conditions dictate it.
- Cancellation is implemented with the use of two directives:
  - **cancel** : used to establish/declare to the OpenMP runtime that a parallel region should be canceled. The affected region can be one of `parallel`, `for`, `sections`, or `taskgroup`.
  - **cancellation point** : allows the executing thread to check if cancellation has been activated.
- The use of the `cancellation point` directive is not mandatory. Cancellation can be also detected by a thread when a barrier (explicit or implicit) is encountered.
- When a thread detects a cancellation event, it transfers execution to the end of the canceled region.
- Normal execution resumes when all the threads reach the end of the canceled region.

# Cancellation constructs example

- When a thread computes a desired outcome, the computation can stop:

```
bool found=false;
#pragma omp parallel firstprivate( found )
{
    #pragma omp for
    for (int i = 0; i < N; i++)
    {
        found = doSmt( i ); // return true if something is accomplished
    }

    #pragma omp cancel for if(found)

    #pragma omp cancellation point for
}
}
```

# SIMD extensions

- Recent CPU architectures have incorporated wide vector processing capabilities. For example, there is the Advanced Vector Extensions (AVX) instruction sets in the x86-64 world or the 128 bit vector instructions in ARM.
- Using these instruction sets requires special compiler optimizations and/or using special libraries.
- OpenMP exposes these capabilities through the `simd` directive, which comes in three flavors:
  - `simd` loop directive : This is used to convert a canonical-form loop into a sequence of SIMD instructions.
  - `for simd` loop directive : This is used to parallelize a loop, where each thread is also employing SIMD instructions for executing its share of the iterations in batches. The loop must be - as always - in a canonical form.
  - `declare simd` directive : This is used to declare that a function can be "inlined" and multiple calls to it with different parameters can be executed in a SIMD fashion. So multiple calls can be replaced with one.

# SIMD extensions (cont.)

- In order to assist the compiler in the conversion, the following optional clauses are available:
  - `simdlen(positive_integer)`: Declares the preferred number of iterations to be executed in SIMD fashion. The actual number that OpenMP will use depends on both the hardware platform and the loop to be transformed.
  - `safelen(positive_integer)`: Specifies the maximum number of successive iterations that can be grouped ``safely" together for SIMD execution.



# SIMD extensions example

- SAXPY vector operation:

```
#pragma omp declare simd uniform(a)
float saxpy (float a, float x, float y)
{
    return a * x + y;
}
```

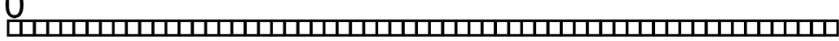
uniform means that a will be the same across calls

```
int main (int argc, char **argv)
{
    int N = atoi (argv[1]);
    int useSIMD = atoi (argv[2]);

    unique_ptr < float[] > x = make_unique < float[] > (N);
    unique_ptr < float[] > y = make_unique < float[] > (N);
    unique_ptr < float[] > z = make_unique < float[] > (N);
    #pragma omp parallel for simd if(useSIMD>0)
    for (int i = 0; i < N; i++)
        x[i] = y[i] = i;

    float a = 1.0;
    #pragma omp parallel for simd if(useSIMD>0)
    for (int i = 0; i < N; i++)
        z[i] = saxpy (a, x[i], y[i]);
}
```

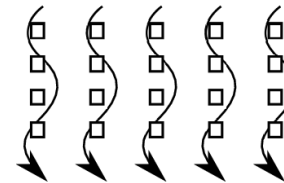
# OpenMP different loops visualization

(a)  0 N-1

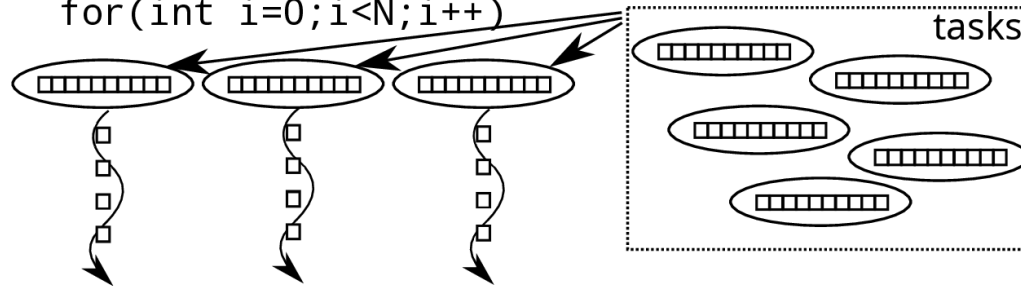
(b) `for(int i=0;i<N;i++)`



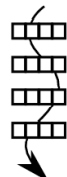
(c) `#pragma omp for`  
`for(int i=0;i<N;i++)`



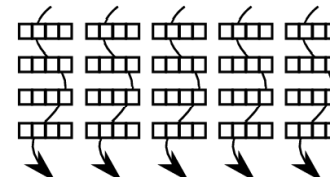
(d) `#pragma omp taskloop`  
`for(int i=0;i<N;i++)`



(e) `#pragma omp simd`  
`for(int i=0;i<N;i++)`



(f) `#pragma omp for simd`  
`for(int i=0;i<N;i++)`



# Offloading to devices

- OpenMP supports the migration of workload (so called offloading) to accelerators such as GPUs or many-core computing devices (such as Intel's Phi).
- OpenMP identifies these devices as **targets**.
- Devices can be queried and selected with the following functions:

```
// Returns the number of available devices
int omp_get_num_devices();

// Returns the number that corresponds to the default device
int omp_get_default_device();

// Sets the default device to be the one identified by "dev_num"
void omp_set_default_device(int dev_num);

// Returns the device number the executing thread is running on
int omp_get_device_num();

// Returns a number identifying the host "device"
int omp_get_initial_device();
```

# Offloading to devices (cont.)

- A target device can be selected also by using:
  - The `device(device_no)` clause in supported directives.
  - Using the `OMP_DEFAULT_DEVICE` environment variable, e.g.  

```
$ export OMP_DEFAULT_DEVICE=0
```
- The directives that OpenMP incorporates for device operations support, fall into two categories:
  - Work-sharing constructs
  - Data managements constructs
- OpenMP uses the latter to ensure data remain consistent across the different memory spaces that the CPU and the devices can have.

# Offloading work sharing directives

- The **target** directive migrates the structured block that follows and the data required to run it to a device:

```
#pragma omp target  
    structured_block
```

- By default, the scalar data and pointers referenced in the structured block are treated as firstprivate and the arrays are copied to the device before the start of the execution and from the device after the execution is complete.

-

# Work sharing : `teams` directive

- The `teams` directive creates a **league**/collection of **teams**/groups of threads.
- One thread from each team is initially assigned to run the associated structured block. The `teams` is equivalent to the `parallel` directive in the sense that all the initial threads execute the same block.
- Each team of threads corresponds to a CUDA block or an OpenCL work group. So the league of teams corresponds to a CUDA grid or an OpenCL index space.
- All the threads that are initiated by `teams` and its nested directives are called the **contention group**.
- The league and team size are controlled by these optional clauses:
  - `num_teams ( [lower-bound:] upper-bound )`: If the optional lower-bound is missing, a fixed number of teams are created, equal to the upper-bound.
  - `thread_limit ( integer-expression )`: Upper limit for the total number of threads for all the teams (not each team individually).



# Work sharing : `distribute` directive

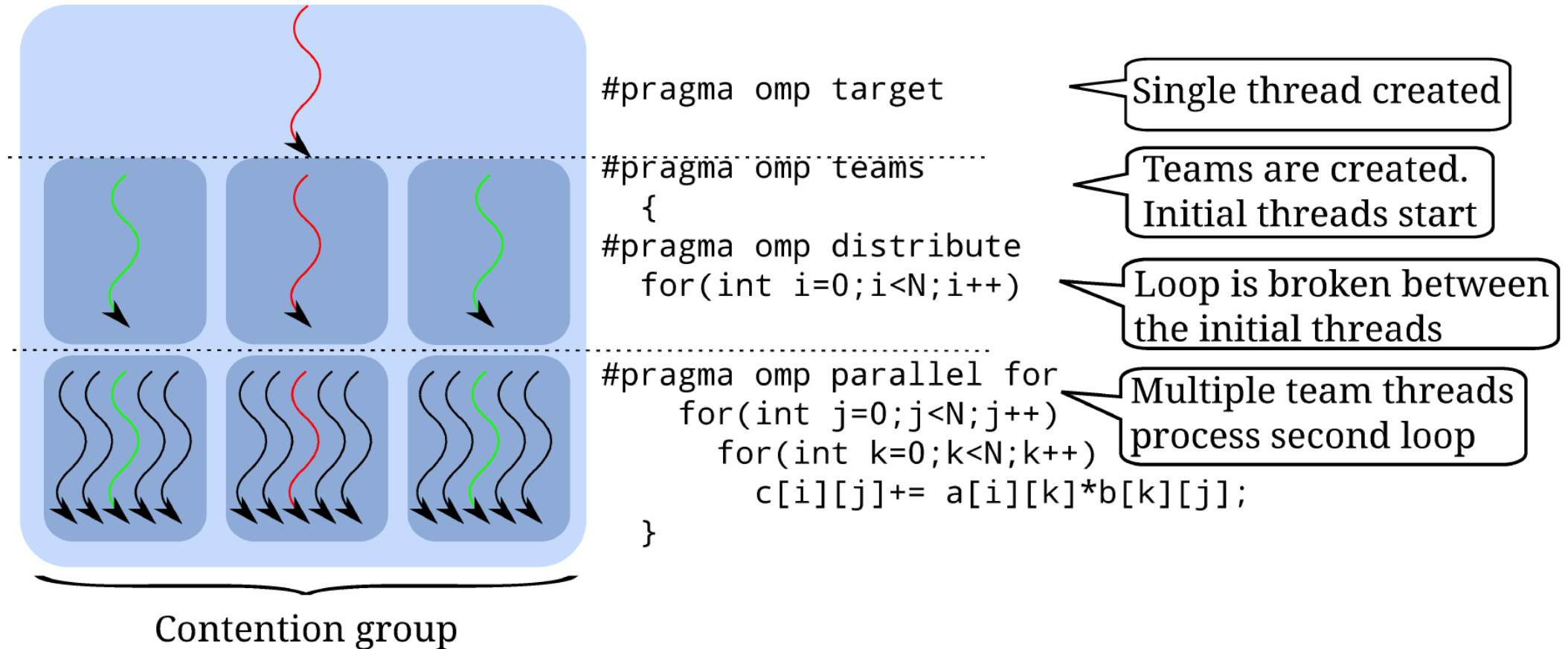
- The `distribute` directive is used to break a canonical-form loop between the teams of a league.
- So it is one way of hinting to OpenMP how many teams it needs to create.
- Any code that is between the `teams` and `distribute` directives will be executed by all the initial team threads.
- A typical scenario would be:

```
#pragma omp target
#pragma omp teams
{
#pragma omp distribute
    for(int i=0;i<N;i++)           // i-loop will be distributed
#pragma omp parallel for
    for(int k=0;k<M;k++)           // k-loop will be run by teams
    ...
```

# Work sharing : thread identification

- The following functions allow a thread to identify its location in the league and team:
  - `omp_get_team_num()` : returns the team number (CUDA equivalent : `blockIdx`).
  - `omp_get_num_teams()` : returns the number of teams created (CUDA equivalent : `gridDim`).
  - `omp_get_thread_num()` : (CUDA equivalent : `threadIdx`).
  - `omp_get_team_size()` : (CUDA equivalent : `blockDim`).
- These are available inside a `teams` construct.

# Teams example



# OpenMP compilation for offloading

- Compilation requires the use of special flags and switches as a cross-compiler needs to be involved for generating the device binaries.

- Example:

- GNU C++:

```
$ g++ -fopenmp -foffload=nvptx-none -fno-stack-protector -foffload=-lm teamsEx.cpp -o teamsEx
```

Makes sure device math libraries are Linked.

- CLANG

```
$ clang++-9 -fopenmp -fopenmp-targets=nvptx64 teamsEx.cpp -o teamsEx
```

- The nvptx string implies the use on NVidia's PTX format.

# Code and data for offloading

- Code and data that need to reside or run on a device have to be declared inside a **omp declare target** construct.

- Example:

```
#pragma omp declare target
double testf (double x)
{
    return x * x + 2 * sin (x);
}
double sum=0;
#pragma omp end declare target
```

- The default action is to make whatever is inside the block available to both the host and the device.
- Alternatively, the optional **device\_type** clause can alter this based on its parameter:
  - **any**: The default option.
  - **host**: Makes the enclosed items available only on the host.
  - **nohost**: Availability is limited to the device.

# reduction clause on teams

- The `teams` clause can be decorated with a reduction clause, but this can produce erroneous results as it requires inter-team coordination.
- A safer option is to perform the reduction at the team-level, and after the teams are finished, to have the host perform a final reduction.
- Example using integration:

numTeams is  
a program constant

```
double integrate (double st, double en, int div)
{
    double localRes = 0;
    double step = (en - st) / div;
    unique_ptr < double[] > TR = make_unique < double[] > (numTeams);
    double *teamResults = TR.get ();
    #pragma omp target data map(tofrom: teamResults[0:numTeams])
    #pragma omp target teams num_teams(numTeams)
    {
        int teamID = omp_get_team_num ();
        #pragma omp distribute parallel for reduction(+:teamResults[teamID])
        for (int i = 1; i < div; i++)
        {
            double x = st + i * step;
            teamResults[teamID] += testf (x);
        }
    }
    // consolidate partial results
    for (int i = 0; i < numTeams; i++)
        localRes += teamResults[i];

    localRes += (testf (st) + testf (en)) / 2;
    localRes *= step;
    return localRes;
}
```



# Data management constructs

- Controlling the flow of data between the host and the device is critical not only for correctness but also for performance.
- Ideally, we should be able to have data persist on the device without unnecessary data copies to the host, between offloading constructs.
- The **target data** directive can be used to establish a data environment by surrounding any other target constructs. For example:

```
#pragma omp target data
{
// first target construct
#pragma omp target teams
. . .
// no host sync here
// second target construct
#pragma omp target teams
. . .
}
```

- A **map** clause can be used to fine-tune data movement.

# The `map` clause

- The `target data` directive can have one or more `map` clauses that describe how data move between the host and the device.

- The syntax of the map clause is:

```
map( [ [map-type-modifier*] map-type : ] list)
```

- The `list` is a list of variables, arrays or array sections.
- The map-type can be one of:
  - `to` : The device-side variable is initialized from the host-side one. This is the default map type for scalar data.
  - `from` : The device-side variable value is copied to the host-side one.
  - `tofrom` : A combination of `to` and `from`. This is the default map-type for arrays.
  - `alloc` : Allocates (but does not initialize) data on the device.
  - `release` : Decrements the reference count of a variable.
  - `delete` : Sets the reference count of a variable to zero and deletes the device data.
- The reference count has to do with automatic host variables. Whenever such a variable is mapped to a device, its count is incremented.

# Array sections

- An array section is just a part of an array that can be used in a `map` clause to limit the data transfer to just the desired array elements.
- An array section can be described as follows:

```
array[ start_offset : length : stride] or  
array[ start_offset : length : ] or  
array[ start_offset : length ] or  
array[ start_offset : : stride] or  
array[ start_offset : : ] or  
array[ start_offset : ] or  
array[ : length : stride] or  
array[ : length : ] or  
array[ : length ] or  
array[ : : stride]  
array[ : : ]  
array[ : ]
```

When the `start_offset` is missing, it defaults to 0.

When the `stride` is missing, it defaults to 1.

# target enter data and target exit data directives

- This pair of directives offers more precise control on data upon entering and upon exiting a `target` construct.
- They can be used as a replacement to the `target data` directive.
- A **target update** directive can be also used within a `target` construct to e.g. allow intermediate results to be moved to the host.
- The allowed map types for these directives are shown below:

| Map type | target | target data | target enter data | target exit data |
|----------|--------|-------------|-------------------|------------------|
| to       | X      | X           | X                 |                  |
| from     | X      | X           |                   | X                |
| tofrom   | X      | X           |                   |                  |
| alloc    | X      | X           | X                 |                  |
| release  |        |             |                   | X                |
| delete   |        |             |                   | X                |

# Data management example

- Computing  $\vec{a} \cdot (\vec{a} + \vec{b})$

```
int a[N], b[N], c[N], d[1];
d[0] = 0;
. . .
#pragma target enter data map(to:a,b,d) map(alloc:c)

#pragma omp target teams distribute parallel for
  for (int i = 0; i < N; i++)
    c[i] = a[i] + b[i];

#pragma omp target update from(c)

#pragma omp target exit data map(delete:b)

#pragma omp target teams
{
#pragma omp distribute parallel for reduction(+:d[0])
  for (int i = 0; i < N; i++)
    d[0] += a[i] * c[i];
}

#pragma target exit data map(from:d) map(release:a,c)
```

d has to be defined as an array if it is to be retrieved from the device

# Data management example data trace

- Assuming  $\vec{a} = \vec{b} = 1, 2, 3$

**#pragma target enter data map(to:a,b,d) map(alloc:c)**

**#pragma omp target teams distribute parallel for**  
 for (int i = 0; i < N; i++)  
 c[i] = a[i] + b[i];

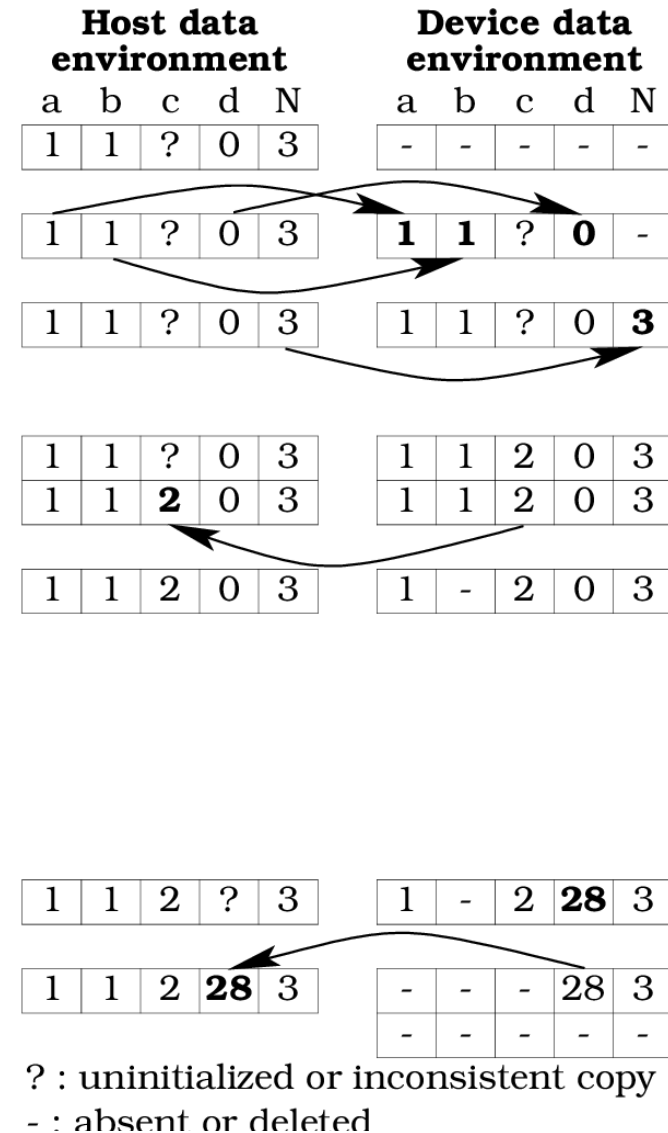
**#pragma omp target update from(c)**

**#pragma omp target exit data map(delete:b)**

**#pragma omp target teams**

{  
**#pragma omp distribute parallel for reduction(+:d[0])**  
 for (int i = 0; i < N; i++)  
 d[0] += a[i] \* c[i];  
 }

**#pragma target exit data map(from:d) map(release:a,c)**





# CUDA interoperability

- CUDA comes with a rich portfolio of accelerated libraries such as CUBLAS, nvGRAPH, CUFFT, etc.
- Being able to switch between CUDA and OpenMP would allow one to leverage both the convenience of the OpenMP directives and the performance of the CUDA library functions.
- This is an easy problem to solve if we use the `#pragma omp requires unified_shared_memory` directive. If this is not an option, then we must declare to OpenMP whether certain pointers point to CUDA-allocated device memory.

# CUDA interoperability (cont.)

- Doing so requires the use of two clauses:
  - `is_device_ptr( list_of_pointers )` : This can be used as part of the `target` directive to establish access to CUDA-allocated data to OpenMP. (CUDA → OpenMP)
  - `use_device_ptr( list_of_pointers )` : This can be used as part of the `target data` directive to establish access to OpenMP-managed device data from CUDA functions. (OpenMP → CUDA).

- Example:

```
cudaMalloc ((void **) &da, sizeof (float) * N);
cudaMalloc ((void **) &db, sizeof (float) * N);
cudaMalloc ((void **) &dc, sizeof (float) * N);
. . .
// data transfer, host -> device
cudaMemcpy (da, ha.get(), sizeof (float) * N, cudaMemcpyHostToDevice);
cudaMemcpy (db, hb.get(), sizeof (float) * N, cudaMemcpyHostToDevice);

#pragma omp target is_device_ptr(da,db,dc)
#pragma omp teams distribute parallel for
for (int i = 0; i < N; i++)
    dc[i] = da[i] + db[i];
```

Data is not moved  
to the device by OpenMP

Compilation will have  
to be done with `nvcc`

# The `loop` directive

- OpenMP 5.0 introduced the `loop` construct as a way to automate and unify the concurrent execution of loop iterations using different mechanisms and targeting both CPUs and GPUs.
- The goal is that future OpenMP development will only have to use the `loop` construct instead of the `parallel for`, or `teams-distribute` combination.
- The `loop` directive can be used on a canonical-form loop the same way a `for` does, but it permits more aggressive optimizations because it does not allow the use of other constructs inside it (apart from `parallel`, `loop` and `simd`).
- The directive can have a **`bind`** clause that hints the compiler about the execution of the loop. Its parameter can be one of:
  - **`parallel`** : The loop will be executed on the CPU if the enclosing construct is `parallel`.
  - **`teams`** : If the innermost region enclosing the construct is a `teams` region, then the loop will be executed on the device.
  - **`thread`** : the compiler will decide

# loop directive example

- For GPU execution to succeed, the compilation needs to be setup accordingly, with the appropriate switches and libraries.

```
// CPU execution
#pragma omp parallel
#pragma omp loop
    for(int i=0;i<N;i++)
        c[i] = a[i]+b[i];

// GPU execution
#pragma omp target teams
#pragma omp distribute
#pragma omp loop
    for(int i=0;i<N;i++)
        c[i] = a[i]+b[i];
. . .
```

# Thread safety

- **Thread-safe** functions are functions that can be called concurrently from multiple threads without any ill-effects to the program.
- Often confused with reentrant functions.
- A function can be reentrant, or thread-safe, or both, or neither of the two.
- A **reentrant** function can be interrupted and called again (re-entered) before the previous calls are complete.
- Thread-safe functions provide linearizable access to shared data.

# Reentrant functions

- The conditions that need to be met for a function to be reentrant are:
  - The function should not use **static** or global data. Global data may be accessed (e.g. hardware status registers) but they should not be modified unless atomic operations are used.
  - In the case of an object method, either the method is an accessor method (getter), or it is a mutator (setter) method, in which case the object should be modified inside a critical section.
  - All data required by the function should be provided by the caller. If a program calls a function multiple times, with the same arguments, it is the responsibility of the caller to ensure that the calls are properly done. For example, the `qsort_r` C-library function is a reentrant implementation of the quicksort algorithm. If two threads call this function with the same input array, the results cannot be predicted.
  - The function does not return pointers to static data. If an array needs to be returned, it can be, either, dynamically allocated, or, provided by the caller.
  - The function does not call any non-reentrant functions.
  - The function does not modify its code, unless private copies of the code are used in each invocation.



# CPU caches

- CPU caches are organized in cache lines, that hold consecutive memory locations in an effort to take advantage of temporal and spatial locality.
- A typical size for cache lines is 64 bytes. Each cache line is associated with an address (where did the data come from) and a state.
- Multicore CPUs usually employ a **coherency protocol**, i.e. a set of rules for how shared data, kept at disjoint caches, are maintained in a consistent state.

# The MESI model

A simple such protocol is MESI, that employs four states:

- **Modified** : the CPU has recently changed part of the cache line, and the cache line holds the only up-to-date value of the corresponding item. No other CPU can hold copies of these data, so the CPU can be considered the owner of the data. The pending changes are supposed to be written back to the main memory according to the rules of the CPU architecture.
- **Exclusive** : similar to the modified state, in that the CPU is considered the owner of the data. No change has been applied though. The main memory and the cache hold identical values.
- **Shared** : at least one more cache holds a copy of the data. Changes to the data can only be performed after coordination with the other CPUs holding copies.
- **Invalid** : represents an empty cache line. It can be used to hold new data from the main memory.

# An example with two threads

```
double x[N];
#pragma omp parallel for schedule(static, 1)
  for( int i = 0; i < N; i++ )
    x[i] = someFunc( x[i] );
```

**Main Memory**

|   |        |        |        |        |        |        |     |
|---|--------|--------|--------|--------|--------|--------|-----|
|   | 0x8000 | 0x8008 | 0x8010 | 0x8018 | 0x8020 | 0x8028 |     |
| x |        |        |        |        |        |        | ... |

**Core 0 Cache**

| State | Address | Data        |      |             |      |             |      |             |      |
|-------|---------|-------------|------|-------------|------|-------------|------|-------------|------|
|       | ...     |             |      |             |      |             |      |             |      |
| S     | 0x8000  | <b>x[0]</b> | x[1] | <b>x[2]</b> | x[3] | <b>x[4]</b> | x[5] | <b>x[6]</b> | x[7] |
|       | ...     |             |      |             |      |             |      |             |      |

What happens to Core 1 cache when Core 0 changes x[0]?

**Core 1 Cache**

| State | Address | Data |             |      |             |      |             |      |             |
|-------|---------|------|-------------|------|-------------|------|-------------|------|-------------|
|       | ...     |      |             |      |             |      |             |      |             |
| S     | 0x8000  | x[0] | <b>x[1]</b> | x[2] | <b>x[3]</b> | x[4] | <b>x[5]</b> | x[6] | <b>x[7]</b> |
|       | ...     |      |             |      |             |      |             |      |             |

# False sharing

- **False sharing** : sharing cache lines without actually sharing data.
- How to fix it:
  - Pad the data
  - Change the mapping of data to cores
  - Use private/local variables

# Padding the data

- Original

```
double x[N];  
#pragma omp parallel for schedule(static, 1)  
    for( int i = 0; i < N; i++ )  
        x[ i ] = someFunc( x [ i ] );
```

- Padded:

```
double x[N][8];  
#pragma omp parallel for schedule(static, 1)  
    for( int i = 0; i < N; i++ )  
        x[ i ][ 0 ] = someFunc( x [ i ][ 0 ] );
```

- Can kill cache effectiveness.
- Wastes memory.

# Data mapping change

```
double x[N];  
#pragma omp parallel for schedule(static, 8)  
    for( int i = 0; i < N; i++)  
        x[i] = someFunc( x[i] );
```



# Using private variables

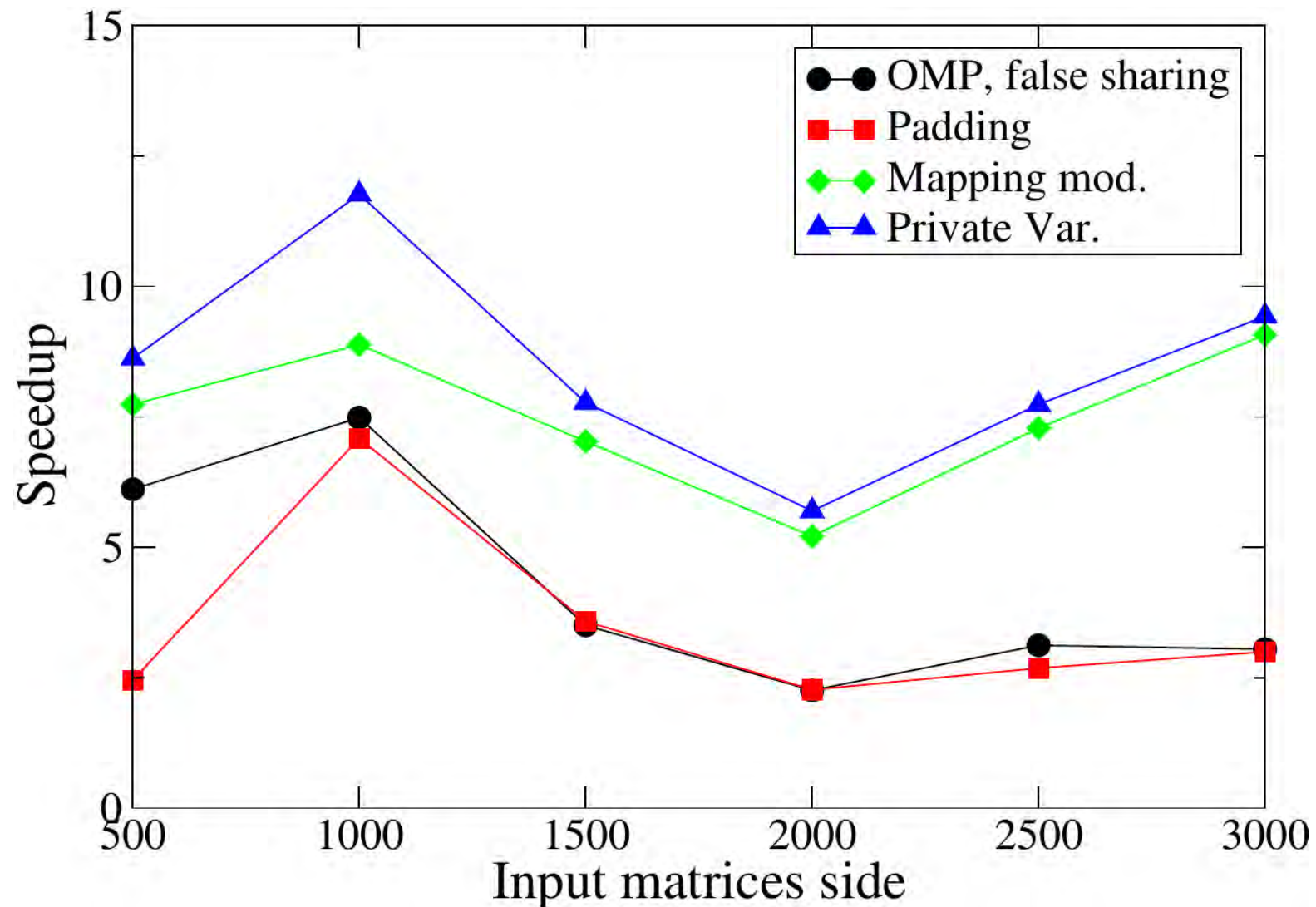
```
// assuming that N is a multiple of 8
double x[N];
#pragma omp parallel for schedule(static, 1)
    for( int i = 0; i < N; i += 8 )
    {
        double temp[ 8 ];
        for( int j = 0; j < 8; j++)
            temp[ j ] = someFunc( x [ i + j ] );
        memcpy( x + i, temp, 8 * sizeof( double ) );
    }
```

# Using a private variable for matrix multiplication

```
#pragma omp parallel for collapse(2)
for (int i = 0; i < N; i++)
    for (int j = 0; j < M; j++)
    {
        double temp = 0;
        for (int l = 0; l < K; l++)
            temp += A[i * K + l] * B[l * M + j];
        C[i * M + j] = temp;
    }
```

- But how severe is the false sharing problem to even consider it?

# Matrix multiplication test



# A case study : sorting in OpenMP

```
template < class T > void mergesort (T * data, int N)
{
    // allocate temporary array
    unique_ptr<T[]> temp = make_unique<T[]>(N);
    // pointers to easily switch between the two arrays
    T *repo1, *repo2, *aux;

    repo1 = data;
    repo2 = temp.get();

    // loop for group size growing exponentially from 1 element
    for (int grpSize = 1; grpSize < N; grpSize <= 1)
    {
        for (int stIdx = 0; stIdx < N; stIdx += 2 * grpSize)
        {
            int nextIdx = stIdx + grpSize;
            int secondGrpSize = min (max (0, N - nextIdx), grpSize);

            // check to see if there are enough data for a second group to merge↵
            with
            if (secondGrpSize == 0)
            {
                // if there is no second part, just copy the first part to repo2↵
                for use in the next iteration
                for (int i = 0; i < N - stIdx; i++)
                    repo2[stIdx + i] = repo1[stIdx + i];
            }
        }
    }
}
```

Sequential alg.



# A case study : sorting in OpenMP (2)

```
        else
        {
            mergeList (repo1 + stIdx, repo1 + nextIdx, grpSize, ←
secondGrpSize, repo2 + stIdx);
        }
    }

    // switch pointers
    aux = repo1;
    repo1 = repo2;
    repo2 = aux;
}

// move data back to the original array
if (repo1 != data)
    memcpy (data, temp.get(), sizeof (T) * N);
}
```

- The two for loops cannot be collapsed. Why?

# OpenMP bottom-up mergesort

- It takes only one line to turn the sequential program into a multithreaded one:

```
...  
    for (int grpSize = 1; grpSize < N; grpSize <=<= 1)  
    {  
#pragma omp parallel for  
        for (int stIdx = 0; stIdx < N; stIdx += 2 * grpSize)  
        {  
...  
    }
```



# OpenMP top-down mergesort

- Sequential recursive function `mergeList` always copies the sorted data back to the original array. Front-end function is not shown.

```
template < class T > void mergesortRec (T * data, T * temp, int N)
{
    // base case
    if (N < 2)
        return;
    else
    {
        int middle = N/2;
        mergesortRec(data, temp, middle);
        mergesortRec(data+middle, temp+middle, N - middle);
        mergeList(data, data+middle, middle, N-middle, temp);
    }
}
```

# Top-down, multi-threaded front-end

```
template < class T > void mergesort (T * data, int N)
{
    // allocate temporary array
    T *temp = new T[N];

    #pragma omp parallel
    {
        #pragma omp single
        {
            int middle = N / 2;
            #pragma omp task
            {
                mergesortRec (data, temp, middle);
            }
            #pragma omp task
            {
                mergesortRec (data + middle, temp + middle, N - middle);
            }

            #pragma omp taskwait

            mergeList (data, data + middle, middle, N - middle, temp);
        }

        delete [] temp;
    }
}
```

Mult}

# Top-down, multi-threaded recursive

```
template < class T > void mergesortRec (T * data, T * temp, int N)
{
    if (N < 2)
        return;
    else
    {
        int middle = N / 2;
#pragma omp task if(N>10000) mergeable
        {
            mergesortRec (data, temp, middle);
        }
#pragma omp task if(N>10000) mergeable
        {
            mergesortRec (data + middle, temp + middle, N - middle);
        }

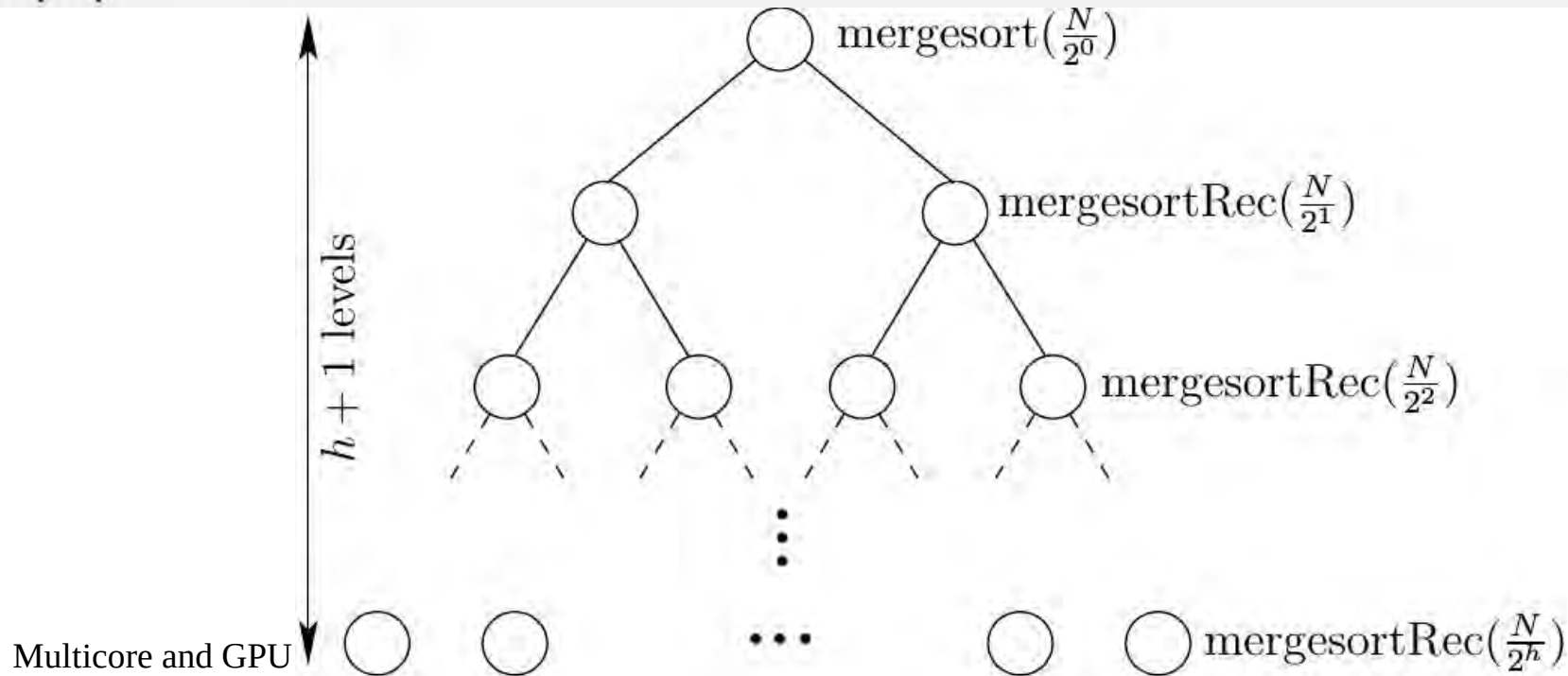
#pragma omp taskwait

        mergeList (data, data + middle, middle, N - middle, temp);
    }
}
```

# Limiting the number of tasks

```
// A global numTasks counter is used to enumerate the number of tasks
// generated and limit the generation of new ones
#pragma omp task if(numTasks < maxTasks) mergeable
{
#pragma omp atomic
    numTasks++;

    mergesortRec (data, temp, middle);
}
. . .
```





# A more effective limit on the number of tasks

```
const int maxTasks=4096;
int _thresh_; // used for the if clauses
. . .
//-----
// sort data array of N elements, using the aux array as temporary ↵
// storage
template < class T > void mergesortRec (T * data, T * temp, int N)
{
    if (N < 2)
        return;
    else
    {
        int middle = N / 2;
#pragma omp task if(N > _thresh_ ) mergeable
//-----
template < class T > void mergesort (T * data, int N)
{
    // allocate temporary array
    T *temp = new T[N];
    _thresh_ = 2.0 * N / (maxTasks + 1);

#pragma omp parallel
    {
        . . .
```

# Results

