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.

Introduction

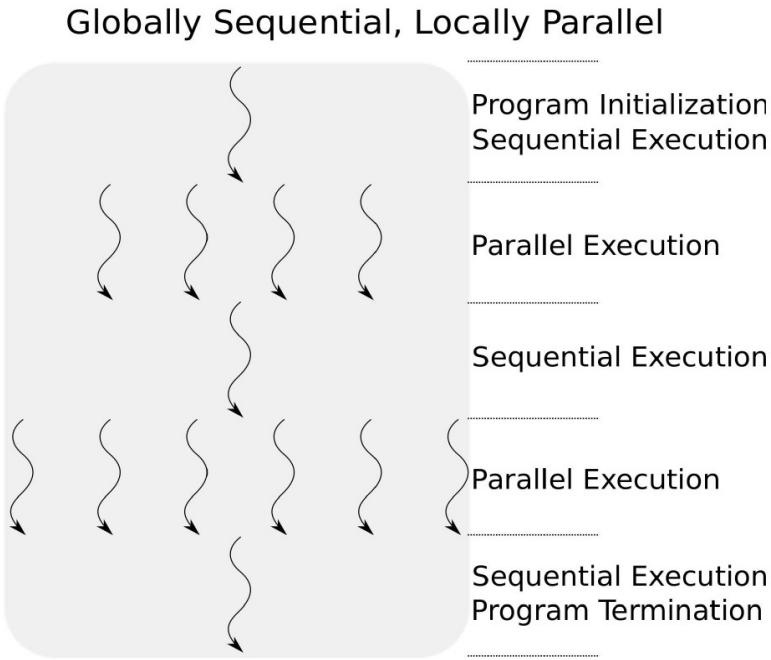
- The decomposition of a sequential program into components that can execute in parallel is a tedious enterprise.
- OpenMP has been designed to alleviate much of the effort involved, by accommodating the incremental conversion of sequential programs into parallel ones, with the assistance of the compiler.
- 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.4.0 was published in 2013.
- Standard controlled by the OpenMP Architecture Review Board (ARB).
- GNU C support:
  - GCC 4.7 supports OpenMP 3.1 specification
  - GCC 4.9 supports OpenMP 4.0.

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

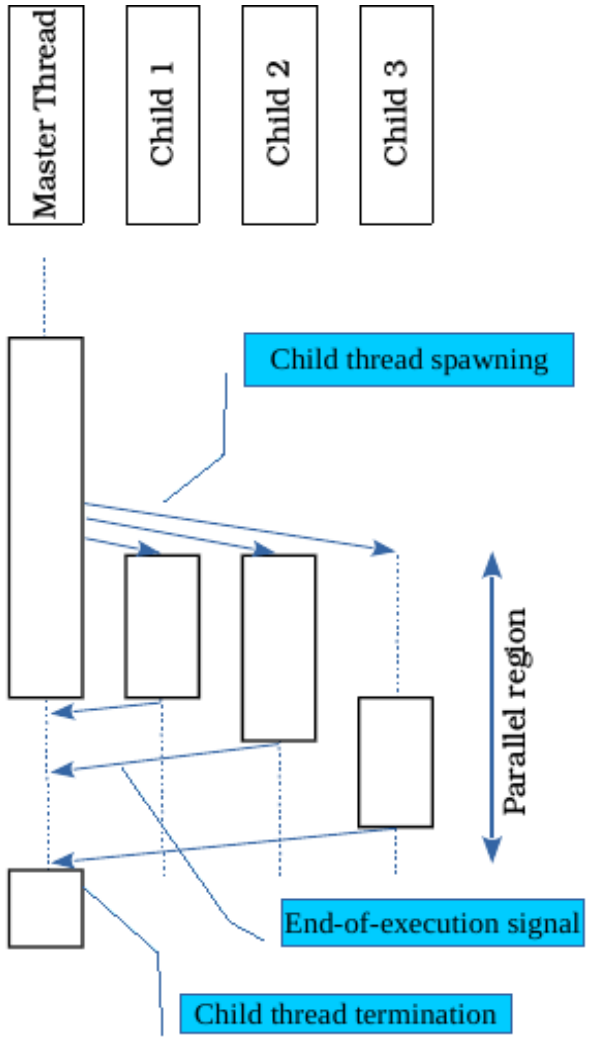
```
#include <iostream>
#include <stdlib.h>
#include <omp.h>

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 environmental 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_number_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

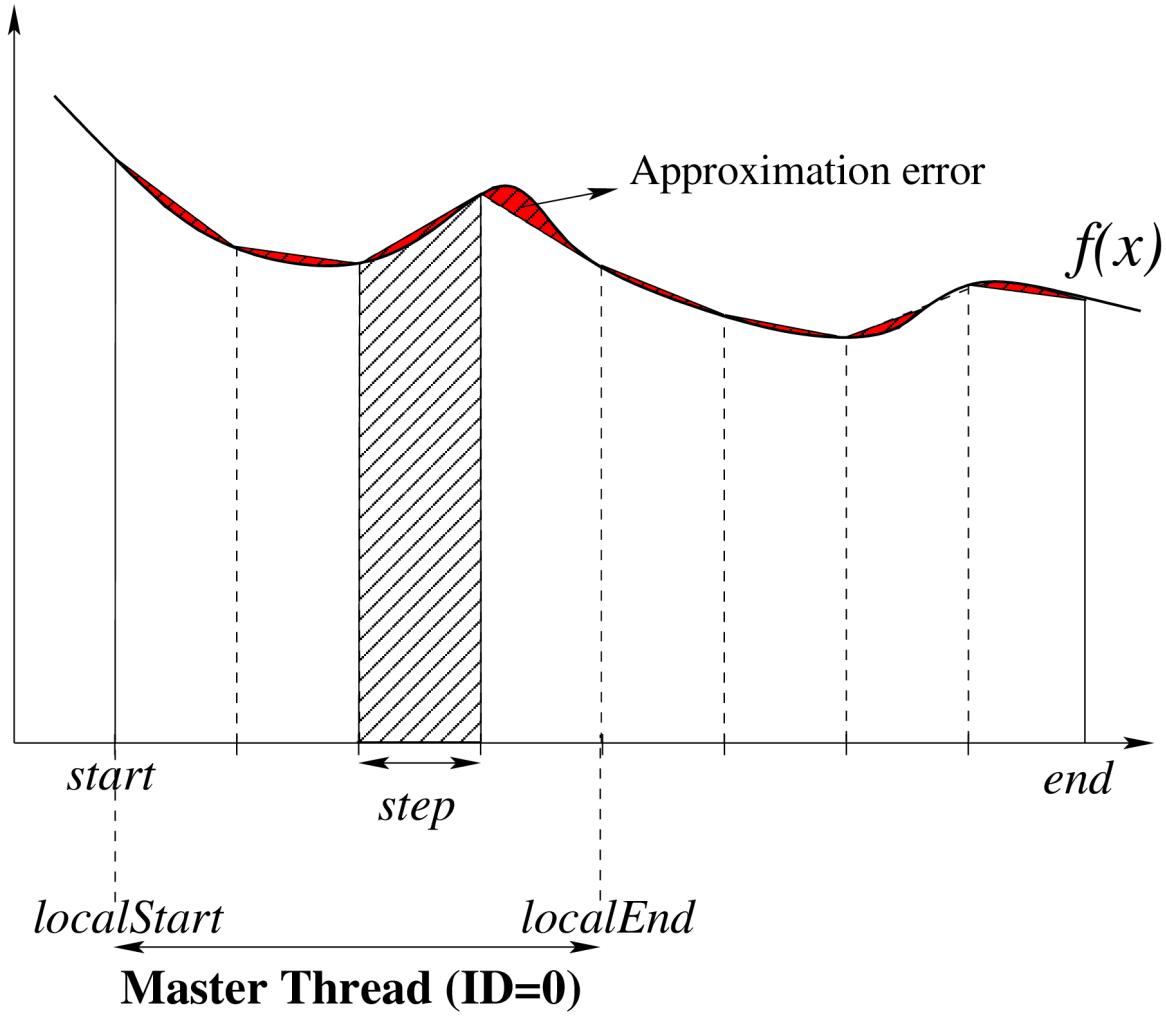
- Outside the parallel regions, normal scope rules apply.
- OpenMP specifies the following types of variables:
  - **Shared** : all variables declared outside a parallel region are **by default** shared. That does not mean that they are in anyway "protected".
  - **Private** : all variables declared 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;
```

Race condition!

OpenMP V.1 : Removing the race cond.

- 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
$+, -, \|, \|\|, \hat{}$	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.

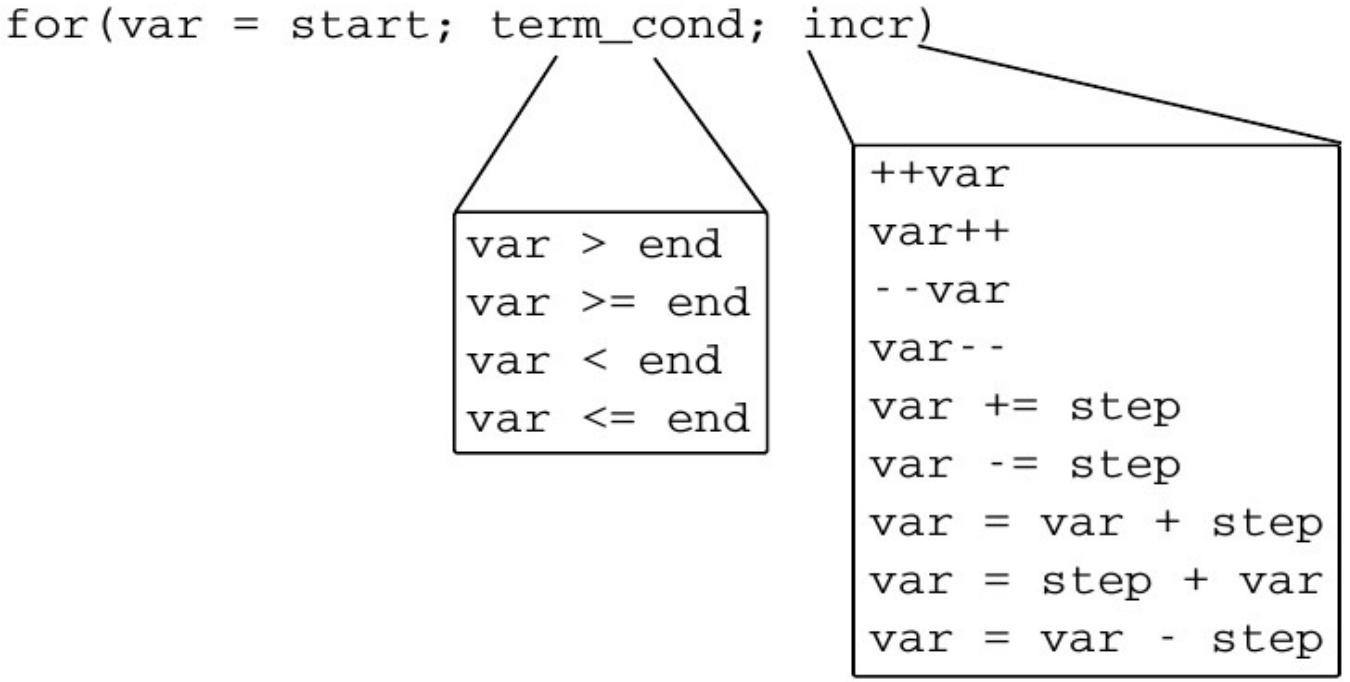
Loop Level Parallelism

- A for-loop has to satisfy certain conditions, which in OpenMP jargon 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:



- `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 implication 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 of 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 $S1 \ \delta^\alpha \ S2$

```
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 `reduce` clause.
- Induction variables : use affine function directly.

```
double v = start;
double sum=0;

#pragma omp parallel for reduction(+ : sum)
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.

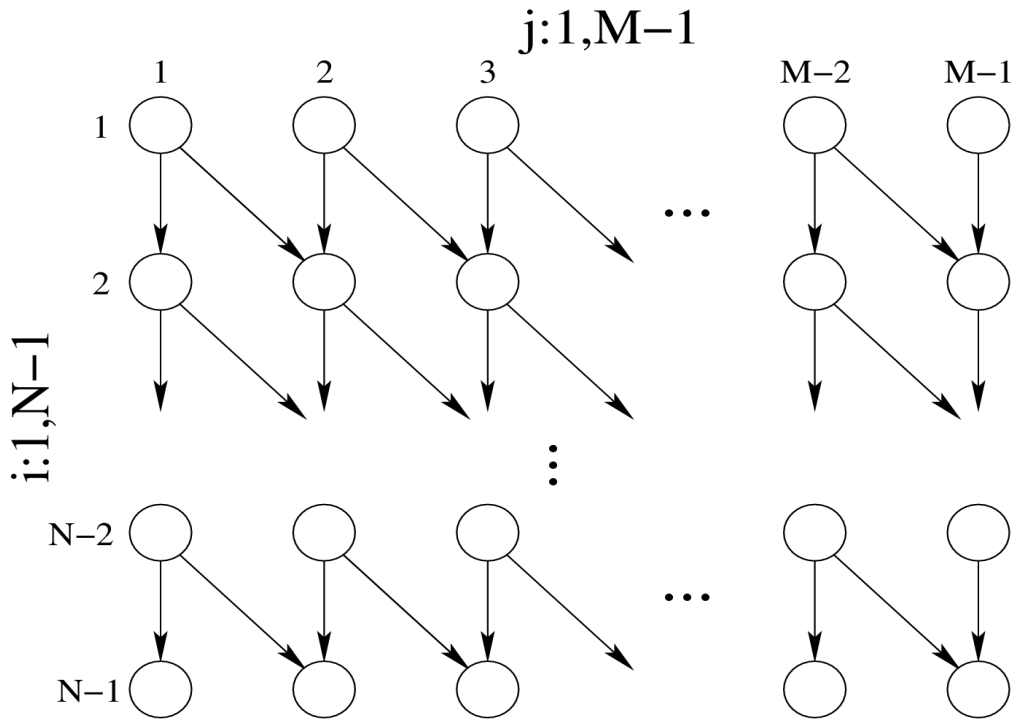
Flow Dependence : Loop Skewing (2)

```
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 an 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

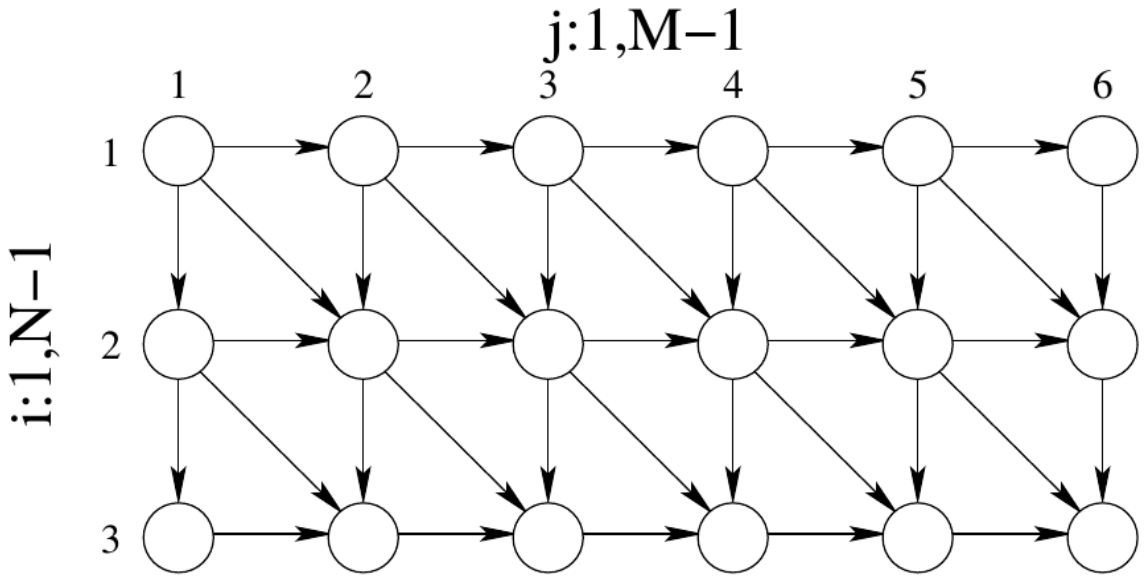
- In the previous example, the j-loop can be parallelized, but the i-loop has to be run sequentially.

```
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

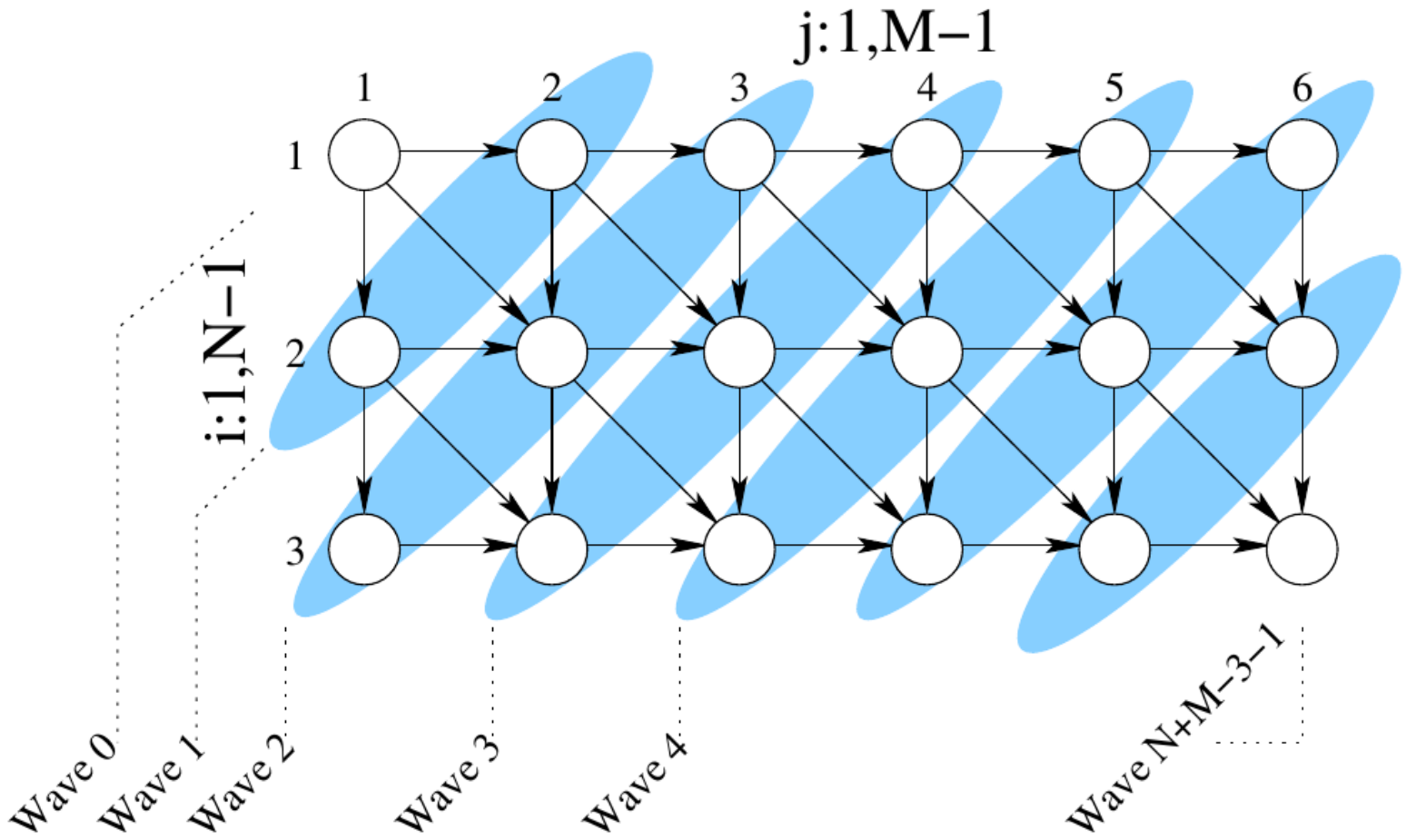
- Refactoring refers to rewriting of the loop(s) so that parallelism can be exposed.
- The ISDG for the following example:

```
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 reduction!

Flow Dependencies : Algorithm Change

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

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

Antidependecies

- 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];
      }
```

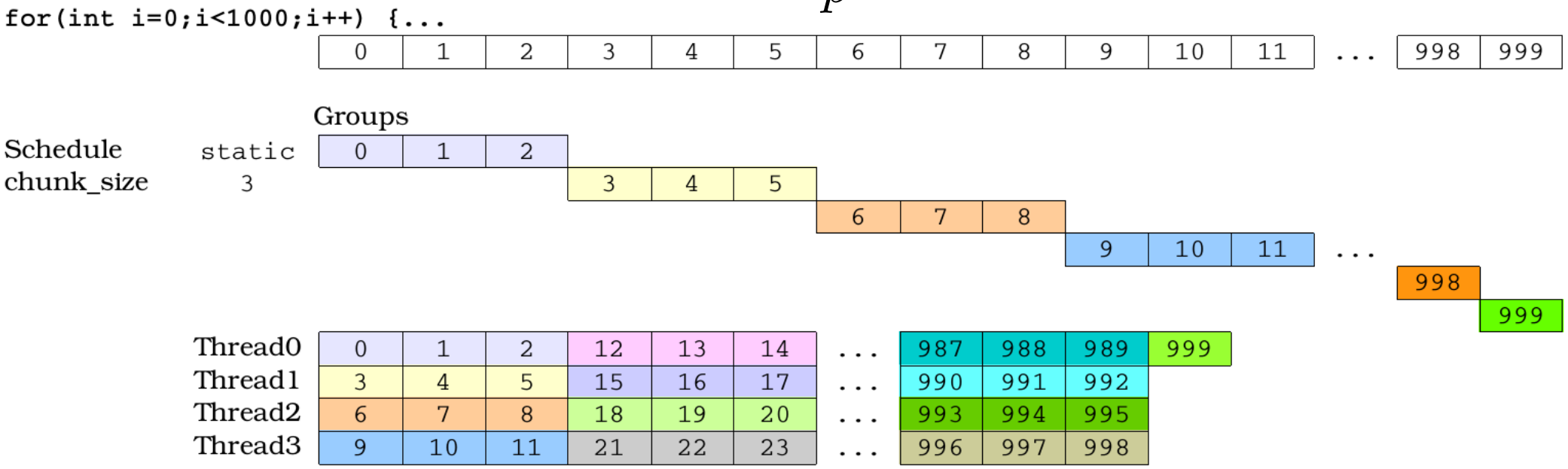
- **Q.** : could we do a modification that would allow `collapse(3)`?

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.

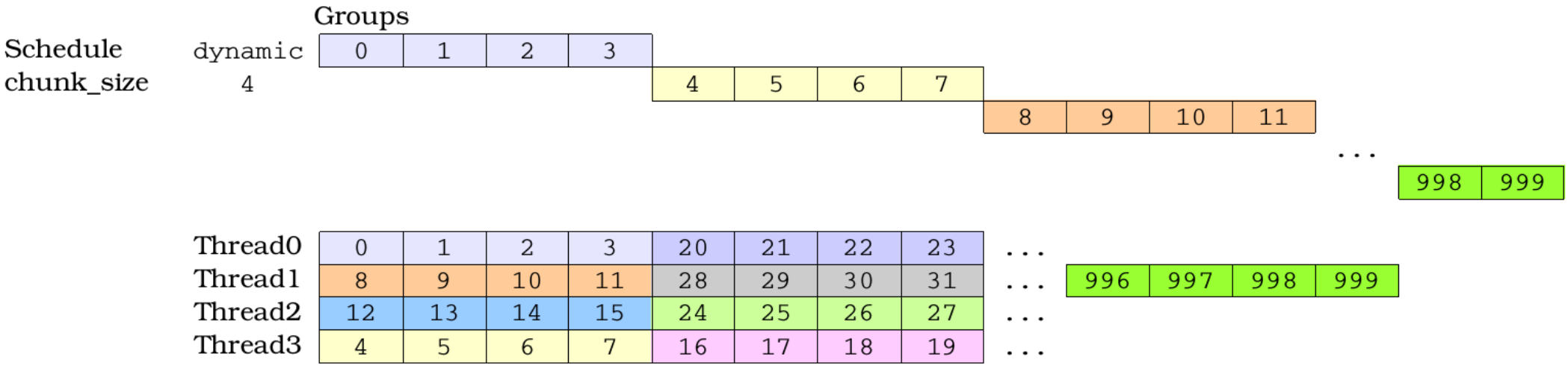
`static` schedule

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



`dynamic` schedule

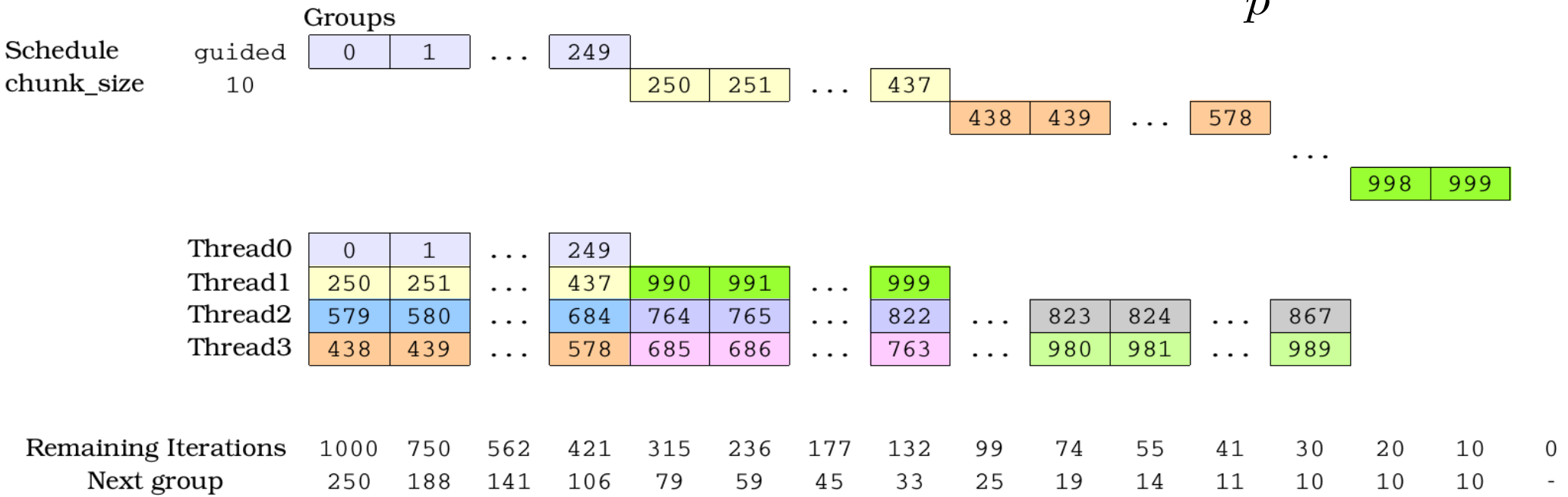
- *N* iterations are broken up into equal pieces of `chuck_size`, and assigned in a **first-come-first-served** basis to the *p* threads.
- Very good candidate for load balancing.
- But, it has a high coordination cost.



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 iterations is too high.
- If in doubt, set:

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

How to select a schedule option

- 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
```

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 Qt classes is more convenient.
- To combine Qt and OpenMP, one just has to add the following lines in a .pro file:

```
QMAKE_CXXFLAGS += -fopenmp
QMAKE_LFLAGS += -fopenmp
```

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

Integration using Prod.-Cons.

```
  Slice *buffer = new Slice[BUFFSIZE];
  int in = 0, out = 0;
  QSemaphore avail, buffSlots (BUFFSIZE);
  QMutex 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, QSemaphore &buffSlots, QSemaphore &↩
    avail, QMutex &l, int &out, QMutex &resLock, double &res)
{
  while (1)
    {
      avail.acquire ();          // wait for an available item
      l.lock ();
      int tmpOut = out;
      out = (out + 1) % BUFFSIZE;        // update the out index
      l.unlock ();

      // take the item out
      double st = buffer[tmpOut].start;
      double en = buffer[tmpOut].end;
      double div = buffer[tmpOut].divisions;

      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

- Complete code 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;
};
```

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

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

The `task` directive clauses

- The task directive can lead to the creation of too many tasks.
- **`if(scalar-expression)`** : if the expression evaluates to 0, the generated task becomes **undeferred**, i.e. the current task is suspended, until the generated task completes execution. The generated task may be executed by a different thread. An undeferred 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 undeferred 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 if clause, and 108 sec without!

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 using 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 until 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 )

    }
  }
```

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 function 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 2 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							…

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

Core 0 Cache

State	Address	Data							
	…								
S	0x8000	**x[0]**	x[1]	**x[2]**	x[3]	**x[4]**	x[5]	**x[6]**	x[7]
	…								

Core 1 Cache

State	Address	Data							
	…								
S	0x8000	x[0]	**x[1]**	x[2]	**x[3]**	x[4]	**x[5]**	x[6]	**x[7]**
	…								

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

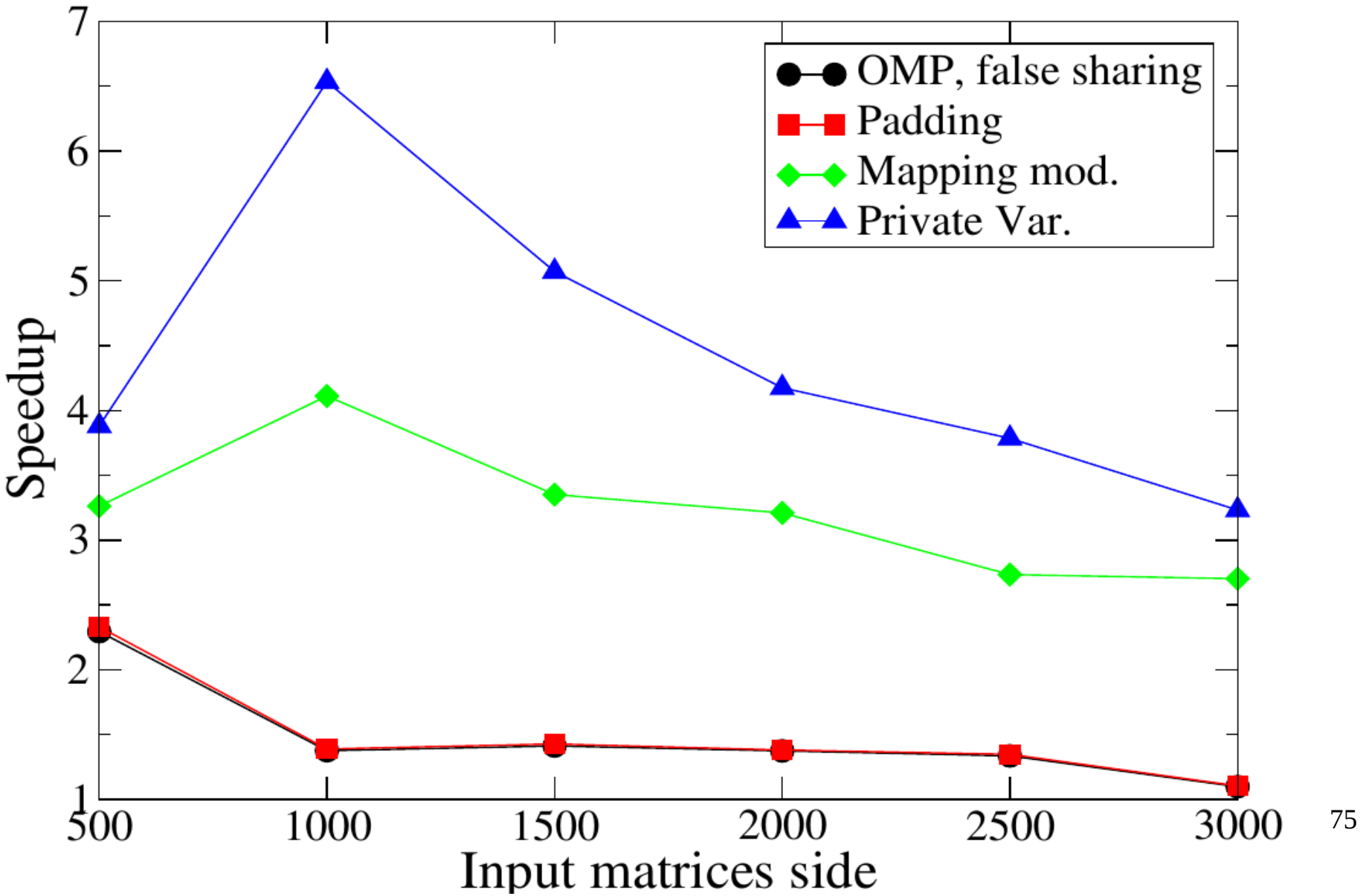
```
// assuming that N is a multiple of 8
double x[N];
#pragma omp parallel for schedule(static , 1)
   for( int i = 0; i < N; i += 8 )
   {
       double temp[ 8 ];
       for(int j = 0; j < 8; j++)
           temp[ j ] = someFunc( x [ i + j] );
       memcpy( x + i, temp, 8 * sizeof( double ));
  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

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
  T *temp = new T[N];
  // pointers to easily switch between the two arrays
  T *repo1, *repo2, *aux;

  repo1 = data;
  repo2 = temp;

  // loop for group size growing exponentially from 1 element to floor↩
       (lgN)
  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, sizeof (T) * N);

  delete[] temp;
}
```

- The two for loops cannot be collapsed. Why?

OpenMP Bottom-up mergesort

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

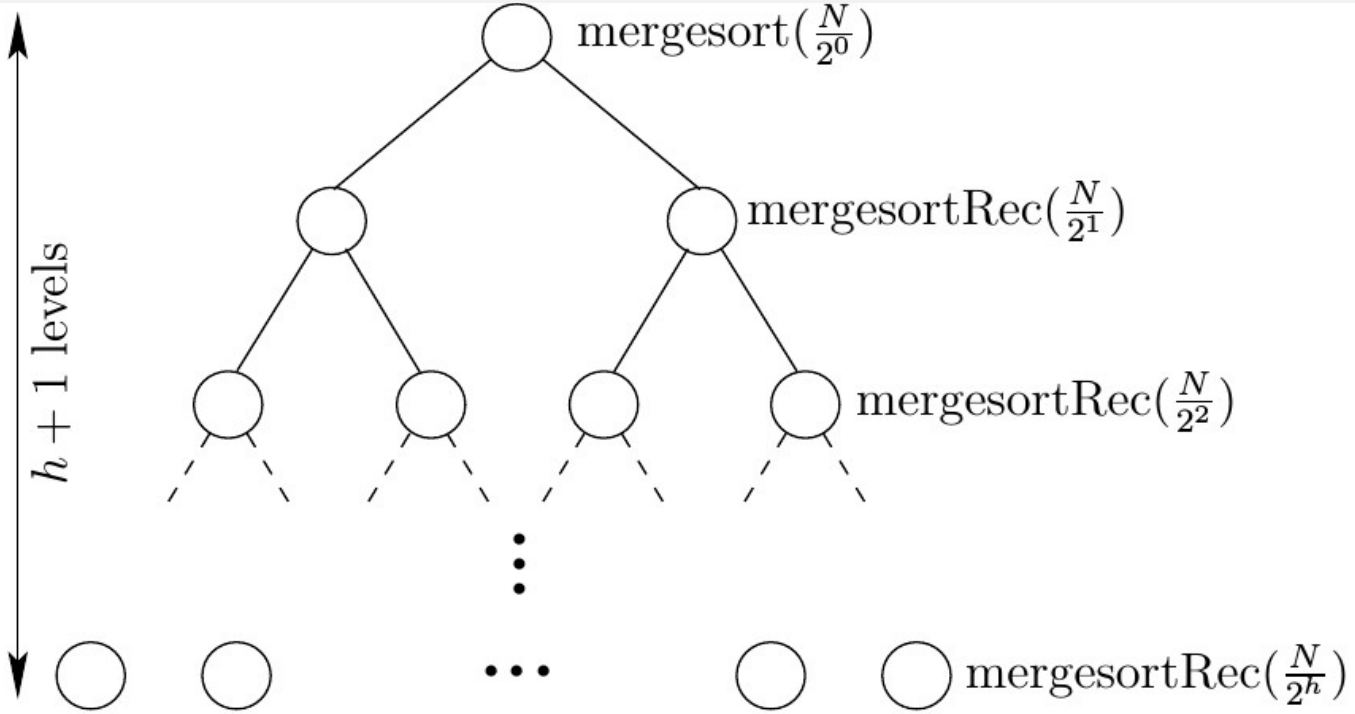
```
template < class T > void mergesort (T * data, int N)
{
  // allocate temporary array
  T *temp = new T[N];

#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;
}
```

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);
        }
. . .
```

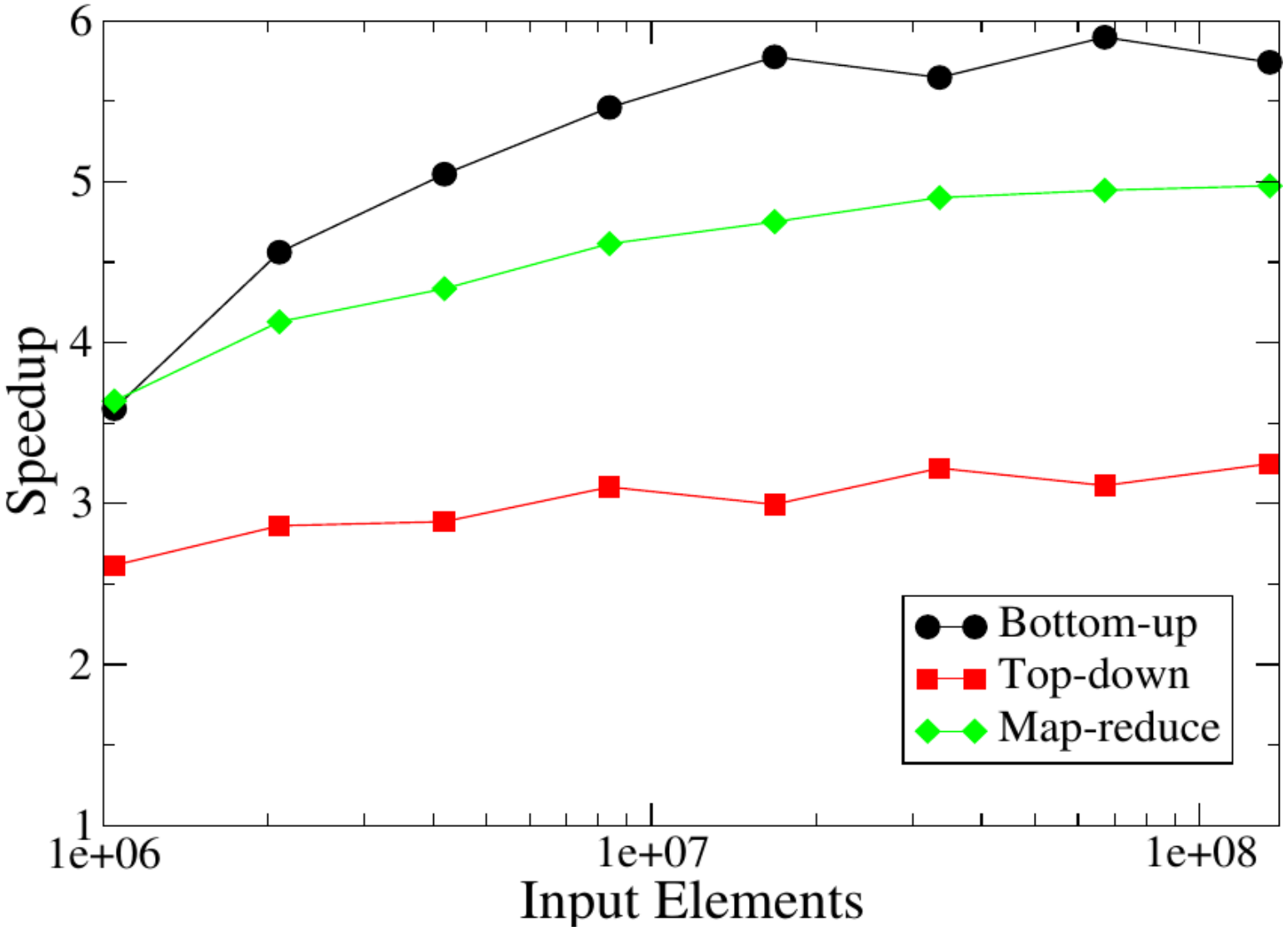


A more effective limit on the number of tasks

```
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


Speedup
6
5
4
3
2
1
1e+06
1e+07
1e+08
Input Elements
Bottom-up
Top-down
Map-reduce