SE 292: High Performance Computing [3:0][Aug:2014]

Shared Memory Parallelism using OpenMP

Yogesh Simmhan

Adapted from:

- *"MPI-Message Passing Interface", Sathish Vadhiyar, SE292 (Aug:2013),*
- *OpenMP C/C++ standard (openmp.org)*
- *OpenMP tutorial (http://www.llnl.gov/computing/tutorials/openMP/#Introduction)*
- *OpenMP sc99 tutorial presentation (openmp.org)*
- *Dr. Eric Strohmaier (University of Tennessee, CS594 class, Feb 9, 2000)*
- *An Introduction Into OpenMP, Ruud van der Pas © Sun Microsystems, 2005*
- *A "Hands-on" Introduction to OpenMP, Mattson & Meadows, SC08*

Introduction

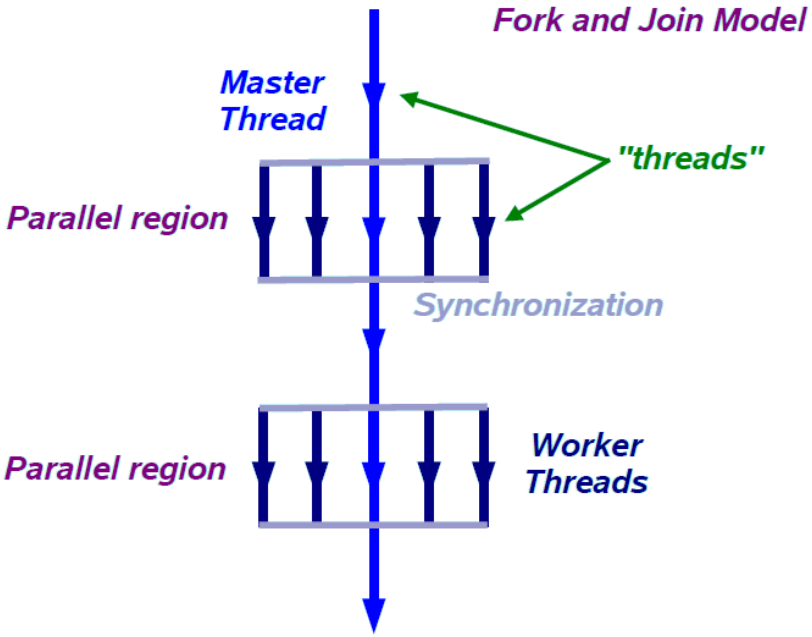
- An API for multi-threaded shared memory parallelism
- A specification for a set of compiler directives, library routines, and environment variables – standardizing pragmas
- Both fine-grain and coarse-grain parallelism
- Much easier to program than MPI

Introduction

- Mainly supports loop-level parallelism
- Follows fork-join model
- The number of threads can be varied from one region to another
- Based on compiler directives

Execution Model

- Begins as a single thread called master thread
- **Fork**: When **parallel** construct is encountered, team of threads are created
- Statements in the parallel region are executed in parallel
- Join: At the end of the parallel region, the team threads synchronize and terminate



Definitions

- Construct – statement containing directive and structured block
- Directive - `#pragma <omp id> <other text>`
  - Based on C #pragma directives

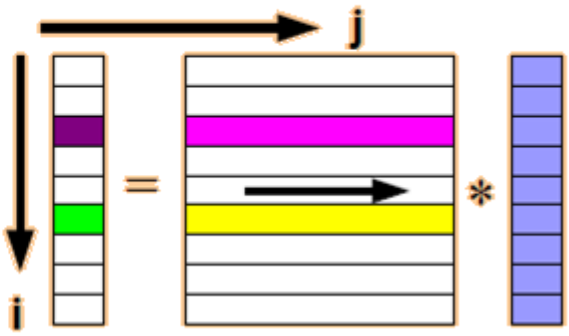
```
#pragma omp directive-name [clause [, clause] …] new-line
```

Example:

```
#pragma omp parallel default(shared) private(beta,pi)
```

E.g. Matrix Vector Product

```
#pragma omp parallel for default(none) \
                private(i,j,sum) shared(m,n,a,b,c)
for (i=0; i<m; i++)
{
    sum = 0.0;
    for (j=0; j<n; j++)
      sum += b[i][j]*c[j];
    a[i] = sum;
 }
```



Thread ID 0

```
for (i=0,1,2,3,4)
 i = 0
    sum = Σ b[i=0][j]*c[j]
    a[0] = sum
 i = 1
    sum = Σ b[i=1][j]*c[j]
    a[1] = sum
```

Thread ID 1

```
for (i=5,6,7,8,9)
 i = 5
    sum = Σ b[i=5][j]*c[j]
    a[5] = sum
 i = 6
    sum = Σ b[i=6][j]*c[j]
    a[6] = sum
```

parallel construct

```
#pragma omp parallel [clause [, clause] …] new-line
    structured-block
```

Clause:

```
if (scalar-expression)
private (variable-list)
firstprivate (variable-list)
default (shared | none)
shared (variable-list)
copyin (variable-list)
reduction (operator : variable-list)
num_threads (integer-expression)
```

Parallel construct

- Parallel region executed by multiple threads
- If num_threads, omp_set_num_threads(), OMP_SET_NUM_THREADS not used, then number of created threads is implementation dependent
- Number of physical processors hosting the thread also implementation dependent
- Threads numbered from 0 to N-1
- Nested parallelism by embedding one parallel construct inside another

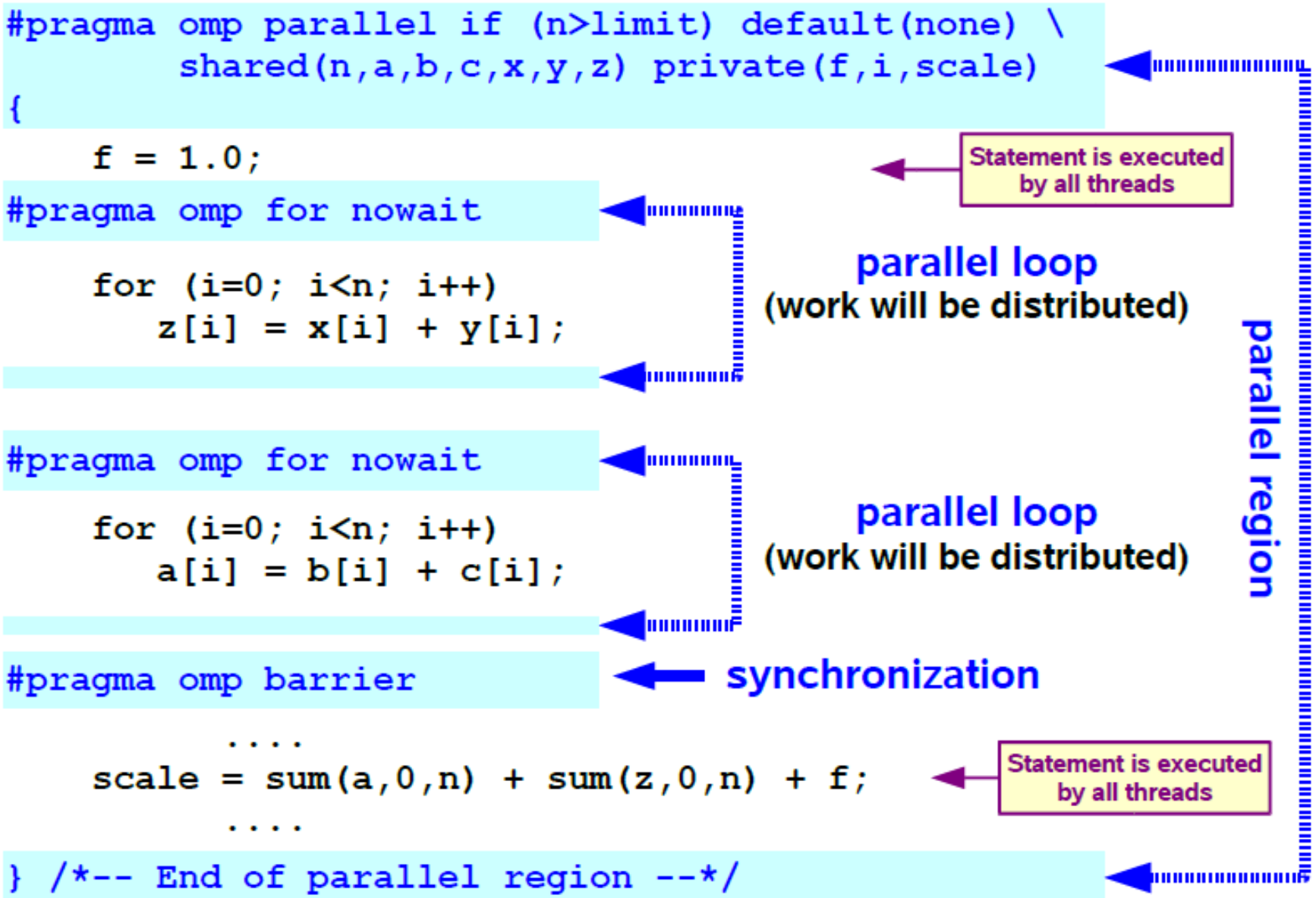
Parallel construct - Example

```
#include <omp.h>

main ()
{
  int nthreads, tid;

  #pragma omp parallel private(nthreads, tid)
   {
        printf("Hello World \n);
  }

}
```


#pragma omp parallel if (n>limit) default(none) \
shared(n,a,b,c,x,y,z) private(f,i,scale)
{
f = 1.0;
Statement is executed by all threads
#pragma omp for nowait
for (i=0; i<n; i++)
z[i] = x[i] + y[i];
parallel loop
(work will be distributed)
#pragma omp for nowait
for (i=0; i<n; i++)
a[i] = b[i] + c[i];
parallel loop
(work will be distributed)
parallel region
#pragma omp barrier
synchronization
....
scale = sum(a,0,n) + sum(z,0,n) + f;
Statement is executed by all threads
....
} /*-- End of parallel region --*/

When to use OpenMP?

- The compiler may not be able to do the parallelization in the way you like to see it:
- A loop is not parallelized
  - The data dependency analysis is not able to determine whether it is safe to parallelize or not
- The granularity is not high enough
  - The compiler lacks information to parallelize at the highest possible level
- This is when explicit parallelization through OpenMP directives and functions comes into the picture

Terminology

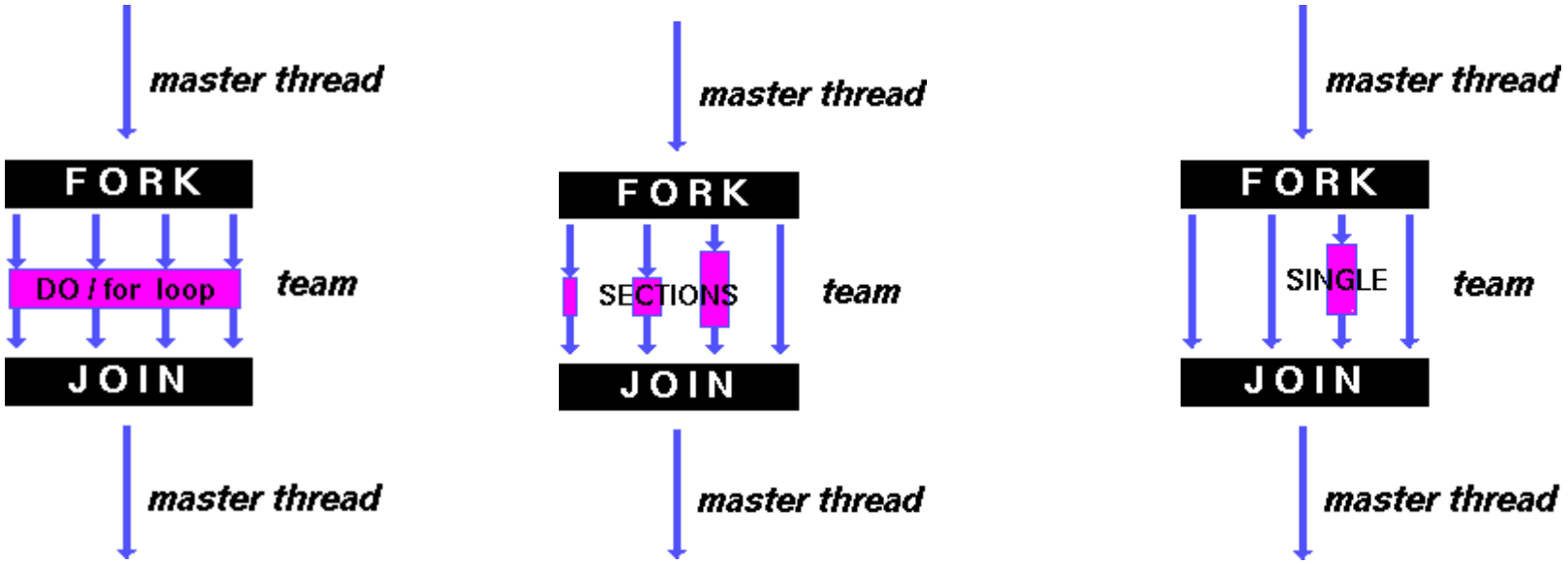
- OpenMP Team = Master + Workers
- A **Parallel Region** is a block of code executed by all threads simultaneously
  - The master thread always has thread ID 0
  - Thread adjustment (if enabled) is only done before entering a parallel region
  - Parallel regions can be nested, but support for this is implementation dependent
  - An "if" clause can be used to guard the parallel region; in case the condition evaluates to "false", the code is executed serially
- A work-sharing construct divides the execution of the enclosed code region among the members of the team; in other words: they split the work

Types of constructs, Calls, Variables

- Work-sharing constructs
- Synchronization constructs
- Data environment constructs
- Library calls, environment variables

Work sharing construct

- For distributing the execution among the threads that encounter it
- 3 types – `for, sections, single`



for construct

- For distributing the iterations among the threads

```
#pragma omp for [clause [, clause] …]
new-line
    for-loop
```

Clause:

```
private (variable-list)
firstprivate (variable-list)
lastprivate (variable-list)
reduction (operator : variable-list)
ordered
schedule (kind[, chunk_size])
nowait
```

for construct

- Restriction in the loop structure so the compiler can determine number of iterations
  - e.g. no branching out of loop
- The assignment of iterations to threads depend on the **schedule** clause
- Implicit barrier at the end of **for** if not **nowait**

schedule clause

1. `schedule(static, chunk_size)` – iterations/chunk_size chunks distributed in round-robin
2. `schedule(runtime)` – decision at runtime. Implementation dependent

for - Example

```
include <omp.h>
#define CHUNKSIZE 100
#define N 1000

main () {
int i, chunk; float a[N], b[N], c[N];

/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;

chunk = CHUNKSIZE;
#pragma omp parallel shared(a,b,c,chunk) private(i)
  {
  #pragma omp for schedule(dynamic,chunk) nowait
    for (i=0; i < N; i++)
      c[i] = a[i] + b[i];
  } /* end of parallel section */
}
```

Synchronization directives

```
#pragma omp master new-line
        structured-block

#pragma omp critical [(name)] new-line
        structured-block

#pragma omp barrier new-line

#pragma omp atomic new-line
     expression-stmt

#pragma omp flush [(variable-list)] new-line


#pragma omp ordered new-line
            structured-block
```

critical - Example

```
#include <omp.h>
main() {
int x;
x = 0;
#pragma omp parallel shared(x)
  {
  #pragma omp critical
        x = x + 1;
  }
}
```

atomic - Example

```
#pragma omp parallel for shared(x, y, index, n)
  for (i=0; i<n; i++) {
    #pragma omp atomic
      x[index[i]] += work1(i);
    y[i] += work2(i);
  }
```

Data Scope Attribute Clauses

Most variables are shared by default

Data scopes explicitly specified by data scope attribute clauses

Clauses:

1. private
2. firstprivate
3. lastprivate
4. shared
5. default
6. reduction
7. copyin
8. copyprivate

private, firstprivate & lastprivate

`private (variable-list)`

- variable-list private to each thread
- A new object with automatic storage duration allocated for the construct

`firstprivate (variable-list)`

- The new object is initialized with the value of the old object that existed prior to the construct

`lastprivate (variable-list)`

- The value of the private object corresponding to the last iteration or the last section is assigned to the original object

private - Example

```
int a;

void f(int n) {

a = 0;

    #pragma omp parallel for private(a)
    for (int i=1; i<n; i++) {

       a = i;
       g(i, n);
       d(a);      // Private copy of "a"
       ...
    }
...
}
void g(int k, int n) {

    h(k,a); //The global "a", not the private "a" in f
}
```

Private variables

- Private variables are undefined on entry and exit of the parallel region
- The value of the original variable (before the parallel region) is undefined after the parallel region !
- A private variable within a parallel region has no storage association with the same variable outside of the region
- Use the first/last private clause to override this behaviour

lastprivate - Example

```
main()
{
  A = 10;

#pragma omp parallel
{
  #pragma omp for private(i) firstprivate(A) lastprivate(B)...
  for (i=0; i<n; i++)
  {
      ....
      B = A + i;          /*-- A undefined, unless declared
                              firstprivate --*/
      ....
  }

  C = B;                  /*-- B undefined, unless declared
                              lastprivate --*/

} /*-- End of OpenMP parallel region --*/

}
```

shared, default, reduction

`shared(variable-list)`

`default(shared | none)`

- Specifies the sharing behavior of all of the variables visible in the construct
- **none**: No implicit defaults. Have to scope all variables explicitly.
- **shared**: All variables are shared. Default behaviour in absence of an explicit “default” clause.

`Reduction(op:variable-list)`

- Private copies of the variables are made for each thread
- The final object value at the end of the reduction will be combination of all the private object values
- `reduction (+:sum)`

default - Example

```
int x, y, z[1000];
#pragma omp threadprivate(x)

void fun(int a) {
  const int c = 1;
  int i = 0;

  #pragma omp parallel default(none) private(a) shared(z)
  {
     int j = omp_get_num_thread();
             //O.K.   - j is declared within parallel region
         a = z[j];

         x = c;

         z[i] = y;
```

Reading

- openmp.org
- Try out OpenMP samples on workstation/cluster

Library Routines (API)

- Querying function (number of threads etc.)
- General purpose locking routines
- Setting execution environment (dynamic threads, nested parallelism etc.)

API

- OMP_SET_NUM_THREADS(num_threads)
- OMP_GET_NUM_THREADS()
- OMP_GET_MAX_THREADS()
- OMP_GET_THREAD_NUM()
- OMP_GET_NUM_PROCS()
- OMP_IN_PARALLEL() … *in a parallel construct?*
- OMP_SET_DYNAMIC(dynamic_threads) … *Allow dynamic # of threads across parallel blocks*
- OMP_GET_DYNAMIC()
- OMP_SET_NESTED(nested)
- OMP_GET_NESTED()

Master & Single

- The **master** construct denotes a block should only be executed by the master thread. Other threads just skip it

```
#pragma omp parallel
{
  do_many_things();
  #pragma omp master
  { exchange_boundaries(); }
  #pragma omp barrier // Barrier has to be forced. Not default.
  do_many_other_things();
}
```

- The single construct denotes a block of code that is executed by only one thread (may not be master). A barrier is implied.

```
#pragma omp parallel
{
  do_many_things();
  #pragma omp single
  { exchange_boundaries(); } // Barrier is implied.
  do_many_other_things();
}
```

API(Contd..)

- omp_init_lock(omp_lock_t *lock)
- omp_init_nest_lock(omp_nest_lock_t *lock)
- omp_destroy_lock(omp_lock_t *lock)
- omp_destroy_nest_lock(omp_nest_lock_t *lock)
- omp_set_lock(omp_lock_t *lock)
- omp_set_nest_lock(omp_nest_lock_t *lock)
- omp_unset_lock(omp_lock_t *lock)
- omp_unset_nest__lock(omp_nest_lock_t *lock)
- omp_test_lock(omp_lock_t *lock)
- omp_test_nest_lock(omp_nest_lock_t *lock)

- omp_get_wtime()
- omp_get_wtick()

Lock details

- Simple locks and nestable locks
- Simple locks are not locked if they are already in a locked state
- Nestable locks can be locked multiple times by the same thread
- Simple locks are available if they are unlocked
- Nestable locks are available if they are unlocked or owned by a calling thread

Example – Lock functions

```
#include <omp.h>
int main()
{
  omp_lock_t lck;
  int id;

  omp_init_lock(&lck);
  #pragma omp parallel shared(lck) private(id)
  {
    id = omp_get_thread_num();

    omp_set_lock(&lck);
    printf("My thread id is %d.\n", id);
// only one thread at a time can execute this printf
    omp_unset_lock(&lck);

    while (! omp_test_lock(&lck)) {
      skip(id);   /* we do not yet have the lock,
                     so we must do something else */
    }
    work(id);      /* we now have the lock
                      and can do the work */
    omp_unset_lock(&lck);
  }
  omp_destroy_lock(&lck);
}
```

Example – Nested lock

```
#include <omp.h>
typedef struct {int a,b; omp_nest_lock_t lck;} pair;

void incr_a(pair *p, int a)
{
  // Called only from incr_pair, no need to lock.
  p->a += a;
}

void incr_b(pair *p, int b)
{
  // Called both from incr_pair and elsewhere,
  // so need a nestable lock.

  omp_set_nest_lock(&p->lck);
  p->b += b;
  omp_unset_nest_lock(&p->lck);
}
```

```
void incr_pair(pair *p, int a, int b)
{
  omp_set_nest_lock(&p->lck);
  incr_a(p, a);
  incr_b(p, b);
  omp_unset_nest_lock(&p->lck);
}

void f(pair *p)
{
  extern int work1(), work2(), work3();
  #pragma omp parallel sections
  {
    #pragma omp section
      incr_pair(p, work1(), work2());
    #pragma omp section
      incr_b(p, work3());
  }
}
```

Hybrid Programming – Combining MPI and OpenMP benefits

- MPI

  - explicit parallelism, no synchronization problems

  - suitable for coarse grain

- OpenMP

  - easy to program, dynamic scheduling allowed

  - only for shared memory, data synchronization problems

- MPI/OpenMP Hybrid

  - Can combine MPI data placement with OpenMP fine-grain parallelism

  - Suitable for cluster of SMPs (Clumps)

  - Can implement hierarchical model