Unit-4

Merge Sorted List

Many PRAM algorithms achieve low time complexity by performing more operations than an optimal RAM algorithm. For example, a RAM algorithm requires at most n-1 comparisons to merge two sorted lists of n/2 elements. The time complexity of merging two sorted list is O(n).

CREW PRAM algorithm is used for assigning each list element its own processor for n processors. The processor knows the index of the element in its own list. It finds the index in the other list using binary search. It adds the two indices to obtain the final position. The total number of operations increased to $O(n\log_2 n)$.

Example: Merging Two Sorted List

Consider two sorted arrays as

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
2	**6**	**8**	**14**	**18**	**20**	**24**

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
3	**5**	**9**	**12**	**13**	**22**	**25**

As mentioned in the algorithm, there is participation of all the processors because of the use of "spawn" function. We can illustrate this algorithm by choosing any processor value as P_3

Working of P_3

A[i = 3] is greater than [i – 1] => [3 – 1] => 2, i.e., 2nd position element in the lower array (A [9] is 2nd index position of lower array : 8 > 5

Now evaluate (high – n/2) because 8 is larger than two elements of lower array and larger than (high – n/2) => 9 – 7 => 2, i.e. 2 elements in the upper array.

Perform the binary search with A[3] in upper array and find the position as: high = index of the largest integer value which is less than 8 => high is 9

No, P3 can calculate the position of 8 in the merged array after computing: (i-1) + (high – n/2). So, the position is (i + high – n/2) => (3 + 9 – 7 = 5), i.e., 5th position.

A[1]	A[2]	A[3]	A[4]	A[5]	…	A[14]

				8		

Similarly, the process will run for all the position using individual processing element.

Matrix multiplication

Matrix multiplication is a binary operation of two matrices of order M1xN1 and M2xN2. The operation thus produces a matrix of order M1xN2 if and only if N1=M2. The result matrix of order M1xN2 is known as the matrix product. The parallel algorithm follows:

$$\begin{bmatrix} -2 & 1 \\ 0 & 4 \end{bmatrix} \times \begin{bmatrix} 6 & 5 \\ -7 & 1 \end{bmatrix} = \begin{bmatrix} -2\times6+1\times-7 & -2\times5+1\times1 \\ 0\times6+4\times-7 & 0\times5+4\times1 \end{bmatrix}$$

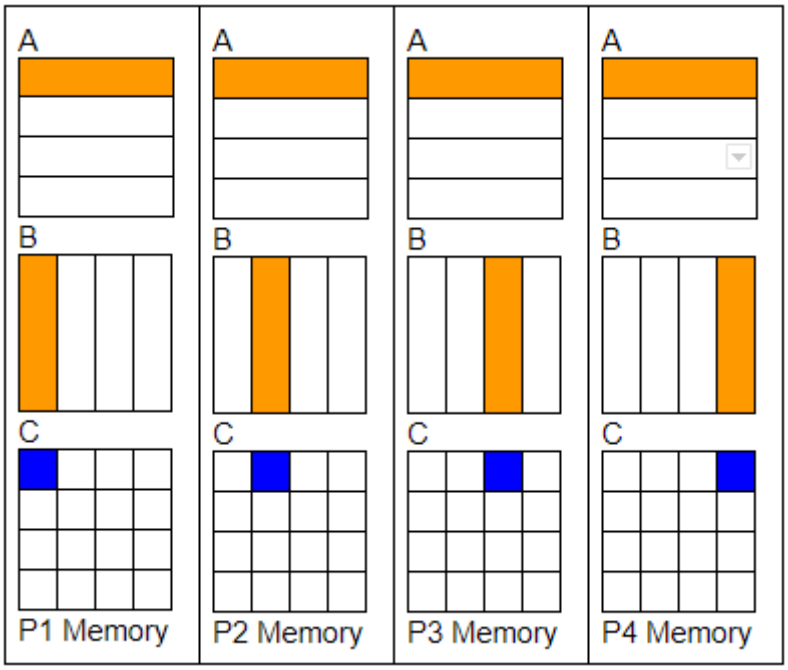
$$= \begin{bmatrix} -19 & -9 \\ -28 & 4 \end{bmatrix}$$

Matrix Multiplication for Big Data

The matrix multiplication becomes complex if you are dealing with big data. In case of Big Data, loading the complete matrix into the memory is not possible because of limited memory. In this case, we load just one row at a time from matrix A to all nodes (processors). The big data may have multiple dimensions. The matrix multiplication process has to be repeated for each dimension. The figure below depicts the big data two-dimensional multiplication. The algorithm for the same is given:

```
PARALLEL_MATRIX_MULTIPLICATION (A, B, C, M1, N1, M2, N2, DIM)

BEGIN
   for each dimension in DIM:
      for each row in A:
copy row to nodes
for each col in B:
   copy col to nodes
   Initialize
Crow col← 0
   for s = 1 to N1:
      Crow col ← Crow col + rows* cols
   end for
end for
      end for
   end for
```

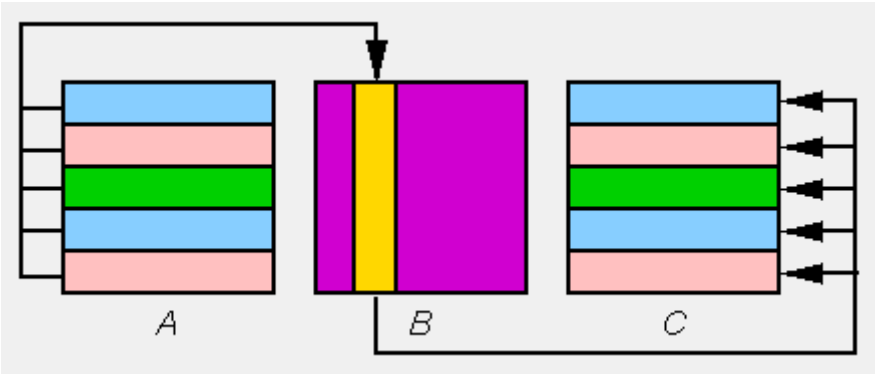
END



The row of matrix A is copied to all PEs along with different columns of matrix B which then calculates the one cell of C. On combining the result from all PEs we get a complete row of resulting matrix C. In this manner, we calculate for all rows. Also because we are calculating for n dimension matrix, the similar processing has to take place for each dimension repetitively. Suppose we have k PEs, then the number of PEs to be employed, k_e, will be calculated as:

$k_e = N_1 \ mod \ k$ where k>0. Broadcasting of each B[j] takes O(N logN) time.

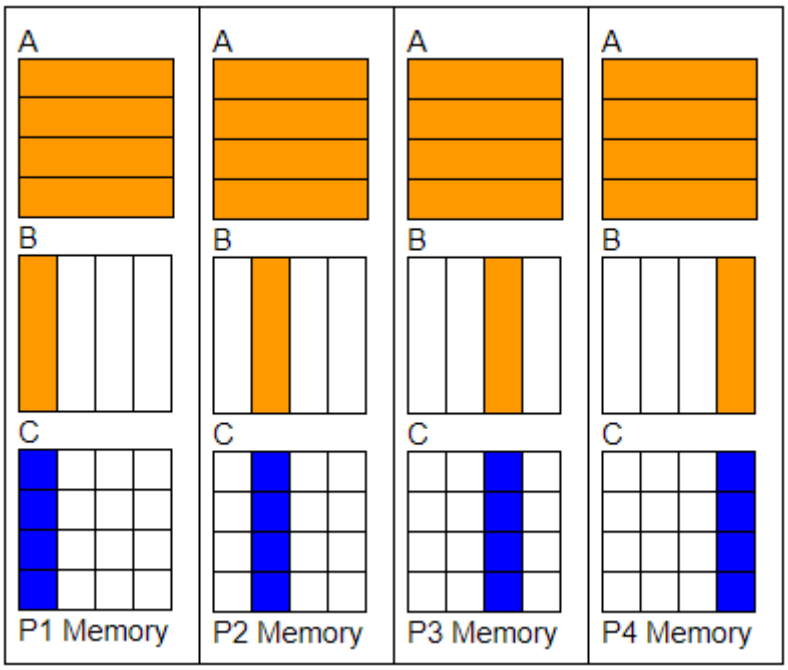
Row/ Column oriented algorithm



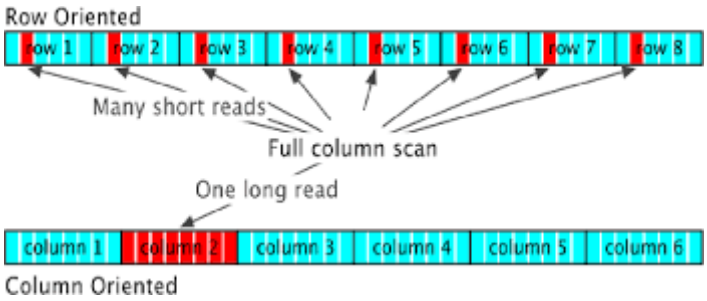
PARALLEL_MATRIX_MULTIPLICATION (A, B, C, M1, N1, M2, N2)

Do if N1=M2

```
Load A[i] on every processor P[i]

For all P[i] do:

   For j=0 to N-1:

      Load B[j]

      C[i][j] = A[i] * B[j]

Collect C[i]
```



Input Data | Output Data



Block oriented Matrix Multiplication

Block matrix multiplication is analogous to scalar matrix multiplication. In this approach sub-matrices are multiplied and shifted.

A=

A_{11}	A_{12}	...	A_{1n}
A_{21}	A_{22}		
...		...	
A_{m1}			A_{mn}

B=

B_{11}	B_{12}	...	B_{1n}
B_{21}	B_{22}		
...		...	
B_{m1}			B_{mn}

C=

$A_{11}\ B_{11} + A_{12}\ B_{21}$	$A_{11}\ B_{12} + A_{12}\ B_{22}$	...	B_{1n}
$A_{21}\ B_{11} + A_{22}\ B_{21}$	$A_{21}\ B_{12} + A_{22}\ B_{22}$		
...		...	
B_{m1}			B_{mn}

Analysis of Algorithm

n^2 matrix, p processors.

- Row-Column-oriented algorithm
    1. Computation: $n^2/p*n/p = n^3/p^2$.
    2. Communication: $2(lambda + beta\ n^2/p)$
    3. p iterations.
- Block-oriented algorithm
    1. Computation: $n^2/p*n/p = n^3/p^2$.
    2. Communication: $4(lambda + beta\ n^2/p)$
    3. $sqrt(p)-1$ iterations.
- Comparison
  $2p\ (lambda + beta\ n^2/p) > 4\ (sqrt(p)-1)\ (lambda + beta\ n^2/p)$
  $lambda\ p + beta\ n^2 > 4\ lambda\ (sqrt(p)-1) + 4\ beta\ (sqrt(p)-1)\ n^2/p$
    1. $p > 4\ (sqrt\ p-1) \rightarrow p > 4$
    2. $1 > 4\ (sqrt\ p-1)/p \rightarrow p > 4$

Solving linear systems

Let the system of equation to be solved be

AX = B

Decompose the coefficient matrix A as LU, where L is general lower triangular matrix and U is a unit upper triangular matrix with each of the leading diagonal elements equal to unity.

Using A = LU,

we get LUX = B

L^{-1} LUX = L^{-1} B

UX = C

Gaussian elimination

The Gaussian elimination stage consists in sequential elimination of the unknowns in the equations of the linear equation system being solved. Let us demonstrate the Gaussian elimination stage using the following system of linear equations as an example:

$$\begin{aligned} x_0 &+3x_1 &+2x_2 &= 1 \\ 2x_0 &+7x_1 &+5x_2 &= 18. \\ x_0 &+4x_1 &+6x_2 &= 26 \end{aligned}$$

At the first iteration the unknown x_0 is eliminated in the second and the third rows. For this the first row multiplied correspondingly by 2 and by 1 is subtracted from these rows. After these transformations the system looks as follows:

$$\begin{aligned} x_0 &+3x_1 &+2x_2 &= 1 \\ x_1 &+x_2 &= 16. \\ x_1 &+4x_2 &= 25 \end{aligned}$$

As a result, we need to perform the last iteration and eliminate the unknown x_1 in the third equation. For this it is necessary to subtract the second row. In the final form the system looks as follows:

$$\begin{aligned} x_0 &+3x_1 &+2x_2 &= 1 \\ x_1 &+x_2 &= 16. \\ &3x_2 &= 9 \end{aligned}$$

A serial version of the Gaussian elimination algorithm shown in Algorithm ?? consists of three nested loops. For each iteration of the outer loop, there is a division step and an elimination step. As the computation proceeds, only the lower-right k×k submatrix of A becomes active. Therefore, the amount of computation increases for elements in the direction of the lower-right corner of the matrix causing a non-uniform computational load. We assume that the matrix U shares storage with A and overwrites the upper-triangular portion of A. The elements ai,kcomputed is actually ui,k. Similarly, the elements ak,kequated is uk,k.

Algorithm 1: Sequential Gaussian elimination algorithm

```
1  for k ← 0 to n − 1 do
2      /* Division step */
3      for i ← k + 1 to n − 1 do
4          a[i][k] ← a[i][k]/a[k][k]
5      y[k] ← b[k]/a[k][k]
6      a[k][k] ← 1
7      /* Elimination step */
8      for i ← k + 1 to n − 1 do
9          for j ← k + 1 to n − 1 do
10             a[i][j] ← a[i][j] − a[i][k] ∗ a[k][j]
11         b[i] ← b[i] − a[i][k] ∗ y[k]
12         a[i][k] ← 0
```

The parallel algorithm for Gaussian Elimination

The parallel implementation of the Gauss method may be based on the data parallelization principle. All the computations can be reduced to uniform computational operations on the rows of coefficient matrix. All the computations related with processing a row of the matrix A and the corresponding element of the vector b may be taken as the basic computational subtask in this case.

Algorithm 4: (Pipe) OpenMP pipelining algorithm of LU form of GE

```
1  #pragma omp parallel private(k, i, j, row) shared(a)
2  {
3  long my_rank = omp_get_thread_num();
4  bsize = n/p;
5  if (my_rank != 0) then
6      for k ← (my_rank * bsize) to (my_rank * bsize) + bsize do
7          row = Get(my_rank);
8          Put(my_rank + 1, row);
9          /* Division step */
10         for i ← k to k + bsize do
11             a[i][row] = a[i][row]/a[row][row];
12         /* Elimination step */
13         for i ← k to k + bsize do
14             for j ← k to n + 1 do
15                 a[i][j] − = (a[i][row] * a[row][j]);
16 else
17     for k ← (my_rank * bsize) to (my_rank * bsize) + bsize do
18         Put(my_rank + 1, k);
19         /* Division step /*
20         for i ← k + 1 to k + bsize do
21             a[i][k] = a[i][k]/a[k][k];
22         /* Elimination step */
23         for i ← k + 1 to k + bsize do
24             for j ← k + 1 to n + 1 do
25                 a[i][j] − = (a[i][k] * a[k][j]);
26 }
```

Fig. illustrates the overall structure of the pipeline model. Each thread acts simultaneously as a producer and as a consumer. The thread consumes data from the left side of the channel and produces data to the right of the channel. The coordination of the threads of the pipeline stages can be used with the help of an array consisting of queues. Each element of the array refers to a queue of type buff_list and each queue has two pointers, a head and a tail of type record_s.

Therefore, each queue is implemented as a linked list. For a pipeline model with stages n, a data structure array buff[n] of type buff_list is used.

Jacobi algorithm

In numerical linear algebra, the Jacobi method is an iterative algorithm for determining the solutions of a strictly diagonally dominant system of linear equations. Let Ax=B, be a square system of n linear equations, where:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

Then A can be decomposed into a diagonal component D, a lower triangular part L and an upper triangular part U:

A = D + L + U

Where D =

$$\begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix}$$

And L + U =

$$\begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ a_{21} & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{bmatrix}.$$

The solution is then obtained iteratively via

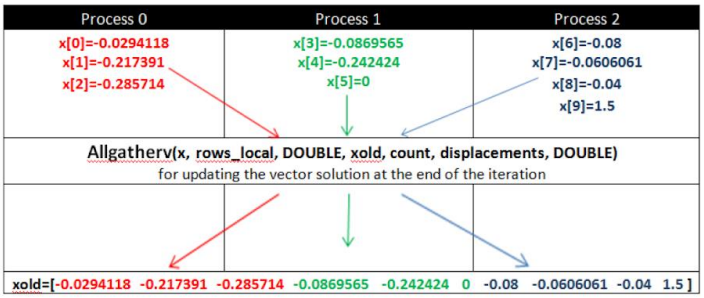
$$x^{(k+1)} = D^{-1}(b - (L + U)\, x^k$$

where x^k is the k^{th} approximation of x.

Hence the formula in terms of its elements can be given as:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right), \quad i = 1, 2, \ldots, n.$$

The parallel algorithm for Jacobi



This method assumes
we have all the input values of x in the previous iteration (k). But, usually all the x values are not given. Make an initial prepossession for x and to generate another group of solutions for x from the equation (1), which in fact
will represent the input values for the next iteration (k+1). After we have found the group of x values for the previous iteration we continue generating new groups again and again until we arrive at an acceptable solution. Jacobi puts borders between iterations; values of the vector-solution x are calculated only from the vector-solution of the previous iteration.

Based on Eq (1), we partition the problem as

D_1: $sum_1=0$

D_2: $sum_2=j=2naijxj$

D_3: $x_1 = (-sum_1 - sum_2 + b_1)/A_{11}$

D_4: $sum_1 = a_{11}x_1$

D_5: $sum_3=j=3naijxj$

D_6: $x_1 = (-sum_1 - sum_2 + b_2)/A_{22}$

D_7: $sum_1 = a_{11}x_1 + a_{12}x_2$

Likewise, we can decompose. On observation you will find that there are independent sub problems like 1, 2, 4, 5, 7, 8, etc. and dependent like 3 depends on 1 and 2, 6 depends on 4 and 5, and so on. A dependency graph can be constructed and strategy for parallelism can be made.

Process 0: sends the first three components of the vector solution x(0,1,2)

Process 1: sends the next coming three components of the vector solution x(3,4,5)

Repeat for k=0 to maxit

Initialize error_sum_local,sum1,sum2 ← 0.0

Repeat for j=0 to i_global

sum1 = sum1 + A[i][j]*xold[j]

Repeat for j=i_global+1 to N

sum2 = sum2 + A[i][j]*xold[j]

x[i] = (-sum1 - sum2 + b[i])/A[i][i_global]

error_sum_local += (x[i]-xold[i_global])*(x[i]-xold[i_global])

Computing $x^{(k+1)} := x^{(k)} + D^{-1}(b - Ax^{(k)})$ with p processors costs:

$$t_{comp} = \frac{n(2n+3)}{p}$$

The communication cost is:

$$t_{comm} = p\left(t_{startup} + \frac{n}{p}t_{data}\right)$$

CHAPTER 6 : Parallel Implementation

Parallel Computing

It is the use of multiple processing elements simultaneously for solving any problem. Problems are broken down into instructions and are solved concurrently as each resource which has been applied to work is working at the same time.

Types of Parallelism:

1. Bit-level parallelism: It is the form of parallel computing which is based on the increasing processor's size. It reduces the number of instructions that the system must execute in order to perform a task on large-sized data.
   Example: Consider a scenario where an 8-bit processor must compute the sum of two 16-bit integers. It must first sum up the 8 lower-order bits, then add the 8 higher-order bits, thus requiring two instructions to perform the operation. A 16-bit processor can perform the operation with just one instruction.

2. Instruction-level parallelism: A processor can only address less than one instruction for each clock cycle phase. These instructions can be re-ordered and grouped which are later on executed concurrently without affecting the result of the program. This is called instruction-level parallelism.

3. Task Parallelism: Task parallelism employs the decomposition of a task into subtasks and then allocating each of the subtasks for execution. The processors perform execution of sub tasks concurrently.

Applications of Parallel Computing:

- Data bases and Data mining.
- Real time simulation of systems.
- Science and Engineering.
- Advanced graphics, augmented reality and virtual reality.

Execution Model

OpenMP uses the fork-join model of parallel execution. Although this fork-join model can be useful for solving a variety of problems, it is somewhat tailored for large array-based applications. OpenMP is intended to support programs that will execute correctly both as parallel programs (multiple threads of execution and a full OpenMP support library) and as sequential programs (directives ignored and a simple OpenMP stubs library). However, it is possible and permitted to develop a program that does not behave correctly when executed sequentially. Furthermore, different degrees of parallelism may result in different numeric results because of changes in the association of numeric operations. For example, a serial addition reduction may have a different pattern of addition associations than a parallel reduction. These different associations may change the results of floating-point addition.

A program written with the OpenMP C/C++ API begins execution as a single thread of execution called the *master thread*. The master thread executes in a serial region until the first parallel construct is encountered. In the OpenMP C/C++ API, the parallel directive constitutes a parallel construct. When a parallel construct is encountered, the master thread creates a team of threads, and the master becomes master of the team. Each thread in the team executes the statements in the dynamic extent of a parallel region, except for the work-sharing constructs. Work-sharing constructs must be encountered by all threads in the team in the same order, and the statements within the associated structured block are executed by one or more of the threads. The barrier implied at the end of a work-sharing construct without a nowait clause is executed by all threads in the team. If a thread modifies a shared object, it affects not only its own execution environment, but also those of the other threads in the program. The modification is guaranteed to be complete, from the point of view of one of the other threads, at the next sequence point (as defined in the base language) only if the object is declared to be volatile. Otherwise, the modification is guaranteed to be complete after first the modifying thread, and then (or concurrently) the other threads, encounter a flush directive that specifies the object (either implicitly or explicitly). Note that when the flush directives that are implied by other OpenMP directives are not sufficient to ensure the desired ordering of side effects, it is the programmer's responsibility to supply additional, explicit flush directives. Upon completion of the parallel construct, the threads in the team synchronize at an implicit barrier, and only the master thread continues execution. Any number of parallel constructs can be specified in a single program. As a result, a program may fork and join many times during execution.

Memory Model

The OpenMP API provides a relaxed-consistency, shared-memory model. All OpenMP threads have access to a place to store and to retrieve variables, called the memory. In addition, each thread is allowed to have its own temporary view of the memory. The temporary view of memory for each thread is not a required part of the OpenMP memory model, but can represent any kind of intervening structure, such as machine registers, cache, or other local storage, between the thread and the memory. The temporary view of memory allows the thread to cache variables and thereby to avoid going to memory for every reference to a variable. Each thread also has access to another type of memory that must not be accessed by other threads, called thread private memory.

A single access to a variable may be implemented with multiple load or store instructions and, thus, is not guaranteed to be atomic with respect to other accesses to the same variable. Accesses to variables smaller than the implementation defined minimum size or to C or C++ bit-fields may be implemented by reading, modifying, and rewriting a larger unit of memory, and may thus interfere with updates of variables or fields in the same unit of memory.

If multiple threads write without synchronization to the same memory unit, including cases due to atomicity considerations as described above, and then a data race occurs. Similarly, if at least one thread reads from a memory unit and at least one thread writes without synchronization to that same

memory unit, including cases due to atomicity considerations as described above, and then a data race occurs. If a data race occurs then the result of the program is unspecified.

Directives

Directives are based on #pragma directives defined in the C and C++ standards. Compilers that support the OpenMP C and C++ API will include a command-line option that activates and allows interpretation of all OpenMP compiler directives.

Directive Format

The syntax of an OpenMP directive is formally specified by the grammar in Appendix C, and informally as follows:

#pragma omp *directive-name [clause[[,] clause]...] new-line*

Each directive starts with #pragma omp, to reduce the potential for conflict with other (non-OpenMP or vendor extensions to OpenMP) pragma directives with the same names. The remainder of the directive follows the conventions of the C and C++ standards for compiler directives. In particular, white space can be used before and after the #, and sometimes white space must be used to separate the words in a directive. Preprocessing tokens following the #pragma omp are subject to macro replacement. Directives are case-sensitive. The order in which clauses appear in directives is not significant. Clauses on directives may be repeated as needed, subject to the restrictions listed in the description of each clause. If *variable-list* appears in a clause, it must specify only variables. Only one *directive-name* can be specified per directive.

Conditional Compilation

The _OPENMP macro name is defined by OpenMP-compliant implementations as the decimal constant *yyyymm*, which will be the year and month of the approved specification. This macro must not be the subject of a #define or a #undef preprocessing directive.

#ifdef _OPENMP

iam = omp_get_thread_num() + index;

#endif

If vendors define extensions to OpenMP, they may specify additional predefined macros.

Parallel Construct

The following directive defines a parallel region, which is a region of the program that is to be executed by multiple threads in parallel. This is the fundamental construct that starts parallel execution.

#pragma omp parallel *[clause[[,]clause] ...] new-line*

structured-block

The *clause* is one of the following:

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*)

When a thread encounters a parallel construct, a team of threads is created if one of the following cases is true:

- No if clause is present.
- The if expression evaluates to a nonzero value.

This thread becomes the master thread of the team, with a thread number of 0, and all threads in the team, including the master thread, execute the region in parallel. If the value of the if expression is zero, the region is serialized.

To determine the number of threads that are requested, the following rules will be considered in order. The first rule whose condition is met will be applied:

1. If the num_threads clause is present, then the value of the integer expression is the number of threads requested.

2. If the omp_set_num_threads library function has been called, then the value of the argument in the most recently executed call is the number of threads requested.

3. If the environment variable OMP_NUM_THREADS is defined, then the value of this environment variable is the number of threads requested.

4. If none of the methods above were used, then the number of threads requested is implementation-defined.

If the num_threads clause is present then it supersedes the number of threads requested by the omp_set_num_threads library function or the OMP_NUM_THREADS environment variable only for the parallel region it is applied to. Subsequent parallel regions are not affected by it.

Work-sharing Constructs

A work-sharing construct distributes the execution of the associated statement among the members of the team that encounter it. The work-sharing directives do not launch new threads, and there is no implied barrier on entry to a work-sharing construct.

The sequence of work-sharing constructs and barrier directives encountered must be the same for every thread in a team. OpenMP defines the following work-sharing constructs, and these are described in the sections that follow:

- for directive
- sections directive
- single directive for Construct

The for directive identifies an iterative work-sharing construct that specifies that the iterations of the associated loop will be executed in parallel. The iterations of the for loop are distributed across threads that already exist in the team executing the parallel construct to which it binds. The syntax of the for construct is as follows:

#pragma omp for *[clause[[,] clause] ...] new-line*

for-loop

The clause is one of the following:

private(*variable-list*)

first private(*variable-list*)

last private(*variable-list*)

reduction(*operator*: *variable-list*)

ordered

schedule(*kind[*, *chunk size]*)

no wait

The for directive places restrictions on the structure of the corresponding for loop. Specifically, the corresponding for loop must have *canonical shape*:

for (*init-expr*; *var logical-op b*; *incr-expr*)

init-expr One of the following:

var = lb

I nteger-type var = lb

incr-expr One of the following:

++var

var++

--var

var--

var += incr

var -= incr

var = var + incr

var = incr + var

var = var - incr

var A signed integer variable. If this variable would otherwise be shared, it is implicitly made private for the duration of the for. This variable must not be modified within the body of the for statement. Unless the variable is specified lastprivate, its value after the loop is indeterminate.

logical-op One of the following:

<

<=

>

>=

lb, *b*, and *incr* Loop invariant integer expressions. There is no synchronization during the evaluation of these expressions. Thus, any evaluated side effects produce indeterminate results.

Restrictions to the for directive are as follows:

- The for loop must be a structured block, and, in addition, its execution must not be terminated by a break statement.
- The values of the loop control expressions of the for loop associated with a for directive must be the same for all the threads in the team.
- The for loop iteration variable must have a signed integer type.
- Only a single schedule clause can appear on a for directive.
- Only a single ordered clause can appear on a for directive.
- Only a single nowait clause can appear on a for directive.
- It is unspecified if or how often any side effects within the *chunk_size, lb, b*, or *incr* expressions occur.
- The value of the *chunk_size* expression must be the same for all threads in the team.

Sections Construct

The sections directive identifies a noniterative work-sharing construct that specifies a set of constructs that are to be divided among threads in a team. Each section is executed once by a thread in the team. The syntax of the sections directive is as follows:

#pragma omp sections *[clause[[,] clause] ...] new-line*

{

[#pragma omp section *new-line]*

structured-block

[#pragma omp section *new-line*

structured-block]

...

}

The clause is one of the following:

private(*variable-list*)

firstprivate(*variable-list*)

lastprivate(*variable-list*)

reduction(*operator*: *variable-list*)

nowait

Each section is preceded by a section directive, although the section directive is optional for the first section. The section directives must appear within the lexical extent of the sections directive. There is an implicit barrier at the end of a sections construct, unless a nowait is specified.

Restrictions to the sections directive are as follows:

- A section directive must not appear outside the lexical extent of the sections directive.
- Only a single nowait clause can appear on a sections directive.

Single Construct

The single directive identifies a construct that specifies that the associated structured block is executed by only one thread in the team (not necessarily the master thread). The syntax of the single directive is as follows:

#pragma omp single *[clause[[,] clause] ...] new-line*

structured-block

The clause is one of the following:

private(*variable-list*)

firstprivate(*variable-list*)

copyprivate(*variable-list*)

nowait

There is an implicit barrier after the single construct unless a nowait clause is specified.

Restrictions to the single directive are as follows:

- Only a single nowait clause can appear on a single directive.
- The copyprivate clause must not be used with the nowait clause.

Combined Parallel Work-sharing Constructs

Combined parallel work–sharing constructs are shortcuts for specifying a parallel region that contains only one work-sharing construct. The semantics of these directives are identical to that of

explicitly specifying a parallel directive followed by a single work-sharing construct. The following sections describe the combined parallel work-sharing constructs:

- The parallel for directive.
- The parallel sections directive.

Parallel for Construct

The parallel for directive is a shortcut for a parallel region that contains only a single for directive. The syntax of the parallel for directive is as follows:

#pragma omp parallel for *[clause[[,] clause] ...] new-line*

for-loop

This directive allows all the clauses of the parallel directive and the for directive, except the nowait clause, with identical meanings and restrictions. The semantics are identical to explicitly specifying a parallel directive immediately followed by a for directive.

Parallel sections Construct

The parallel sections directive provides a shortcut form for specifying a parallel region containing only a single sections directive. The semantics are identical to explicitly specifying a parallel directive immediately followed by a sections directive. The syntax of the parallel sections directive is as follows:

#pragma omp parallel sections *[clause[[,] clause] ...] new-line*

{

[#pragma omp section *new-line]*

structured-block

[#pragma omp section *new-line*

structured-block]

...

}

The *clause* can be one of the clauses accepted by the parallel and sections directives, except the nowait clause.

Master and Synchronization Directives

The following sections describe :

- the master construct.
- the critical construct.
- the barrier directive.
- the atomic construct.
- the flush directive.
- the ordered construct.

Master Construct

The master directive identifies a construct that specifies a structured block that is executed by the master thread of the team. The syntax of the master directive is as follows:

#pragma omp master *new-line*

structured-block

Other threads in the team do not execute the associated structured block. There is no implied barrier either on entry to or exit from the master constructs.

Critical Construct

The critical directive identifies a construct that restricts execution of the associated structured block to a single thread at a time. The syntax of the critical directive is as follows:

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

structured-block

An optional *name* may be used to identify the critical region. Identifiers used to identify a critical region have external linkage and are in a name space which is separate from the name spaces used by labels, tags, members, and ordinary identifiers.

A thread waits at the beginning of a critical region until no other thread is executing a critical region (anywhere in the program) with the same name. All unnamed critical directives map to the same unspecified name.

Barrier Directive

The barrier directive synchronizes all the threads in a team. When encountered, each thread in the team waits until all of the others have reached this point. The syntax of the barrier directive is as follows:

#pragma omp barrier *new-line*

After all threads in the team have encountered the barrier, each thread in the team begins executing the statements after the barrier directive in parallel.

Note that because the barrier directive does not have a C language statement as part of its syntax, there are some restrictions on its placement within a program. See Appendix C for the formal grammar. The example below illustrates these restrictions.

```
/* ERROR - The barrier directive cannot be the immediate

* substatement of an if statement

*/

if (x!=0)

#pragma omp barrier

...

/* OK - The barrier directive is enclosed in a

* compound statement.

*/

if (x!=0) {

#pragma omp barrier

}
```

Atomic Construct

The atomic directive ensures that a specific memory location is updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads. The syntax of the atomic directive is as follows:

#pragma omp atomic *new-line*

expression-stmt

The expression statement must have one of the following forms:

x binop= expr

x++

++*x*

x--

--*x*

In the preceding expressions:

- *x* is an lvalue expression with scalar type.
- *expr* is an expression with scalar type, and it does not reference the object designated by *x*.
- *binop* is not an overloaded operator and is one of +, *, -, /, &, ^, |, <<, or >>.

Restrictions to the atomic directive are as follows:

- All atomic references to the storage location *x* throughout the program are required to have a compatible type.

Flush Directive

The flush directive, whether explicit or implied, specifies a "cross-thread" sequence point at which the implementation is required to ensure that all threads in a team have a consistent view of certain objects (specified below) in memory. This means that previous evaluations of expressions that reference those objects are complete and subsequent evaluations have not yet begun. For example, compilers must restore the values of the objects from registers to memory, and hardware may need to flush write buffers to memory and reload the values of the objects from memory.

The syntax of the flush directive is as follows:

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

If the objects that require synchronization can all be designated by variables, then those variables can be specified in the optional *variable-list*. If a pointer is present in the *variable-list*, the pointer itself is flushed, not the object the pointer refers to. A flush directive without a *variable-list* synchronizes all shared objects except inaccessible objects with automatic storage duration. (This is likely to have more overhead than a flush with a *variable-list*.) A flush directive without a *variable-list*

is implied for the following directives:

- barrier
- At entry to and exit from critical
- At entry to and exit from ordered
- At entry to and exit from parallel
- At exit from for
- At exit from sections
- At exit from single

- At entry to and exit from parallel for
- At entry to and exit from parallel sections

The directive is not implied if a nowait clause is present. It should be noted that the flush directive is not implied for any of the following:

- At entry to for
- At entry to or exit from master
- At entry to sections
- At entry to single

A reference that accesses the value of an object with a volatile-qualified type behaves as if there were a flush directive specifying that object at the previous sequence point. A reference that modifies the value of an object with a volatile-qualified type behaves as if there were a flush directive specifying that object at the subsequent sequence point.

Note that because the flush directive does not have a C language statement as part of its syntax, there are some restrictions on its placement within a program. See Appendix C for the formal grammar. The example below illustrates these restrictions.

ERROR - The flush directive cannot be the immediate

* substatement of an if statement.

*/

if (x!=0)

#pragma omp flush (x)

...

/* OK - The flush directive is enclosed in a

* compound statement

*/

if (x!=0) {

#pragma omp flush (x)

}

Restrictions to the flush directive are as follows:

- A variable specified in a flush directive must not have a reference type.

Ordered Construct

The structured block following an ordered directive is executed in the order in which iterations would be executed in a sequential loop. The syntax of the ordered directive is as follows:

#pragma omp ordered *new-line*

structured-block

An ordered directive must be within the dynamic extent of a for or parallel for construct. In the execution of a for or parallel for construct with an ordered clause, ordered constructs are executed strictly in the order in which they would be executed in a sequential execution of the loop.

Restrictions to the ordered directive are as follows:

- An iteration of a loop with a for construct must not execute the same ordered directive more than once, and it must not execute more than one ordered directive.

Run-time Library Routines

The header <omp.h> declares two types, several functions that can be used to control and query the parallel execution environment, and lock functions that can be used to synchronize access to data.

The type omp_lock_t is an object type capable of representing that a lock is available, or that a thread owns a lock. These locks are referred to as *simple locks*.

The type omp_nest_lock_t is an object type capable of representing either that a lock is available, or both the identity of the thread that owns the lock and a *nesting count* (described below). These locks are referred to as *nestable locks*.

Execution Environment Functions

The functions described in this section affect and monitor threads, processors, and the parallel environment:

- the omp_set_num_threads function.
- the omp_get_num_threads function.
- the omp_get_max_threads function.
- the omp_get_thread_num function.
- the omp_get_num_procs function.
- the omp_in_parallel function.

- the omp_set_dynamic function.
- the omp_get_dynamic function.
- the omp_set_nested function.
- the omp_get_nested function

omp_set_num_threads Function

The omp_set_num_threads function sets the default number of threads to use for subsequent parallel regions that do not specify a num_threads clause. The format is as follows:

```
#include <omp.h>

void omp_set_num_threads(int num_threads);
```

The value of the parameter *num_threads* must be a positive integer. Its effect depends upon whether dynamic adjustment of the number of threads is enabled. For a comprehensive set of rules about the interaction between the omp_set_num_threads function and dynamic adjustment of threads.

This function has the effects described above when called from a portion of the program where the omp_in_parallel function returns zero. If it is called from a portion of the program where the omp_in_parallel function returns a nonzero value, the behavior of this function is undefined.

This call has precedence over the OMP_NUM_THREADS environment variable. The default value for the number of threads, which may be established by calling omp_set_num_threads or by setting the OMP_NUM_THREADS environment variable, can be explicitly overridden on a single parallel directive by specifying the num_threads clause.

omp_get_num_threads Function

The omp_get_num_threads function returns the number of threads currently in the team executing the parallel region from which it is called. The format is as follows:

```
#include <omp.h>

int omp_get_num_threads(void);
```

The num_threads clause, the omp_set_num_threads function, and the OMP_NUM_THREADS environment variable control the number of threads in a team. If the number of threads has not been explicitly set by the user, the default is implementation-defined. This function binds to the closest enclosing parallel directive.

omp_get_max_threads Function

The omp_get_max_threads function returns an integer that is guaranteed to be at least as large as the number of threads that would be used to form a team if a parallel region without a num_threads clause were to be encountered at that point in the code. The format is as follows:

#include <omp.h>

int omp_get_max_threads(void);

The following expresses a lower bound on the value of omp_get_max_threads

: *threads-used-for-next-team* <= omp_get_max_threads

Note that if a subsequent parallel region uses the num_threads clause to request a specific number of threads, the guarantee on the lower bound of the result ofomp_get_max_threads no long holds.

The omp_get_max_threads function's return value can be used to dynamically allocate sufficient storage for all threads in the team formed at the subsequent parallel region.

omp_get_thread_num Function

The omp_get_thread_num function returns the thread number, within its team, of the thread executing the function. The thread number lies between 0 and omp_get_num_threads()–1, inclusive. The master thread of the team is thread 0. The format is as follows:

#include <omp.h>

int omp_get_thread_num(void);

If called from a serial region, omp_get_thread_num returns 0. If called from within a nested parallel region that is serialized, this function returns 0.

omp_get_num_procs Function

The omp_get_num_procs function returns the number of processors that are available to the program at the time the function is called. The format is as follows:

#include <omp.h>

int omp_get_num_procs(void);

omp_in_parallel Function

The omp_in_parallel function returns a nonzero value if it is called within the dynamic extent of a parallel region executing in parallel; otherwise, it returns 0. The format is as follows:

#include <omp.h>

int omp_in_parallel(void);

This function returns a nonzero value when called from within a region executing in parallel, including nested regions that are serialized.

omp_set_dynamic Function

The omp_set_dynamic function enables or disables dynamic adjustment of the number of threads available for execution of parallel regions. The format is as follows:

#include <omp.h>

void omp_set_dynamic(int *dynamic_threads*);

If *dynamic_threads* evaluates to a nonzero value, the number of threads that are used for executing subsequent parallel regions may be adjusted automatically by the runtime environment to best utilize system resources. As a consequence, the number of threads specified by the user is the maximum thread count. The number of threads in the team executing a parallel region remains fixed for the duration of that parallel region and is reported by the omp_get_num_threads function.

If *dynamic_threads* evaluates to 0, dynamic adjustment is disabled.

This function has the effects described above when called from a portion of the program where the omp_in_parallel function returns zero. If it is called from a portion of the program where the omp_in_parallel function returns a nonzero value, the behavior of this function is undefined.

A call to omp_set_dynamic has precedence over the OMP_DYNAMIC environment variable.

omp_get_dynamic Function

The omp_get_dynamic function returns a nonzero value if dynamic adjustment of threads is enabled, and returns 0 otherwise. The format is as follows:

#include <omp.h>

int omp_get_dynamic(void);

If the implementation does not implement dynamic adjustment of the number of threads, this function always returns 0.

omp_set_nested Function

The omp_set_nested function enables or disables nested parallelism. The format is as follows:

#include <omp.h>

void omp_set_nested(int *nested*);

If *nested* evaluates to 0, nested parallelism is disabled, which is the default, and nested parallel regions are serialized and executed by the current thread. If *nested* evaluates to a nonzero value,

nested parallelism is enabled, and parallel regions that are nested may deploy additional threads to form nested teams.

This function has the effects described above when called from a portion of the program where the omp_in_parallel function returns zero. If it is called from a portion of the program where the omp_in_parallel function returns a nonzero value, the behavior of this function is undefined.

This call has precedence over the OMP_NESTED environment variable.

omp_get_nested Function

The omp_get_nested function returns a nonzero value if nested parallelism is enabled and 0 if it is disabled. The format is as follows:

#include <omp.h>

int omp_get_nested(void);

If an implementation does not implement nested parallelism, this function always returns 0.

Lock Functions

The function locks used for synchronization. For the following functions, the lock variable must have type omp_lock_t. This variable must only be accessed through these functions. All lock functions require an argument that has a pointer to omp_lock_t type.

- The omp_init_lock function initializes a simple lock.
- The omp_destroy_lock function removes a simple lock.
- The omp_set_lock function waits until a simple lock is available.
- The omp_unset_lock function releases a simple lock.
- The omp_test_lock function tests a simple lock.

For the following functions, the lock variable must have type omp_nest_lock_t.

This variable must only be accessed through these functions. All nestable lock functions require an argument that has a pointer to omp_nest_lock_t type.

- The omp_init_nest_lock function initializes a nestable lock.
- The omp_destroy_nest_lock function removes a nestable lock.
- The omp_set_nest_lock function waits until a nestable lock is available.
- The omp_unset_nest_lock function releases a nestable lock.

- The omp_test_nest_lock function tests a nestable lock.

The OpenMP lock functions access the lock variable in such a way that they always read and update the most current value of the lock variable. Therefore, it is not necessary for an OpenMP program to include explicit flush directives to ensure that the lock variable's value is consistent among different threads. (There may be a need for flush directives to make the values of other variables consistent.)

omp_init_lock and omp_init_nest_lock Functions

These functions provide the only means of initializing a lock. Each function initializes the lock associated with the parameter *lock* for use in subsequent calls. The format is as follows:

```
#include <omp.h>

void omp_init_lock(omp_lock_t *lock);

void omp_init_nest_lock(omp_nest_lock_t *lock);
```

The initial state is unlocked (that is, no thread owns the lock). For a nestable lock, the initial nesting count is zero. It is noncompliant to call either of these routines with a lock variable that has already been initialized.

omp_destroy_lock and omp_destroy_nest_lock Functions

These functions ensure that the pointed to lock variable *lock* is uninitialized. The format is as follows:

```
#include <omp.h>

void omp_destroy_lock(omp_lock_t *lock);

void omp_destroy_nest_lock(omp_nest_lock_t *lock);
```

It is noncompliant to call either of these routines with a lock variable that is uninitialized or unlocked.

omp_set_lock and omp_set_nest_lock Functions

Each of these functions blocks the thread executing the function until the specified lock is available and then sets the lock. A simple lock is available if it is unlocked. A nestable lock is available if it is unlocked or if it is already owned by the thread executing the function. The format is as follows:

```
#include <omp.h>

void omp_set_lock(omp_lock_t *lock);

void omp_set_nest_lock(omp_nest_lock_t *lock);
```

For a simple lock, the argument to the omp_set_lock function must point to an initialized lock variable. Ownership of the lock is granted to the thread executing the function.

For a nestable lock, the argument to the omp_set_nest_lock function must point to an initialized lock variable. The nesting count is incremented, and the thread is granted, or retains, ownership of the lock.

omp_unset_lock and omp_unset_nest_lock Functions

These functions provide the means of releasing ownership of a lock. The format is as follows:

```
#include <omp.h>

void omp_unset_lock(omp_lock_t *lock);

void omp_unset_nest_lock(omp_nest_lock_t *lock);
```

The argument to each of these functions must point to an initialized lock variable owned by the thread executing the function. The behavior is undefined if the thread does not own that lock.

For a simple lock, the omp_unset_lock function releases the thread executing the function from ownership of the lock.

For a nestable lock, the omp_unset_nest_lock function decrements the nesting count, and releases the thread executing the function from ownership of the lock if the resulting count is zero.

omp_test_lock and omp_test_nest_lock Functions

These functions attempt to set a lock but do not block execution of the thread. The format is as follows:

```
#include <omp.h>

int omp_test_lock(omp_lock_t *lock);

int omp_test_nest_lock(omp_nest_lock_t *lock);
```

The argument must point to an initialized lock variable. These functions attempt to set a lock in the same manner as omp_set_lock and omp_set_nest_lock, except that they do not block execution of the thread.

For a simple lock, the omp_test_lock function returns a nonzero value if the lock is successfully set; otherwise, it returns zero.

For a nestable lock, the omp_test_nest_lock function returns the new nesting count if the lock is successfully set; otherwise, it returns zero.

Timing Routines

The functions described in this section support a portable wall-clock timer:

- The omp_get_wtime function returns elapsed wall-clock time.
- The omp_get_wtick function returns seconds between successive clock ticks.

omp_get_wtime Function

The omp_get_wtime function returns a double-precision floating point value equal to the elapsed wall clock time in seconds since some “time in the past”. The actual “time in the past” is arbitrary, but it is guaranteed not to change during the execution of the application program. The format is as follows:

```
#include <omp.h>

double omp_get_wtime(void);
```

It is anticipated that the function will be used to measure elapsed times as shown in the following example:

```
double start;

double end;

start = omp_get_wtime();

... work to be timed ...

end = omp_get_wtime();

printf(“Work took %f sec. time.\n”, end-start);
```

The times returned are “per-thread times” by which is meant they are not required to be globally consistent across all the threads participating in an application.

omp_get_wtick Function

The omp_get_wtick function returns a double-precision floating point value equal to the number of seconds between successive clock ticks. The format is as follows:

```
#include <omp.h>

double omp_get_wtick(void);
```