CY TECH	**Architecture et Programmation Parallèle** **Examen Session Rattrapage**
Mariem Allouch Mahdi - Thierry Garcia - Mohamed Haddache - Juan Angel Lorenzo	
ING2-GSI-MI	Année 2022 - 2023

Modalités

- Durée : 2 heures.
- Vous devez rédiger votre copie à l'aide d'un stylo à encre exclusivement.
- Toutes vos affaires (sacs, vestes, trousse, etc.) doivent être placées à l'avant de la salle.
- Aucun document n'est autorisé sauf les feuilles MPI et OpenMP fournies avec ce sujet.
- Aucune question ne peut être posée aux enseignants, posez des hypothèses en cas de doute.
- Aucune sortie n'est autorisée avant une durée incompressible d'une heure.
- Aucun déplacement n'est autorisé.
- Aucun échange, de quelque nature que ce soit, n'est possible.

Exercice 1 : Questions de cours (5 points)

1. Que veut dire qu'OpenMP *est basé sur le modèle d'exécution fork-join* ? **(1 point)**.

2. Dans le code suivant, combien de threads seront en exécution dans la ligne 15 ? **(1 point)**.

```
1  # include <omp.h>
2  # include <stdio.h>
3
4  int main () {
5
6     omp_set_num_threads(8);
7     printf("How many threads (master): %d \n",omp_get_num_threads());
8
9     # pragma omp parallel num_threads(4)
10     {
11         int id = omp_get_thread_num();
12         printf("I am thread %d \n",id)  ;
13         printf("How many threads in the parallel region (%d): %d \n\n",id,
                omp_get_num_threads());
14     }
15
16     printf(" The master thread ends... \n");
17     return 0;
18 }
```

3. Dans quelle catégorie de la taxonomie de Flynn pouvons-nous inclure les GPUs ? Justifiez **(1 point)**.

4. Quelle est la différence entre les clauses "private" et "shared" en OpenMP ? **(1 point)**.

5. Qu'est-ce que la communication collective en MPI ? Donnez deux exemples de fonctions MPI utilisées pour la communication collective **(1 point)**.

Exercice 2 : Évaluation de la performance (5 points)

Nous souhaitons améliorer la performance d'un ordinateur en modifiant le répertoire d'instructions de sa CPU. Nous considérons les alternatives suivantes :

Type d'instruction à améliorer	**Percentage d'utilisation**	**Facteur d'amélioration atteignable**
Instructions d'addition	30 %	5
Instructions de saut conditionnel	39 %	4
Instructions de recherche et stockage	32 %	2
Reste d'instructions	4 %	7

Répondre aux questions suivantes:

a) Indiquer **et justifier** quelle alternative donne la meilleure amélioration de la performance de l'ordinateur **(2,5 points)**.
b) Si, avant l'amélioration, le temps d'exécution d'un programme était de 28.3 secondes, calculer combien sera le temps d'exécution après l'amélioration choisie dans la question précédente **(2,5 points)**.

Exercice 3 : OpenMP (5 points)

Considérons le code OpenMP suivant :

```
#include <stdio.h>
void work1() {}
void work2() {}

omp_set_num_threads(4);

void example()
{
   #pragma omp parallel
   {
         printf("Beginning work1.\n");
         work1();
        printf("Finishing work1.\n");
        printf("Finished work1 and beginning work2.\n");
        work2();
   }
}
```

Répondre aux questions suivantes :

a) Expliquer le fonctionnement de la fonction *example()* **(2 points).**
b) Modifiez cette fonction avec OpenMP pour que les trois messages *printf* ne s'affichent qu'une seule fois chacun **(1 point).**
c) Prise en compte qu'il n'y a pas de dépendances entre les fonctions *work1()* et *work2()*, modifiez le code de la section précédente pour qu'un thread puisse commencer à travailler sur la fonction *work2()* sans attendre que les messages *printf* s'affichent **(2 points)**.

Exercice 4 : MPI (5 points)

1. Nous cherchons à réaliser un "Ping-Pong" entre deux machines A et B. Dans ce cadre, on va réaliser un programme en langage C en plusieurs étapes **(2,5 points)** :

- la machine A (de rang 0) envoie un message contenant une série aléatoire de 1000 réels à la machine B (de rang 1),
- la machine B (de rang 1) renvoie ce message vers le processus A (de rang 0).
- Modifier le programme afin de faire un match de Ping-Pong, c'est-à-dire enchaîner 9 Ping-Pong, en faisant varier la taille du message.
- Comment mettriez-vous une mesure de temps pris par les **"communications"** à l'aide de la fonction MPI_Wtime() ?

2. On considère une matrice carrée réelle A, de taille n × n. On souhaite calculer, en parallèle et en utilisant MPI, la trace de la matrice A, avec Trace(A) = $\sum_{i=1}^{n} a_{ii}$.

Ecrire un programme MPI (en utilisant des communications collectives et de réductions) qui effectue les actions suivantes **(2,5 points)** :

- La machine de rang 0 initialise la matrice A et la distribue sur p machines. On considère cette distribution par lignes et on suppose que n est divisible par p,
- Chaque machine calcule la trace locale correspondant à sa portion de la matrice A,
- La machine de rang 0 récupère toutes les traces locales pour calculer la trace globale de la matrice.

OpenMP 4.5 API C/C++ Syntax Reference Guide

OpenMP Application Program Interface (API) is a portable, scalable model that gives parallel programmers a simple and flexible interface for developing portable parallel applications. OpenMP supports multi-platform shared-memory parallel programming in C/C++ and Fortran on all architectures, including Unix platforms and Windows platforms. See www.openmp.org for specifications.

- Text in this color indicates functionality that is new or changed in the OpenMP API 4.5 specification.
- [n.n.n] Refers to sections in the OpenMP API 4.5 specification.
- [n.n.n] Refers to sections in the OpenMP API 4.0 specification.

Directives and Constructs for C/C++

An OpenMP executable directive applies to the succeeding structured block or an OpenMP construct. Each directive starts with **#pragma omp**. The remainder of the directive follows the conventions of the C and C++ standards for compiler directives. A *structured-block* is a single statement or a compound statement with a single entry at the top and a single exit at the bottom.

parallel [2.5] [2.5]

Forms a team of threads and starts parallel execution.

#pragma omp parallel *[clause[[,]clause] ...]*
structured-block

clause:

if(*[* **parallel :** *]* *scalar-expression*)
num_threads(*integer-expression*)
default(**shared** | **none**)
private(*list*)
firstprivate(*list*)
shared(*list*)
copyin(*list*)
reduction(*reduction-identifier*: *list*)
proc_bind(**master** | **close** | **spread**)

for [2.7.1] [2.7.1]

Specifies that the iterations of associated loops will be executed in parallel by threads in the team in the context of their implicit tasks.

#pragma omp for *[clause[[,]clause] ...]*
for-loops

clause:

private(*list*)
firstprivate(*list*)
lastprivate(*list*)
linear(*list [: linear-step]*)
reduction(*reduction-identifier* : *list*)
schedule(*[modifier [, modifier]* : *]* *kind*[**,** *chunk_size*])
collapse(*n*)
ordered*[(n)]*
nowait

kind:

- **static:** Iterations are divided into chunks of size *chunk_size* and assigned to threads in the team in round-robin fashion in order of thread number.
- **dynamic:** Each thread executes a chunk of iterations then requests another chunk until none remain.
- **guided:** Each thread executes a chunk of iterations then requests another chunk until no chunks remain to be assigned.
- **auto:** The decision regarding scheduling is delegated to the compiler and/or runtime system.
- **runtime:** The schedule and chunk size are taken from the *run-sched-var* ICV.

modifier:

- **monotonic:** Each thread executes the chunks that it is assigned in increasing logical iteration order.
- **nonmonotonic:** Chunks are assigned to threads in any order and the behavior of an application that depends on execution order of the chunks is unspecified.
- **simd:** Ignored when the loop is not associated with a SIMD construct, otherwise the *new_chunk_size* for all except the first and last chunks is ⌈*chunk_size*/ *simd_width*⌉ ∗ *simd_width* where *simd_width* is an implementation-defined value.

sections [2.7.2] [2.7.2]

A noniterative worksharing construct that contains a set of structured blocks that are to be distributed among and executed by the threads in a team.

#pragma omp sections *[clause[[,] clause] ...]*
{
*[***#pragma omp section***]**
structured-block
*[***#pragma omp section**
structured-block]
...
}

clause:

private(*list*)
firstprivate(*list*)
lastprivate(*list*)
reduction(*reduction-identifier*: *list*)
nowait

single [2.7.3] [2.7.3]

Specifies that the associated structured block is executed by only one of the threads in the team.

#pragma omp single *[clause[[,]clause] ...]*
structured-block

clause:

private(*list*)
firstprivate(*list*)
copyprivate(*list*)
nowait

simd [2.8.1] [2.8.1]

Applied to a loop to indicate that the loop can be transformed into a SIMD loop.

#pragma omp simd *[clause[[,]clause] ...]*
for-loops

clause:

safelen(*length*)
simdlen(*length*)
linear(*list[* **:** *linear-step]*)
aligned(*list[* **:** *alignment]*)
private(*list*)
lastprivate(*list*)
reduction(*reduction-identifier* **:** *list*)
collapse(*n*)

declare simd [2.8.2] [2.8.2]

Enables the creation of one or more versions that can process multiple arguments using SIMD instructions from a single invocation from a SIMD loop.

#pragma omp declare simd *[clause[[,]clause] ...]*
*[***#pragma omp declare simd** *[clause[[,]clause] ...]]*
[...]
function definition or declaration

clause:

simdlen(*length*)
linear(*linear-list[* **:** *linear-step]*)
aligned(*argument-list[* **:** *alignment]*)
uniform(*argument-list*)
inbranch
notinbranch

for simd [2.8.3] [2.8.3]

Specifies that a loop that can be executed concurrently using SIMD instructions, and that those iterations will also be executed in parallel by threads in the team.

#pragma omp for simd *[clause[[,]clause] ...]*
for-loops

clause:

Any accepted by the **simd** or **for** directives with identical meanings and restrictions.

task [2.9.1] [2.11.1]

Defines an explicit task. The data environment of the task is created according to data-sharing attribute clauses on **task** construct and any defaults that apply.

#pragma omp task *[clause[[,]clause] ...]*
structured-block

clause:

if(*[* **task :** *]* *scalar-expression*)
final(*scalar-expression*)
untied
default(**shared** | **none**)
mergeable
private(*list*)
firstprivate(*list*)
shared(*list*)
depend(*dependence-type* **:** *list*)
priority(*priority-value*)

taskloop [2.9.2]

Specifies that the iterations of one or more associated loops will be executed in parallel using OpenMP tasks.

#pragma omp taskloop *[clause[[,]clause] ...]*
for-loops

clause:

if(*[* **taskloop :** *]* *scalar-expression*)
shared(*list*)
private(*list*)
firstprivate(*list*)
lastprivate(*list*)
default(**shared** | **none**)
grainsize(*grain-size*)
num_tasks(*num-tasks*)
collapse(*n*)
final(*scalar-expression*)
priority(*priority-value*)
untied
mergeable
nogroup

Continued

Directives and Constructs for C/C++ (continued)

taskloop simd [2.9.3]
Specifies that a loop that can be executed concurrently using SIMD instructions, and that those iterations will also be executed in parallel using OpenMP tasks.

#pragma omp taskloop simd *[clause[[,]clause] ...]*
for-loops

clause: Any accepted by the **simd** or **taskloop** directives with identical meanings and restrictions.

taskyield [2.9.4] [2.11.2]
Specifies that the current task can be suspended in favor of execution of a different task.

#pragma omp taskyield

target data [2.10.1] [2.9.1]
Creates a device data environment for the extent of the region.

#pragma omp target data *clause[[[,]clause] ...]*
structured-block

clause:
if(*[* **target data :** *]* *scalar-expression*)
device(*integer-expression*)
map(*[[map-type-modifier[,]] map-type***:]** *list*)
use_device_ptr(*list*)

target enter data [2.10.2]
Specifies that variables are mapped to a device data environment.

#pragma omp target enter data *[clause[[,]clause] ...]*

clause:
if(*[* **target enter data :** *]* *scalar-expression*)
device(*integer-expression*)
map(*[[map-type-modifier[,]] map-type* **:]** *list*)
depend(*dependence-type* **:** *list*)
nowait

target exit data [2.10.3]
Specifies that list items are unmapped from a device data environment.

#pragma omp target exit data *[clause[[,]clause] ...]*

clause:
if(*[* **target exit data :** *]* *scalar-expression*)
device(*integer-expression*)
map(*[[map-type-modifier[,]] map-type* **:]** *list*)
depend(*dependence-type* **:** *list*)
nowait

target [2.10.4] [2.9.2]
Map variables to a device data environment and execute the construct on that device.

#pragma omp target *[clause[[,]clause] ...]*
structured-block

clause:
if(*[* **target :** *]* *scalar-expression*)
device(*integer-expression*)
private(*list*)
firstprivate(*list*)
map(*[[map-type-modifier[,]] map-type***:]** *list*)
is_device_ptr(*list*)
defaultmap(**tofrom : scalar**)
nowait
depend(*dependence-type* **:** *list*)

target update [2.10.5] [2.9.3]
Makes the corresponding list items in the device data environment consistent with their original list items, according to the specified motion clauses.

#pragma omp target update *clause[[[,]clause] ...]*

clause is *motion-clause* or one of:
if(*[* **target update :** *]* *scalar-expression*)
device(*integer-expression*)
nowait
depend(*dependence-type* **:** *list*)

motion-clause:
to(*list*)
from(*list*)

declare target [2.10.6] [2.9.4]
A declarative directive that specifies that variables and functions are mapped to a device.

#pragma omp declare target
declarations-definition-seq
#pragma omp end declare target

#pragma omp declare target *(extended-list)*

#pragma omp declare target *clause[[,]clause ...]*

clause:
to(*extended-list*)
link(*list*)

teams [2.10.7] [2.9.5]
Creates a league of thread teams where the master thread of each team executes the region.

#pragma omp teams *[clause[[,]clause] ...]*
structured-block

clause:
num_teams(*integer-expression*)
thread_limit(*integer-expression*)
default(**shared** | **none**)
private(*list*)
firstprivate(*list*)
shared(*list*)
reduction(*reduction-identifier* **:** *list*)

distribute [2.10.8] [2.9.6]
Specifies loops which are executed by the thread teams.

#pragma omp distribute *[clause[[,]clause] ...]*
for-loops

clause:
private(*list*)
firstprivate(*list*)
lastprivate(*list*)
collapse(*n*)
dist_schedule(*kind[, chunk_size]*)

distribute simd [2.10.9] [2.9.7]
Specifies loops which are executed concurrently using SIMD instructions.

#pragma omp distribute simd *[clause[[,]clause] ...]*
for-loops

clause: Any accepted by the **distribute** or **simd** directives.

distribute parallel for [2.10.10] [2.9.8]
These constructs specify a loop that can be executed in parallel by multiple threads that are members of multiple teams.

#pragma omp distribute parallel for *[clause[[,]clause] ...]*
for-loops

clause: Any accepted by the **distribute** or **parallel for** directives

distribute parallel for simd [2.10.11] [2.9.9]
These constructs specify a loop that can be executed in parallel using SIMD semantics in the simd case by multiple threads that are members of multiple teams.

#pragma omp distribute parallel for simd *[clause[[,]clause] ...]*
for-loops

clause: Any accepted by the **distribute** or **parallel for simd** directives.

parallel for [2.11.1] [2.10.1]
Shortcut for specifying a **parallel** construct containing one or more associated loops and no other statements.

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

clause: Any accepted by the **parallel** or **for** directives, except the **nowait** clause, with identical meanings and restrictions.

parallel sections [2.11.2] [2.10.2]
Shortcut for specifying a **parallel** construct containing one **sections** construct and no other statements.

#pragma omp parallel sections *[clause[[,]clause] ...]*
{
*[***#pragma omp section***]*
structured-block
*[***#pragma omp section**
structured-block]
...
}

clause: Any accepted by the **parallel** or **sections** directives, except the **nowait** clause, with identical meanings and restrictions.

parallel for simd [2.11.4] [2.10.4]
Shortcut for specifying a **parallel** construct containing one **for simd** construct and no other statements.

#pragma omp parallel for simd *[clause[[,]clause] ...]*
for-loops

clause: Any accepted by the **parallel** or **for simd** directives, except the **nowait** clause, with identical meanings and restrictions.

target parallel [2.11.5]
Shortcut for specifying a **target** construct containing a **parallel** construct and no other statements.

#pragma omp target parallel *[clause[[,]clause] ...]*
structured-block

clause: Any accepted by the **target** or **parallel** directives, except for **copyin**, with identical meanings and restrictions.

target parallel for [2.11.6]
Shortcut for specifying a **target** construct containing a **parallel for** construct and no other statements.

#pragma omp target parallel for *[clause[[,]clause] ...]*
for-loops

clause: Any accepted by the **target** or **parallel for** directives, except for **copyin**, with identical meanings and restrictions.

target parallel for simd [2.11.7]
Shortcut for specifying a **target** construct containing a **parallel for simd** construct and no other statements.

#pragma omp target parallel for simd *[clause[[,]clause] ...]*
for-loops

clause: Any accepted by the **target** or **parallel for simd** directives, except for **copyin**, with identical meanings and restrictions.

Continued

Directives and Constructs for C/C++ (continued)

target simd [2.11.8]

Shortcut for specifying a **target** construct containing a **simd** construct and no other statements.

#pragma omp target simd *[clause[[,]clause] ...]*
for -loops

clause: Any accepted by the **target** or **simd** directives with identical meanings and restrictions.

target teams [2.11.9] [2.10.5]

Shortcut for specifying a **target** construct containing a **teams** construct and no other statements.

#pragma omp target teams *[clause[[,]clause] ...]*
structured-block

clause: Any accepted by the **target** or **teams** directives with identical meanings and restrictions.

teams distribute [2.11.10] [2.10.6]

Shortcuts for specifying a **teams** construct containing a **distribute** construct and no other statements.

#pragma omp teams distribute *[clause[[,]clause] ...]*
for-loops

clause: Any clause used for teams or distribute, with identical meanings and restrictions.

teams distribute simd [2.11.11] [2.10.7]

Shortcuts for specifying a **teams** construct containing a **distribute simd** construct and no other statements.

#pragma omp teams distribute simd *[clause[[,]clause] ...]*
for-loops

clause: Any clause used for **teams** or **distribute simd**, with identical meanings and restrictions.

target teams distribute [2.11.12] [2.10.8]

Shortcuts for specifying a **target** construct containing a **teams distribute** construct and no other statements.

#pragma omp target teams distribute *[clause[[,]clause] ...]*
for-loops

clause: Any clause used for **target** or **teams distribute**

target teams distribute simd [2.11.13] [2.10.9]

Shortcuts for specifying a **target** construct containing a **teams distribute simd** construct and no other statements.

#pragma omp target teams distribute simd *[clause[[,] clause] ...]*
for-loops

clause: Any clause used for **target** or **teams distribute simd**

teams distribute parallel for [2.11.14] [2.10.10]

Shortcuts for specifying a **teams** construct containing a **distribute parallel for** construct and no other statements.

#pragma omp teams distribute parallel for *[clause[[,] clause] ...]*
for-loops

clause: Any clause used for **teams** or **distribute parallel for**

target teams distribute parallel for [2.11.15] [2.10.11]

Shortcut for specifying a **target** construct containing a **teams distribute parallel for** construct and no other statements.

#pragma omp target teams distribute parallel for *[clause[[,]clause] ...]*
for-loops

clause: Any clause used for **teams distribute parallel for** or **target**

teams distribute parallel for simd [2.11.16] [2.10.12]

Shortcut for specifying a **teams** construct containing a **distribute parallel for simd** construct and no other statements.

#pragma omp teams distribute parallel for simd *[clause[[,] clause] ...]*
for-loops

clause: Any clause used for **distribute parallel for simd** or **teams**

target teams distribute parallel for simd [2.11.17] [2.10.13]

Shortcut for specifying a **target** construct containing a **teams distribute parallel for simd** construct and no other statements.

#pragma omp target teams distribute parallel for simd *[clause[[,]clause] ...]*
for-loops

clause: Any clause used for **teams distribute parallel for simd** or **target**

master [2.13.1] [2.12.1]

Specifies a structured block that is executed by the master thread of the team.

#pragma omp master
structured-block

critical [2.13.2] [2.12.2]

Restricts execution of the associated structured block to a single thread at a time.

#pragma omp critical *[(name)* **[hint (***hint-expression***)]]**
structured-block

barrier [2.13.3] [2.12.3]

Specifies an explicit barrier at the point at which the construct appears.

#pragma omp barrier

taskwait [2.13.4] [2.12.4]

Specifies a wait on the completion of child tasks of the current task.

#pragma omp taskwait

taskgroup [2.13.5] [2.12.5]

Specifies a wait on the completion of child tasks of the current task, then waits for descendant tasks.

#pragma omp taskgroup
structured-block

atomic [2.13.6] [2.12.6]

Ensures that a specific storage location is accessed atomically. May take one of the following three forms:

#pragma omp atomic *[***seq_cst***[,]] atomic-clause [[,]***seq_cst***]*
expression-stmt

#pragma omp atomic *[***seq_cst***]*
expression-stmt

(**atomic** continues in the next column)

#pragma omp atomic *[***seq_cst***[,]]* **capture** *[[,]***seq_cst***]*
structured-block

atomic clause*:* **read**, **write**, **update**, or **capture**

(**atomic** continues in the next column)

atomic (continued)

expression-stmt may be one of*:*

if *atomic clause* **is...**	***expression-stmt*:**
read	*v = x;*
write	*x = expr;*
update or is not present	*x++; x--; ++x; --x; x binop= expr; x = x binop expr; x = expr binop x;*
capture	*v=x++; v=x--; v=++x; v= --x; v=x binop= expr; v=x = x binop expr; v=x = expr binop x;*

structured-block may be one of the following forms:

{v = x; x binop= expr;}
{v = x; x = x binop expr;}
{x = x binop expr; v = x;}
{v = x; x = expr;}
{v = x; ++x;}
{x++; v = x;}
{v = x; --x;}
{x--; v = x;}
{x binop= expr; v = x;}
{v = x; x = expr binop x;}
{x = expr binop x; v = x;}
{v = x; x++;}
{++x; v = x;}
{v = x; x--;}
{--x; v = x;}

flush [2.13.7] [2.12.7]

Executes the OpenMP flush operation, which makes a thread's temporary view of memory consistent with memory, and enforces an order on the memory operations of the variables.

#pragma omp flush *[(list)]*

ordered [2.13.8] [2.12.8]

Specifies a structured block in a loop, **simd**, or loop SIMD region that will be executed in the order of the loop iterations.

#pragma omp ordered *[clause[[,] clause]...]*
structured-block

clause:
threads
simd

#pragma omp ordered *clause[[[,] clause]...]*

clause:
depend (source)
depend (sink : *vec*)

cancel [2.14.1] [2.13.1]

Requests cancellation of the innermost enclosing region of the type specified. The **cancel** directive may not be used in place of the statement following an **if**, **while**, **do**, **switch**, or **label**.

#pragma omp cancel *construct-type-clause[[,] if-clause]*

construct-type-clause:
parallel
sections
for
taskgroup

if-clause:
if(*scalar-expression*)

cancellation point [2.14.2] [2.13.2]

Introduces a user-defined cancellation point at which tasks check if cancellation of the innermost enclosing region of the type specified has been activated.

#pragma omp cancellation point *construct-type-clause*

construct-type-clause:
parallel
sections
for
taskgroup

Continued

Directives and Constructs for C/C++ (continued)

threadprivate [2.15.2] [2.14.2]

Specifies that variables are replicated, with each thread having its own copy. Each copy of a threadprivate variable is initialized once prior to the first reference to that copy.

#pragma omp threadprivate(*list***)**

list: A comma-separated list of file-scope, namespace-scope, or static block-scope variables that do not have incomplete types.

declare reduction [2.16] [2.15]

Declares a *reduction-identifier* that can be used in a **reduction** clause.

#pragma omp declare reduction(*reduction-identifier* **:** *typename-list* **:** *combiner***)** *[initializer-clause]*

reduction-identifier: A base language identifer (for C), or an *id-expression* (for C++), or one of the following operators: +, -, *, &, |, ^, && and ||

typename-list: A list of type names

combiner: An expression

initializer-clause: **initializer** (*initializer-expr*) where *initializer-expr* is **omp_priv** = *initializer* or *function-name* (*argument-list*)

Runtime Library Routines for C/C++

Execution environment routines affect and monitor threads, processors, and the parallel environment. The library routines are external functions with "C" linkage.

Execution Environment Routines

omp_set_num_threads [3.2.1] [3.2.1]

Affects the number of threads used for subsequent parallel regions not specifying a **num_threads** clause, by setting the value of the first element of the *nthreads-var* ICV of the current task to *num_threads*.

void omp_set_num_threads(int *num_threads***);**

omp_get_num_threads [3.2.2] [3.2.2]

Returns the number of threads in the current team. The binding region for an **omp_get_num_threads** region is the innermost enclosing **parallel** region.

int omp_get_num_threads(void**);**

omp_get_max_threads [3.2.3] [3.2.3]

Returns an upper bound on the number of threads that could be used to form a new team if a **parallel** construct without a **num_threads** clause were encountered after execution returns from this routine.

int omp_get_max_threads(void**);**

omp_get_thread_num [3.2.4] [3.2.4]

Returns the thread number of the calling thread within the current team.

int omp_get_thread_num(void**);**

omp_get_num_procs [3.2.5] [3.2.5]

Returns the number of processors that are available to the device at the time the routine is called.

int omp_get_num_procs(void**);**

omp_in_parallel [3.2.6] [3.2.6]

Returns *true* if the *active-levels-var* ICV is greater than zero; otherwise it returns *false*.

int omp_in_parallel(void**);**

omp_set_dynamic [3.2.7] [3.2.7]

Enables or disables dynamic adjustment of the number of threads available for the execution of subsequent **parallel** regions by setting the value of the *dyn-var* ICV.

void omp_set_dynamic(int *dynamic_threads***);**

omp_get_dynamic [3.2.8] [3.2.8]

This routine returns the value of the *dyn-var* ICV, which is *true* if dynamic adjustment of the number of threads is enabled for the current task.

int omp_get_dynamic(void**);**

omp_get_cancellation [3.2.9] [3.2.9]

Returns the value of the *cancel-var* ICV, which is *true* if cancellation is activated; otherwise it returns *false*.

int omp_get_cancellation(void**);**

omp_set_nested [3.2.10] [3.2.10]

Enables or disables nested parallelism, by setting the *nest-var* ICV.

void omp_set_nested(int *nested***);**

omp_get_nested [3.2.11] [3.2.10]

Returns the value of the *nest-var* ICV, which indicates if nested parallelism is enabled or disabled.

int omp_get_nested(void**);**

omp_set_schedule [3.2.12] [3.2.12]

Affects the schedule that is applied when **runtime** is used as schedule kind, by setting the value of the *run-sched-var* ICV.

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

kind: One of the following, or an implementation-defined schedule*:*

omp_sched_static	= 1
omp_sched_dynamic	= 2
omp_sched_guided	= 3
omp_sched_auto	= 4

omp_get_schedule [3.2.13] [3.2.13]

Returns the value of *run-sched-var* ICV, which is the schedule applied when **runtime** schedule is used.

void omp_get_schedule(
omp_sched_t **kind*, int **chunk_size***);**

See *kind* for **omp_set_schedule**.

omp_get_thread_limit [3.2.14] [3.2.14]

Returns the value of the *thread-limit-var* ICV, which is the maximum number of OpenMP threads available.

int omp_get_thread_limit(void**);**

omp_set_max_active_levels [3.2.15] [3.2.15]

Limits the number of nested active parallel regions, by setting *max-active-levels-var* ICV.

void omp_set_max_active_levels(int *max_levels***);**

omp_get_max_active_levels [3.2.16] [3.2.16]

Returns the value of *max-active-levels-var* ICV, which determines the maximum number of nested active parallel regions.

int omp_get_max_active_levels(void**);**

omp_get_level [3.2.17] [3.2.17]

For the enclosing device region, returns the *levels-vars* ICV, which is the number of nested **parallel** regions that enclose the task containing the call.

int omp_get_level(void**);**

omp_get_ancestor_thread_num [3.2.18] [3.2.18]

Returns, for a given nested level of the current thread, the thread number of the ancestor of the current thread.

int omp_get_ancestor_thread_num(int *level***);**

omp_get_team_size [3.2.19] [3.2.19]

Returns, for a given nested level of the current thread, the size of the thread team to which the ancestor or the current thread belongs.

int omp_get_team_size(int *level***);**

omp_get_active_level [3.2.20] [3.2.20]

Returns the value of the *active-level-vars* ICV, which determines the number of active, nested **parallel** regions enclosing the task that contains the call.

int omp_get_active_level(void**);**

omp_in_final [3.2.21] [3.2.21]

Returns *true* if the routine is executed in a final task region; otherwise, it returns *false*.

int omp_in_final(void**);**

omp_get_proc_bind [3.2.22] [3.2.22]

Returns the thread affinity policy to be used for the subsequent nested **parallel** regions that do not specify a **proc_bind** clause.

omp_proc_bind_t omp_get_proc_bind(void**);**

Returns one of:

omp_proc_bind_false	= 0
omp_proc_bind_true	= 1
omp_proc_bind_master	= 2
omp_proc_bind_close	= 3
omp_proc_bind_spread	= 4

omp_get_ num_places [3.2.23]

Returns the number of places available to the execution environment in the place list.

int omp_get_num_places(void**);**

omp_get_place_num_procs [3.2.24]

Returns the number of processors available to the execution environment in the specified place.

int omp_get_place_num_procs(int *place_num***);**

omp_get_place_proc_ids [3.2.25]

Returns the numerical identifiers of the processors available to the execution environment in the specified place.

void omp_get_place_proc_ids(
int *place_num*, int **ids***);**

omp_get_place_num [3.2.26]

Returns the place number of the place to which the encountering thread is bound.

int omp_get_place_num(void**);**

omp_get_partition_num_places [3.2.27]

Returns the number of places in the place partition of the innermost implicit task.

int omp_get_partition_num_places(void**);**

omp_get_partition_place_nums [3.2.28]

Returns the list of place numbers corresponding to the places in the *place-partition-var* ICV of the innermost implicit task.

void omp_get_partition_place_nums(int **place_nums***);**

Continued

Runtime Library Routines for C/C++ (continued)

omp_set_default_device [3.2.29] [3.2.23]
Controls the default target device by assigning the value of the *default-device-var* ICV.

```
void omp_set_default_device(int device_num);
```

omp_get_default_device [3.2.30] [3.2.24]
Returns the value of the *default-device-var* ICV, which determines default target device.

```
int omp_get_default_device(void);
```

omp_get_num_devices [3.2.31] [3.2.25]
Returns the number of target devices.

```
int omp_get_num_devices(void);
```

omp_get_num_teams [3.2.32] [3.2.26]
Returns the number of teams in the current **teams** region, or 1 if called from outside of a **teams** region.

```
int omp_get_num_teams(void);
```

omp_get_team_num [3.2.33] [3.2.27]
Returns the team number of calling thread. The team number is an integer between 0 and one less than the value returned by **omp_get_num_teams()**, inclusive.

```
int omp_get_team_num(void);
```

omp_is_initial_device [3.2.34] [3.2.28]
Returns *true* if the current task is executing on the host device; otherwise, it returns *false*.

```
int omp_is_initial_device(void);
```

omp_get_initial_device [3.2.35]
Returns a device number representing the host device.

```
int omp_get_initial_device(void);
```

omp_get_max_task_priority [3.2.36]
Returns the maximum value that can be specified in the **priority** clause.

```
int omp_get_max_task_priority(void);
```

Lock Routines
General-purpose lock routines. Two types of locks are supported: simple locks and nestable locks. A nestable lock can be set multiple times by the same task before being unset; a simple lock cannot be set if it is already owned by the task trying to set it.

Initialize lock [3.3.1] [3.3.1]
Initialize an OpenMP lock.

```
void omp_init_lock(omp_lock_t *lock);
void omp_init_nest_lock(omp_nest_lock_t *lock);
```

Initialize lock with hint [3.3.2]
Initialize an OpenMP lock with a hint.

```
void omp_init_lock_with_hint(
    omp_lock_t *lock,
    omp_lock_hint_t hint);
void omp_init_nest_lock_with_hint(
    omp_nest_lock_t *lock,
    omp_nest_lock_hint_t hint);
```

hint:

omp_lock_hint_none	= 0
omp_lock_hint_uncontended	= 1
omp_lock_hint_contended	= 2
omp_lock_hint_nonspeculative	= 4
omp_lock_hint_speculative	= 8

Destroy lock [3.3.3] [3.3.2]
Ensure that the OpenMP lock is uninitialized.

```
void omp_destroy_lock(omp_lock_t *lock);
void omp_destroy_nest_lock(omp_lock_t *lock);
```

Set lock [3.3.4] [3.3.3]
Sets an OpenMP lock. The calling task region is suspended until the lock is set.

```
void omp_set_lock(omp_lock_t *lock);
void omp_set_nest_lock(omp_nest_lock_t *lock);
```

Unset lock [3.3.5] [3.3.4]
Unsets an OpenMP lock.

```
void omp_unset_lock(omp_lock_t *lock);
void omp_unset_nest_lock(omp_nest_lock_t *lock);
```

Test lock [3.3.6] [3.3.5]
Attempt to set an OpenMP lock but do not suspend execution of the task executing the routine.

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

Timing Routines
Timing routines support a portable wall clock timer. These record elapsed time per-thread and are not guaranteed to be globally consistent across all the threads participating in an application.

omp_get_wtime [3.4.1] [3.4.1]
Returns elapsed wall clock time in seconds.

```
double omp_get_wtime(void);
```

omp_get_wtick [3.4.2] [3.4.2]
Returns the precision of the timer (seconds between ticks) used by **omp_get_wtime**.

```
double omp_get_wtick(void);
```

Device Memory Routines
Timing routines support allocation and management of pointers in the data environments of target devices.

omp_target_alloc [3.5.1]
Allocates memory in a device data environment.

```
void* omp_target_alloc(size_t size, int device_num);
```

omp_target_free [3.5.2]
Frees the device memory allocated by the **omp_target_alloc** routine.

```
void omp_target_free(void * device_ptr, int device_num);
```

omp_target_is_present [3.5.3]
Validates whether a host pointer has an associated device buffer on a given device.

```
int omp_target_is_present(void * ptr, int device_num);
```

omp_target_memcpy [3.5.4]
Copies memory between any combination of host and device pointers.

```
int omp_target_memcpy(void * dst, void * src,
    size_t length, size_t dst_offset, size_t src_offset,
    int dst_device_num, int src_device_num);
```

omp_target_memcpy_rect [3.5.5]
Copies a rectangular subvolume from a multi-dimensional array to another multi-dimensional array.

```
int omp_target_memcpy_rect(
    void * dst, void * src, size_t element_size, int num_dims,
    const size_t * volume, const size_t * dst_offsets,
    const size_t * src_offsets, const size_t * dst_dimensions,
    const size_t * src_dimensions, int dst_device_num,
    int src_device_num);
```

omp_target_associate_ptr [3.5.6]
Maps a device pointer, which may be returned from **omp_target_alloc** or implementation-defined runtime routines, to a host pointer.

```
int omp_target_associate_ptr(void * host_ptr,
    void * device_ptr, size_t size, size_t device_offset,
    int device_num);
```

omp_target_disassociate_ptr [3.5.7]
Removes the associated pointer for a given device from a host pointer.

```
int omp_target_disassociate_ptr(void * ptr,
    int device_num);
```

Notes

Clauses

The set of clauses that is valid on a particular directive is described with the directive. Most clauses accept a comma-separated list of list items. All list items appearing in a clause must be visible, according to the scoping rules of the base language. Not all of the clauses listed in this section are valid on all directives.

If Clause [2.12]

The effect of the **if** clause depends on the construct to which it is applied.

if(*[directive-name-modifier :] scalar-expression*)
For combined or composite constructs, it only applies to the semantics of the construct named in the *directive-name-modifier* if one is specified. If none is specified for a combined or composite construct then the **if** clause applies to all constructs to which an **if** clause can apply.

Depend Clause [2.13.9]

Enforces additional constraints on the scheduling of tasks or loop iterations. These constraints establish dependences only between sibling tasks or between loop iterations.

depend(*dependence-type* **:** *list*)
Where *dependence-type* may be **in**, **out**, or **inout**:

in: The generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an **out** or **inout** *dependence-type* list.

out and **inout**: The generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an **in**, **out**, or **inout** *dependence-type list*.

---

depend(*dependence-type*)
Where *dependence-type* may be **source**.

---

depend(*dependence-type [*: *vec]*)
Where *dependence-type* may be **sink** and is the iteration vector, which has the form:
x_1 [± d_1], x_2 [± d_2], . . . , x_n [± d_n]

Data Sharing Attribute Clauses [2.15.3] [2.14.3]

Data-sharing attribute clauses apply only to variables whose names are visible in the construct on which the clause appears.

default(**shared** | **none**)
Explicitly determines the default data-sharing attributes of variables that are referenced in a **parallel**, **teams**, or task generating construct, causing all variables referenced in the construct that have implicitly determined data-sharing attributes to be shared.

shared(*list*)
Declares one or more list items to be shared by tasks generated by a **parallel**, **teams**, or task generating construct. The programmer must ensure that storage shared by an explicit **task** region does not reach the end of its lifetime before the explicit task region completes its execution.

private(*list*)
Declares one or more list items to be private to a task or a SIMD lane. Each task that references a list item that appears in a **private** clause in any statement in the construct receives a new list item.

firstprivate(*list*)
Declares list items to be private to a task, and initializes each of them with the value that the corresponding original item has when the construct is encountered.

lastprivate(*list*)
Declares one or more list items to be private to an implicit task or to a SIMD lane, and causes the corresponding original list item to be updated after the end of the region.

linear(*linear-list[:linear-step]*)
Declares one or more list items to be private to a SIMD lane and to have a linear relationship with respect to the iteration space of a loop. Clause *linear-list* is *list* or *modifer(list)*. *modifier* may be one of **ref**, **val**, or **uval**; except in C it may only be **val**.

reduction(*reduction-identifier*:*list*)
Specifies a *reduction-identifier* and one or more list items. The *reduction-identifier* must match a previously declared *reduction-identifier* of the same name and type for each of the list items.

Operators for reduction (initialization values)			
+	(0)	**|**	(0)
*****	(1)	**^**	(0)
-	(0)	**&&**	(1)
&	(~0)	**||**	(0)
max (Least representable number in **reduction** list item type)			
min (Largest representable number in **reduction** list item type)			

SIMD Clauses [2.8]

safelen(*length*)
If used then no two iterations executed concurrently with SIMD instructions can have a greater distance in the logical iteration space than its value.

collapse(*n*)
A constant positive integer expression that specifies how many loops are associated with the loop construct.

simdlen(*length*)
A constant positive integer expression that specifies the number of concurrent arguments of the function.

aligned(*argument-list[:alignment]*)
Declares one or more list items to be aligned to the specified number of bytes. *alignment*, if present, must be a constant positive integer expression. If no optional parameter is specified, implementation-defined default alignments for SIMD instructions on the target platforms are assumed.

uniform(*argument-list*)
Declares one or more arguments to have an invariant value for all concurrent invocations of the function in the execution of a single SIMD loop.

inbranch
Specifies that the function will always be called from inside a conditional statement of a SIMD loop.

notinbranch
Specifies that the function will never be called from inside a conditional statement of a SIMD loop.

Data Copying Clauses [2.15.4] [2.14.4]

copyin(*list*)
Copies the value of the master thread's threadprivate variable to the threadprivate variable of each other member of the team executing the **parallel** region.

copyprivate(*list*)
Broadcasts a value from the data environment of one implicit task to the data environments of the other implicit tasks belonging to the **parallel** region.

Map Clause [2.15.5] [2.14.5]

map(*[[map-type-modifier[,]] map-type:]list*)
Map a variable from the task's data environment to the device data environment associated with the construct.
map-type:

alloc: On entry to the region each new corresponding list item has an undefined initial value.

to: On entry to the region each new corresponding list item is initialized with the original list item's value.

from: On exit from the region the corresponding list item's value is assigned to each original list item

tofrom: (Default) On entry to the region each new corresponding list item is initialized with the original list item's value, and on exit from the region the corresponding list item's value is assigned to each original list item.

release: On exit from the region, the corresponding list item's reference count is decremented by one.

delete: On exit from the region, the corresponding list item's reference count is set to zero.

map-type-modifer:
Must be **always**.

Defaultmap Clause [2.15.5.2]

defaultmap(tofrom:scalar)
Causes all scalar variables referenced in the construct that have implicitly determined data-mapping attributes to have the **tofrom** *map-type*.

Tasking Clauses [2.9]

final(*scalar-logical-expr*)
The generated task will be a final task if the final expression evaluates to true.

mergeable
Specifies that the generated task is a mergeable task.

priority(*priority-value*)
A non-negative numerical scalar expression that specifies a hint for the priority of the generated task.

grainsize(*grain-size*)
Causes the number of logical loop iterations assigned to each created task to be greater than or equal to the minimum of the value of the *grain-size* expression and the number of logical loop iterations, but less than two times the value of the *grain-size* expression.

num_tasks(*num-tasks*)
Create as many tasks as the minimum of the *num-tasks* expression and the number of logical loop iterations.

MPI Quick Reference in C

#include <mpi.h>

Environmental Management:

int **MPI_Init**(int *argc, char **argv[])

int **MPI_Finalize**(void)

int **MPI_Initialized**(int *flag)

int **MPI_Finalized**(int *flag)

int **MPI_Comm_size**(MPI_Comm comm, int *size)

int **MPI_Comm_rank**(MPI_Comm comm, int *rank)

int **MPI_Abort**(MPI_Comm comm, int errorcode)

double **MPI_Wtime**(void)

double **MPI_Wtick**(void)

Blocking Point-to-Point-Communication:

int **MPI_Send** (void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

Related: MPI_Bsend, MPI_Ssend, MPI_Rsend

int **MPI_Recv** (void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

int **MPI_Probe** (int source, int tag, MPI_Comm comm, MPI_Status *status)

int **MPI_Get_count** (MPI_Status *status, MPI_Datatype datatype, int *count)

Related: MPI_Get_elements

int **MPI_Sendrecv** (void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status *status)

int **MPI_Sendrecv_replace** (void *buf, int count, MPI_Datatype datatype, int dest, int sendtag, int source, int recvtag, MPI_Comm comm, MPI_Status *status)

int **MPI_Buffer_attach** (void *buffer, int size)

int **MPI_Buffer_detach** (void *bufferptr, int *size)

Non-Blocking Point-to-Point-Communication:

int **MPI_Isend** (void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)

Related: MPI_Ibsend, MPI_Issend, MPI_Irsend

int **MPI_Irecv** (void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)

int **MPI_Iprobe** (int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status)

int **MPI_Wait** (MPI_Request *request, MPI_Status *status)

int **MPI_Test** (MPI_Request *request, int *flag, MPI_Status *status)

int **MPI_Waitall** (int count, MPI_Request request_array[], MPI_Status status_array[])

Related: MPI_Testall

int **MPI_Waitany** (int count, MPI_Request request_array[], int *index, MPI_Status *status)

Related: MPI_Testany

int **MPI_Waitsome** (int incount, MPI_Request request_array[], int *outcount, int index_array[], MPI_Status status_array[])

Related: MPI_Testsome,

int **MPI_Request_free** (MPI_Request *request)

Related: MPI_Cancel

int **MPI_Test_cancelled** (MPI_Status *status, int *flag)

Collective Communication:

int **MPI_Barrier** (MPI_Comm comm)

int **MPI_Bcast** (void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

int **MPI_Gather** (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

int **MPI_Gatherv** (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount_array[], int displ_array[], MPI_Datatype recvtype, int root, MPI_Comm comm)

int **MPI_Scatter** (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

int **MPI_Scatterv** (void *sendbuf, int sendcount_array[], int displ_array[] MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

int **MPI_Allgather** (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)

Related: MPI_Alltoall

int **MPI_Allgatherv** (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount_array[], int displ_array[], MPI_Datatype recvtype, MPI_Comm comm)

Related: MPI_Alltoallv

int **MPI_Reduce** (void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

int **MPI_Allreduce** (void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

Related: MPI_Scan, MPI_Exscan

int **MPI_Reduce_scatter** (void *sendbuf, void *recvbuf, int recvcount_array[], MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

int **MPI_Op_create** (MPI_User_function *func, int commute, MPI_Op *op)

int **MPI_Op_free** (MPI_Op *op)

Derived Datatypes:

int **MPI_Type_commit** (MPI_Datatype *datatype)

int **MPI_Type_free** (MPI_Datatype *datatype)

int **MPI_Type_contiguous** (int count, MPI_Datatype oldtype, MPI_Datatype

*newtype)

int **MPI_Type_vector** (int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)

int **MPI_Type_indexed** (int count, int blocklength_array[], int displ_array[], MPI_Datatype oldtype, MPI_Datatype *newtype)

int **MPI_Type_create_struct** (int count, int blocklength_array[], MPI_Aint displ_array[], MPI_Datatype oldtype_array[], MPI_Datatype *newtype)

int **MPI_Type_create_subarray** (int ndims, int size_array[], int subsize_array[], int start_array[], int order, MPI_Datatype oldtype, MPI_Datatype *newtype)

int **MPI_Get_address** (void *location, MPI_Aint *address)

int **MPI_Type_size** (MPI_Datatype *datatype, int *size)

int **MPI_Type_get_extent** (MPI_Datatype datatype, MPI_Aint *lb, MPI_Aint *extent)

int **MPI_Pack** (void *inbuf, int incount, MPI_Datatype datatype, void *outbuf, int outcount, int *position, MPI_Comm comm)

int **MPI_Unpack** (void *inbuf, int insize, int *position, void *outbuf, int outcount, MPI_Datatype datatype, MPI_Comm comm)

int **MPI_Pack_size** (int incount, MPI_Datatype datatype, MPI_Comm comm, int *size)

Related: MPI_Type_create_hvector, MPI_Type_create_hindexed, MPI_Type_create_indexed_block, MPI_Type_create_darray, MPI_Type_create_resized, MPI_Type_get_true_extent, MPI_Type_dup, MPI_Pack_external, MPI_Unpack_external, MPI_Pack_external_size

Groups and Communicators:

int **MPI_Group_size** (MPI_Group group, int *size)

int **MPI_Group_rank** (MPI_Group group, int *rank)

int **MPI_Comm_group** (MPI_Comm comm, MPI_Group *group)

int **MPI_Group_translate_ranks** (MPI_Group group1, int n, int rank1_array[], MPI_Group group2, int rank2_array[])

int **MPI_Group_compare** (MPI_Group group1, MPI_Group group2, int *result)

MPI_IDENT, MPI_COMGRUENT, MPI_SIMILAR, MPI_UNEQUAL

int **MPI_Group_union** (MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)

Related: MPI_Group_intersection, MPI_Group_difference

int **MPI_Group_incl** (MPI_Group group, int n, int rank_array[], MPI_Group *newgroup)

Related: MPI_Group_excl

int **MPI_Comm_create** (MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)

int **MPI_Comm_compare** (MPI_Comm comm1, MPI_Comm comm2, int *result)

MPI_IDENT, MPI_COMGRUENT, MPI_SIMILAR, MPI_UNEQUAL

int **MPI_Comm_dup** (MPI_Comm comm, MPI_Comm *newcomm)

int **MPI_Comm_split** (MPI_Comm comm, int color, int key, MPI_Comm *newcomm)

int **MPI_Comm_free** (MPI_Comm *comm)

Topologies:

int **MPI_Dims_create** (int nnodes, int ndims, int *dims)

int **MPI_Cart_create** (MPI_Comm comm_old, int ndims, int dims_array[], int periods_array[], int reorder, MPI_Comm *comm_cart)

int **MPI_Cart_shift** (MPI_Comm comm, int direction, int disp, int *rank_source, int *rank_dest)

int **MPI_Cartdim_get** (MPI_Comm comm, int *ndim)

int **MPI_Cart_get** (MPI_Comm comm, int naxdim, int *dims, int *periods, int *coords)

int **MPI_Cart_rank** (MPI_Comm comm, int coords_array[], int *rank)

int **MPI_Cart_coords** (MPI_Comm comm, int rank, int maxdims, int *coords)

int **MPI_Cart_sub** (MPI_Comm comm_old, int remain_dims_array[], MPI_Comm *comm_new)

int **MPI_Cart_map** (MPI_Comm comm_old, int ndims, int dims_array[], int periods_array[], int *new_rank)

int **MPI_Graph_create** (MPI_Comm comm_old, int nnodes, int index_array[], int edges_array[], int reorder, MPI_Comm *comm_graph)

int **MPI_Graph_neighbors_count** (MPI_Comm comm, int rank, int *nneighbors)

int **MPI_Graph_neighbors** (MPI_Comm comm, int rank, int maxneighbors, int *neighbors)

int **MPI_Graphdims_get** (MPI_Comm comm, int *nnodes, int *nedges)

int **MPI_Graph_get** (MPI_Comm comm, int maxindex, int maxedges, int *index, int *edges)

int **MPI_Graph_map** (MPI_Comm comm_old, int nnodes, int index_array[], int edges_array[], int *new_rank)

int **MPI_Topo_test** (MPI_Comm comm, int *topo_type)

Wildcards:

MPI_ANY_TAG, MPI_ANY_SOURCE

Basic Datatypes:

MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG, MPI_UNSIGNED_CHAR, MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LONG MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE, MPI_BYTE, MPI_PACKED

Predefined Groups and Communicators:

MPI_GROUP_EMPTY, MPI_GROUP_NULL, MPI_COMM_WORLD, MPI_COMM_SELF, MPI_COMM_NULL

Reduction Operations:

MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_BAND, MPI_BOR, MPI_BXOR, MPI_LAND, MPI_LOR, MPI_LXOR

Status Object:

status.MPI_SOURCE, status.MPI_TAG, status.MPI_ERROR