CPDS: OpenMP cheatsheet

Directives syntax

In C/C++, OpenMP is used through compiler directives. The syntax is ignored if the compiler does not know OpenMP.

```
#pragma omp construct [clauses]
```

Memory model

OpenMP defines a relaxed memory model. Threads can see different values for the same variable (Variables can be shared or private to each thread). Memory consistency is only guaranteed at specific points.

Constructs

The `parallel` construct

```
#pragma omp parallel [clauses]
    // structured block...
```

In that sense, directives work just like `if` statements. If the conditional block of the `if` is just one block, it can be written just like this:

```
if (something) doIt();
```

This syntax can be used for *joining* different blocks even though they are not in the same line, as shown below. The `for` loop will only run if the condition `something` is true, even though we didn't enclose the conditional block between curly braces ({ }).

```
if (something)
for (;;) {
    // do something...
}
```

The same goes for OpenMP directives. The following two pieces of code are equivalent.

```
#pragma omp parallel
{
    for (;;) {
        // do something...
    }
}
```

```
#pragma omp parallel
for (;;) {
    // do something...
}
//
```

Number of threads

The `nthreads-var` ICV (internal variable) is used to determine the number of threads to be used for parallel regions. It's a list of positive integer values. For each occurrence of `parallel`, the first element is popped from the list.

The value of the variable can be set with the environment variable `OMP_NUM_THREADS`, through the function `omp_set_num_threads`, or by defining the directive `num_threads`.

```
unsigned int N = ...;
omp_set_num_threads(N);
#pragma omp parallel num_threads(N)
```

The `if` clause

It can be used to conditionally run regions parallelly. When the condition evaluates to false, the region is run on one only thread.

```
int iterations = ...;
#pragma omp parallel if (iterations > 10)
```

Example: amount of threads

```
void main() {
    #pragma omp parallel
    // ...
    omp_set_num_threads(2);
    #pragma omp parallel
    //...
    #pragma omp parallel num_threads(random() % 4 + 1)
     ↪   if(0)
    //...
}
```

How many threads are used in each parallel region? The first region uses the default amount of threads. The second region will use 2 threads. The third region will not run parallely, as the `if` will always evaluate to false.

Data-sharing attributes

Shared

When a variable is marked as `shared`, all threads use the same variable, this is, access the same location in memory. By default, variables are implicitly shared.

Private

When a variable is marked as `private`, it means that each thread has a different variable with an originally undefined value that can be accessed without any kind of synchronization.

Firstprivate

Whan a variable is marked as `firstprivate`, it means that each thread has a different variable initialized to the original value that can be accessed without any kind of synchronization.

Example: Data-sharing

```
int x = 1;
#pragma omp parallel XXXXXX num_threads(2)
{
    x++;
    printf("%d\n", x);
}
printf("%d\n", x);
```

What gets printed on screen if `XXXXXX` is `shared(x)`, `private(x)` or `firstprivate(x)`? Note that the final `printf` falls out of the `parallel` construct.

When `x` is shared, it is difficult to know. A race condition will occur. One of the threads will run `x++` before the other. So, the two first lines will be a 2 and a 3 (with undetermined order), followed by a 3.

When `x` is private, the result is undefined, as the variables that the threads will be able to access are undefined.

When `x` is firstprivate, it will print 2 twice, because each thread modifies a private variable initialized to 1. The original variable remains untouched, so it's just where it was initialized.

Try it yourself

Compile the source `data-sharing.c` with `make data-sharing` and run it with `./data-sharing`. Because of the compilation options, the uninitialized variables will be set to 0 by default. Thus, the example with private variables will print 1 three times.

Example: Computation of π

An approximation of π can be calculated with the following sequential code.

```
1  static long num_steps = 100000;
2  double step;
3
4  void main() {
5      int i;
6      double x, pi, sum = 0.0;
7
8      step = 1.0 / (double) num_steps;
9
10     for (i = 1; i <= num_steps; i++) {
11         x = (i - 0.5) * step;
12         sum = sum + 4.0 / (1.0 + x * x);
13     }
14     pi = step * sum;
15 }
```

Say the goal is to parallelize the code. The `for` loop can be parallelized. How would the `#pragma` construct affect the data sharing? Variables `i`, `x`, and `sum`, are accessed and written in the loop, and, by default, these variables are shared. Not having a proper data sharing design would alter the result.

As opposed to the other variables, `sum` is only read just before writing it by means of an addition. Additions are commutative ($3+2=2+3$) and associative ($(2+3)+4=2+(3+4)$), so `sum` can be shared. If `sum` was to be private, the initial value would be undefined, this is, the sum would not start from 0. If it was to be firstprivate, the results of each thread would need to be collected somehow after the fact.

Meanwhile, different values of `i` and `x` can be read in crucial moments of the calculation. Specifically, the addition in line 12 may access a different value of `x` than the one calculated by the same thread a line before.

The solution would be to make `i` and `x` private, as shown in the following code snippet:

```
#pragma omp parallel private(i, x)
for (i = 1; i <= num_steps; i++) {
    x = (i - 0.5) * step;
    sum = sum + 4.0 / (1.0 * x * x);
```

```
}
pi = step * sum;
```

Recall that only the `for` loop will be parallelized. The last assignment (`pi = step * sum`) is run sequentially.

Some API calls

- `int omp_get_num_threads()` returns the number of threads in the current team.
- `int omp_get_thread_num()` Returns the id of the thread in the current team. Goes from 0 to `omp_get_thread_num() - 1`
-

Thread synchronization

OpenMP follows a shared memory model. Threads communicate by sharing variables. Unintended sharing of data may cause race conditions. Threads need to synchronize to impose some ordering in their sequence of actions.

Thread barrier

```
#pragma omp barrier
```

Threads cannot proceed past a barrier point until all the parallel threads reach the barrier. Some constructs, such as `parallel`, have an implicit barrier at the end.

```
1  #pragma omp parallel
2  {
3      foo();
4      #pragma omp barrier
5      bar();
6  }
```

The explicitly defined barrier in line 4 forces all threads to finish running `foo()` before running `bar()`. At the same time, the end of the parallel region at line 6 implicitly means that all threads must finish running `bar()` before the code keeps running sequentially.

Exclusive access: critical construct

```
#pragma omp critical [(name)]
    // structured block
```

Makes a parallel region accessible to only one thread at any given time, this is, mutual exclusion. Unless explicitly named, all critical regions are the same.

```
#pragma omp parallel
{
    foo();
    #pragma omp critical
    bar();
    #pragma omp critical
    baz();
}
```

In the example above, there are two different critical regions. Nevertheless, as none of them are named, only one thread can run either one of the regions. The two of them will never be run at the same time. This can be fixed by naming either of the regions, as shown below.

```
#pragma omp parallel
{
    foo();
    #pragma omp critical part1
    bar();
    #pragma omp critical part2
    baz();
}
```

Exclusive access: atomic construct

```
#pragma omp atomic [ update | read | write ]
    // expression
```

The construct ensures that a specific storage location is accessed in a mutually exclusive way, avoiding the possibility of multiple, simultaneous reading and writing threads. It is usually more efficient than a `critical` construct. There are three types of atomic accesses:

- Updates: `x++, x -= foo()`. An operation that reads from and writes in the same memory space. (These are those operations that can be represented as `+=`, `--`, `<<=` or `/=`, among others)
- Reads: `value = *p`. In this case, the value of `p` is being directly read.
- Writes: `*p = value`. In this case, the value of `p` is being directly written on.

Reduction clause

```
#pragma omp parallel [...] reduction(operator:variable)
    // block...
```

Reduction is a pattern where all threads accumulate values into a single variable. Valid operators are `+`, `-`, `*`, `|` (bitwise OR), `||` (logical OR), `&` (bitwise AND), `&&` (logical AND) and `^` (bitwise XOR). The compiler implicitly creates a properly initialized private copy of the variable for each thread and, at the end of the region, it takes care of safely updating it with the partial solutions.

```
#pragma omp parallel private(x, i, id) reduction(+:sum)
{
    id = omp_get_thread_num();
    for (i = id + 1; i <= num_steps; i += NUM_THREADS) {
        x = (i - 0.5) * step;
        sum = sum + 4.0 / (1.0 + x * x);
    }
}
pi = sum * step;
```

In the example above, each thread will have a private copy of `sum`, probably initialized to 0. Threads will calculate the sum of the respective iterations they run, depending on their ID. As additions are commutative and associative, the results can be aggregated after the barrier without any loss of information.

Locks

OpenMP provides lock primitives for low-level synchronization. Locks work much like critical regions. They are acquired before entering a mutual exclusion region and must be released afterwards.

```
1  #include <omp.h>
2  void foo() {
3      omp_lock_t lock;
4      omp_init_lock(&lock);
5      #pragma omp parallel
6      {
7          omp_set_lock(&lock);
8          // mutual exclusion region
9          omp_unset_lock(&lock);
10     }
11     omp_destroy_lock(&lock);
12 }
```

Note, in the example above, that locks need initialization and destruction, which is taken care of with the functions on lines 4 and 11, respectively.

Memory consistency

```
#pragma omp flush (list)
```

It enforces consistency between the temporary view and memory for those variables in the list. Synchronization constructs (implicit or explicit) have an associated flush operation.

Loop parallelism

The worksharing concept

Worksharing constructs divide the execution of a code region among the members of a team. Threads cooperate to do some work. It is a better way to split work than thread IDs, and has a lower overhead than tasks, even though it's less flexible.

The `for` construct

```
#pragma omp for [clauses]
    for (init-expr; test-expr; inc-expr)
```

The iterations of the loops will be divided among the threads. Loop iterations must be independent and it must follow a shape that allows induction of amount of iterations. Valid types for inductions are integers, pointers and random access iterators (C++). This inducted variable is private.

```
void main() {
    int i, id;
    double x, pi, sum;

    step = 1.0 / (double) num_steps;
    #pragma omp parallel for private(x) reduction(+:sum)
    for (i = 1; i <= num_steps; i++) {
        x = (i - 0.5) * step;
        sum  = sum + 4.0 / (1.0 + x * x);
    }
    pi = sum * step;
}
```

The `for` construct automatically detects `i` as the variable for iteration control. OpenMP makes it private and then distributes iterations among the threads. `x` still must be marked as private, race conditions can still occur. `sum` will have a different copy for each of the threads, initialized to 0. Thanks to the `reduct` clause, all the local results of `sum` per thread will be added after the synchronization.

The `schedule` clause

This clause determines which iterations are executed by each thread. If no clause is present, the implementation should take care of this. There are several options:

- `static`. The iteration space is broken in chunks of size $\frac{N}{\text{num\_threads}}$. This is, the iterations are evenly divided across the threads. Chunks are assigned to threads in Round-Robin fashion.
- `static,N` (interleaved). The iteration space is broken in chunks of size N. Then this chunks are scheduled to threads in Round-Robin fashion.

The overall characteristics of static scheduling are low overhead, (usually) good locality and the possibility of load imbalance problems.

- `dynamic,N`. Threads dynamically grab chunks of N iterations until all iterations have been executed. $N = 1$ by default.
- `guided,N`. The size of the chunks decreases as the threads grab iterations, but it is at least of size N, $N = 1$ by default.

These dynamic schedules result in higher overhead, not very good locality (usually) but they can solve imbalance problems.

The `nowait` clause

The `nowait` clause removes the implicit `barrier` from the end of a parallel region. This allows to overlap the execution of non-dependent loops/tasks/worksharings.

```
#pragma omp for nowait
for (i = 0; i < n; i++)
    v[i] = 0;
#pragma omp for
for (i = 0; i < n; i++)
    a[i] = 0;
```

In the example above, the work of the second loop is independent from the work of the first loop, hence there is no need for the threads to wait for synchronization after running the first loop. The `nowait` construct allows this.

```
#pragma omp for schedule(static, 2) nowait
for (i = 0; i < n; i++)
    v[i] = 0;
#pragma omp for schedule(static, 2)
for (i = 0; i < n; i++)
    a[i] = v[i] * v[i];
```

In the example above, a static scheduling policy is defined. 2 different chunks will be created with the halves of the iterations of each `for` loop. So, even though the second loop is directly dependent on the results of the first loop, each of the threads will be dependent only of the elements calculated in the iterations run by themselves. The `nowait` construct allows the threads to continue running when they finish with the first loop.

The `collapse` clause

The `collapse` clause allows to distribute work from a set of n nested loops. The loops must be perfectly nested. The nest must traverse a rectangular iteration space.

```
#pragma omp for collapse(2)
for (i = 0; i < N; i++)
    for (j = 0; j < M; j++)
        foo(i, j);
```

In the example above, the loops of i and j are folded and iterations distributed among all threads, and both variables are privatized.

The `single` construct

```
#pragma omp single [clauses]
    // structured block
```

The construct makes the structured block to be run in only one thread. The clauses can be `private`, `firstprivate` and `nowait`. There is an implicit `barrier` at the end.

Task parallelism

Task parallelism model

Tasks are work units whose execution may be deferred, but they can also be executed immediately. Threads, separated in teams, cooperate to execute them.

Task creation

Parallel regions create tasks. One implicit task is created and assigned to each thread. Each thread that encounters a `task` construct packages the code and data and creates a new explicit task.

Explicit task creation

```
#pragma omp task [clauses]
    // structured block
```

Where some possible clauses are `shared`, `private`, `firstprivate`, `if(expr)`, `final(expr)` and `mergeable`.

```
void traverse_list(List l) {
    Element e;
    for (e = l->first; e; e = e -> next) {
        #pragma omp task
        process(e); // e is firstprivate by default
    }
}
```

The code above defines a task block, the `process` function. But tasks are useless if they are not defined within a parallel region. Let's complete the code.

```
List l;

#pragma omp parallel
traverse_list(l);

void traverse_list(List l) {
    // ...
}
```

In the code above the `traverse_list` function is going to be run by as many threads as defined. All of the threads will run all of the tasks, so the traversal of the list will be calculated by all threads. Execution is not actually parallelized.

```
List l;

#pragma omp parallel
#pragma single
traverse_list(l);

void traverse_list(List l) {
    // ...
}
```

With the addition of the construct `single`, only one of the threads will proper run the `traverse_list` function. This thread will create the tasks. The rest of threads (and the first thread, once the task generation is complete) will run the tasks in parallel.

Default task data-sharing attributes

When no data clauses are specified, global variables are shared, variables declared within the scope of a task are private, and the rest are firstprivate, except when a `shared` attribute is inherited.

```
int a;
void foo() {
    int b, c;
    #pragma omp parallel shared(b)
    #pragma omp parallel private(b)
    {
        int d;
        #pragma omp task
        {
            int e;
        }
    }
}
```

In the code above, the variables are:

- `a` is global, and so shared by default.
- `b` is firstprivate.
- `c` is shared because, in the context of the task, it is effectively global.
- `d` is firstprivate, because it is declared within the parallel region and it its attribute is not explicitly defined.
- `e` is private, because it is declared withing the scope of the task.

Immediate task execution

The `if` clause

When an `if` clause is present on a task construct, and its expression evaluates to `false`, an undeferred task is generated. The encountering thread must suspend the current task region, the execution of which cannot be resumed until the generated task is completed. This allows implementations to optimize task creation.

The `final` clause

If the expression of a `final` clause evaluates to `true`, then the generated task and its children will be final and included. Execution of included tasks is done immediately after the generating task. So, all tasks created within a `final` region will be run sequentially and immediately, and if more tasks are created, they will also be final and have the same final task generating capabilities.
When a `mergeable` clause is present on a task construct, and the generated task is an included task, the implementation may generate a merged task instead (i.e. no task and context creation for it).

Task synchronization

There are two types of task barriers:

- `taskwait`: Suspends the current task waiting on the completion of child tasks of the current task. This construct is stand-alone.
- `taskgroup`: Suspends the current task at the end of a structured block waiting on completion of child tasks of the current task and their descendent tasks.

`taskwait`

```
#pragma omp task {}     // T1
#pragma omp task        // T2
{
    #pragma omp task {} // T3
}
#pragma omp task {}     // T4

#pragma omp taskwait
```

With the code above, only tasks 1, 2 and 4 are guaranteed to have finished after `taskwait`, as it only guarantees the completion of child tasks of the current task.

```
int fib(int n) {
    int i, j;

    if (n < 2) return n;

    #pragma omp task shared(i) final(n <= THOLD) mergeable
    i = fib(n - 1);
    #pragma omp task shared(j) final(n <= THOLD) mergeable
    j = fib(n - 2);

    #pragma omp taskwait
    return i + j;
}
```

`taskgroup`

```
#pragma omp task {}         // T1
#pragma omp taskgroup {
    #pragma omp task        // T2
    {
        #pragma omp task {} // T3
    }
    #pragma omp task {}     // T4
}
```

With the code above, only tasks 2 to 4 are guaranteed to have finished after the `taskgroup` clause, as it guarantees the completion of all child tasks, and their child tasks recursively.

Data sharing inside tasks

In addition one can use `critical` and `atomic` to synchronize the access to shared data inside tasks.

```
void process(Element e) {
    // ...
    #pragma omp atomic
    solutions_found++;
    // ...
}
```

Task dependencies

Dependence between sibling tasks can be expressed as follows:

```
#pragma omp task    [depend (in : var_list)]
                    [depend (out : var_list)]
                    [depend (inout : var_list)]
```

The exact dependence between tasks is inferred from the dependence type and the items in the variable list. This list may include array sections.

- Tasks with the `in` dependence-type will be dependent of all previously generated sibling tasks that reference, at least, one of the items in the variable list in an `out` or `inout` dependence-type list.
- Tasks with the `out` or `inout` dependence-types will be dependent on all the previously generated tasks mentioning, at least, one of the items in the variable list.

```
#pragma omp parallel private(i, j)
#pragma omp single
{
    for (i = 1; i < n; i++) {
        for (j = 1; j < n; j++) {
            #pragma omp task
                    depend(in : block[i - 1],
                     ↪  block[i][j - 1])
                    depend(out : block[i][j]
        foo(i, j);
        }
    }
}
```

Unai Perez Mendizabal ©https://github.com/unaipme

CPDS: MPI cheatsheet

Basic concepts

- **Communicator**: It is the context for a communication operator. Messages are received within the context from where they were sent. Messages sent to different contexts do not interfere. The global context is `MPI_COMM_WORLD`
- **Process group**: Set of processes that share a communication context.

Main environmental functions

MPI_INIT

```
int MPI_Init(int * argc_ptr,     // in
             char** argv_ptr[] ); // in
```

It initializes the MPI environment. All MPI programs must call this routine once and only once before any other MPI routines.

MPI_COMM_SIZE

```
int MPI_Comm_size(MPI_Comm comm, // in
                  int* size );     // out
```

It returns the number of processors (`size`) in the group associated to communicator `comm`.

MPI_COMM_RANK

```
int MPI_Comm_rank(MPI_Comm comm, // in
                  int* rank );     // out
```

It returns the identifier of the local process in the group associated with communicator `comm`. The identifier (`rank`) of the process is within range [0, size-1].

MPI_FINALIZE

```
int MPI_Finalize(void);
```

It terminates all MPI processing. This routine MUST be the last MPI call, all pending communications involving a process must have completed before the process calls `MPI_FINALIZE`.

MPI_ABORT

```
int MPI_Abort(MPI_Comm comm, // in
              int errorcode ); // in
```

Forces all processes of an MPI job to terminate.

Barrier synchronization

```
int MPI_Barrier(MPI_Comm comm); // in
```

Blocks all processes in communicator `comm`'s context until all processes have reached the point and called the barrier.

```
#include "mpi.h"

int rank;
int nproc;

int main(int argc, char* argv[] ) {
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // Ensure all processes arive here before timing
    MPI_Barrier(MPI_COMM_WORLD);
    // Something to do in local

    MPI_Finalize();
}
```

Point-to-point communication

Blocking operations

Return form the procedure indicates the user is allowed to reuse resources specified in the call. In other words, the execution will wait for the communication to complete.

MPI_SEND

```
int MPI_Send(void* buf,              // in
             int count,              // in
             MPI_Datatype datatype,  // in
             int destination,        // in
             int tag,                // in
             MPI_Comm comm );        // in
```

Performs a blocking send operation. The message can be received by either `MPI_RECV` (blocking call) or `MPI_IRECV` (non-blocking call). The three last parameters (`destination`, `tag` and `comm`) are called the message envelope; it is the information used to distinguish messages and selectively receive them.

MPI_RECV

```
int MPI_Recv(void* buf,              // out
             int count,              // in
             MPI_Datatype datatype,  // in
             int source,             // in
             int tag,                // in
             MPI_Comm comm,          // in
             MPI_Status* status );   // out
```

Performs a blocking receive operation. The message received must be less than or equal to the length of the receive buffer `buf`. It can receive a message sent by either `MPI_SEND` or `MPI_ISEND`. The message envelope this time is `source`, `tag` and `comm`.

Example

```
#include "mpi.h"

int rank, nproc;

int main(int argc, char* argv []) {
    int isbuf, irbuf;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        isbuf = 9;
        MPI_Send(&isbuf, 1, MPI_INTEGER, 1, 1,
         ↪  MPI_COMM_WORLD);
    } else if (rank == 1) {
        MPI_Recv(&irbuf, 1, MPI_INTEGER, 0, 1,
         ↪  MPI_COMM_WORLD, &status);
        printf("%d\n", irbuf);
    }
    MPI_Finalize();
}
```

MPI_SENDRECV

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

Sends and receives a message in a blocking manner.

Non-blocking operations

The procedure may return before the operation completes, and before the user is allowed to reuse resources specified in the call. In other words, when the procedure is called, the execution continues, and it is up to the developer to check for the reception of the message.

MPI_ISEND

```
int MPI_Isend(void* buf,              // in
              int count,              // in
              MPI_Datatype datatype,  // in
              int dest,               // in
              int tag,                // in
              MPI_Comm comm,          // in
              MPI_Request* request ); // out
```

Performs a non-blocking send operation. `request` is an identifier for later enquiry with `MPI_WAIT` or `MPI_TEST`. The send buffer `buf` must not be modified until the request has been completed by `MPI_WAIT` or `MPI_TEST`. The message can be received by either `MPI_RECV` or `MPI_IRECV`.

MPI_IRECV

```
int MPI_Irecv(void* buf,              // out
              int count,              // in
              MPI_Datatype datatype,  // in
              int source,             // in
              int tag,                // in
              MPI_Comm comm,          // in
              MPI_Request* request ); // out
```

Performs a non-blocking receive operation. The receive buffer `buf` must not be accessed until the receive is completed by `MPI_WAIT` or `MPI_TEST`. The message received must be less than or equal to the length of the receive buffer `buf`. The function can receive a message sent by either `MPI_SEND` or `MPI_ISEND`.

MPI_WAIT

```
int MPI_Wait(MPI_Request* request,  // inout
             MPI_Status* status );  // out
```

Waits for a non-blocking operation to complete, with identifier stored in `request`. Information on the completed operation is found in `status`. If wildcards (`MPI_ANY_SOURCE`, `MPI_ANY_TAG`) were used by the recieve for either the source or tag, the actual source and tag can be retrieved from `status->MPI_SOURCE` and `status->MPI_TAG`.

MPI_TEST

```
int MPI_Test(MPI_Request* request,  // inout
             int* flag,             // out
             MPI_Status* status );   // out
```

Test for the completion of a non-blocking send or receive. `flag` equals `MPI_SUCCESS` if MPI routine completed successfully.

MPI_GET_COUNT

```
int MPI_Get_count(MPI_Status status,    // in
                  MPI_Datatype datatype,  // in
                  int* count );           // out
```

Returns the number of elements in a message (indicated by `status`). The `datatype` argument and the argument provided by the call that set the `status` variable should match.

MPI_PROBE

```
int MPI_Probe(int source,           // in
              int tag,              // in
              MPI_Comm comm,        // in
              MPI_Status *status );  // out
```

Blocking call that returns only after a matching message is found. Wildcards can be used to wait for messages coming from any source (`MPI_ANY_SOURCE` or with any tag (`MPI_ANY_TAG`). There also is a non-blocking `MPI_Iprobe`.

Example

```
#include "mpi.h"

int main(int argc, char* argv[]) {
    int rank, nproc;
    int isbuf, irbuf, count;
    MPI_Request request;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        isbuf = 9;
        MPI_Isend(&isbuf, 1, MPI_INTEGER, 1, 1,
         ↪  MPI_COMM_WORLD, &request);
    } else if (rank == 1) {
        MPI_Irecv(&irbuf, 1, MPI_INTEGER,
         ↪  MPI_ANY_SOURCE, MPI_ANY_TAG,
         ↪  MPI_COMM_WORLD, &request);
        // Other work to do
        MPI_Get_count(status, MPI_INTEGER, &count);
        printf("irbuf = %d, source = %d, tag = %d,
         ↪  count = %d\n", irbuf, status.MPI_SOURCE,
         ↪  status.MPI_TAG, count);
    }
    MPI_Finalize();
}
```

MPI data types

`MPI_Datatype` can be one of the following: `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`.

Collective communication

If all processes in a process group need to invoke the procedure.

MPI_BCAST

```
int MPI_Bcast(void* buffer,          // inout
              int count,             // in
              MPI_Datatype datatype, // in
              int root,              // in
              MPI_Comm comm );       // in
```

It broadcasts a message from `root` to all processes in communicator `comm`. The type signature of `count` and `datatype` on any process must be equal to the type signature of `count` and `datatype` at the `root`.

MPI_REDUCE

```
int MPI_Reduce(void* sendbuf,        // in
               void recvbuf,         // out
               int count,            // in
               MPI_Datatype datatype, // in
               MPI_Op op,            // in
               int root,             // in
               MPI_Comm comm);       // in
```

Applies a reduction operation to the vector `sendbuf` over the set of processes specified by communicator `comm` and places the result in `recvbuf` on `root`. Both the input and output buffers have the same number of elements with the same type.
Users may define their own operations or use the predefined operations provided by MPI: `MPI_{SUM, PROD, MAX, MIN, MAXLOC, MINLOC, LAND, LOR, LXOR, BAND, BOR, BXOR}`.

Broadcast and reduce: Example

```
void main(int argc, char* argv[]) {
    int i, rank, nproc, num_steps;
    double x, pi, step, sum = 0.0;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);

    if (rank == 0) {
        scanf("%d", &num_steps);
    }
    MPI_Bcast(&num_steps, 1, MPI_INT, 0,
     ↪  MPI_COMM_WORLD);
    step = 1.0 / (double) num_steps;
    my_steps = num_steps / nproc;

    for (i = rank * my_steps; i < (rank + 1) *
     ↪  my_steps; i++) {
        x = (i + 0.5) * step;
        sum += 4.0 / (1.0 + x * x);
    }
    sum *= step;
    MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
     ↪  MPI_COMM_WORLD);
    MPI_Finalize();
}
```

In the code above, only the thread with rank 0 will allow for user input. This user input is the number of steps (iterations). Then, all threads will run the `MPI_Bcast` function, with thread ranked 0 as the root. This means that thread of rank 0 will send all the threads its value of the variable `num_steps` and those will store it in their `num_steps` variable.
The `for` loop will let each thread to run a certain amount of iterations. Each thread will have its share of `sum`. With function `MPI_Reduce`, all threads will send their `sum` values to the root (again, thread with rank 0) and, using operation `MPI_SUM`, which corresponds to a sum, it will aggregate into one variable.

MPI_SCATTER

```
int MPI_Scatter(void* sendbuf,          // in
                int sendcount,          // in
                MPI_Datatype sendtype,  // in
                void* recvbuf,          // out
                int recvcount,          // in
                MPI_Datatype recvtype,  // in
```

```
                int root,               // in
                MPI_Comm comm );        // in
```

It distributes individual messages from `root` to each process in the communicator. It represents the inverse operation to `MPI_GATHER`.

MPI_GATHER

```
int MPI_Gather(void* sendbuf,       // in
               int sendcount,       // in
               MPI_Datatype sendtype, // in
               void* recvbuf,       // out
               int recvcount,       // in
               MPI_Datatype recvtype, // in
               int root,            // in
               MPI_Comm comm );     // in
```

Collects individual messages from each process in the communicator to the `root` process and stores them in rank order.

Scatter and gather: Example

```
int gsize, localbuf[100];
int root = 0, rank, nproc, *rootbuf;
// ...
MPI_Comm_size(MPI_COMM_WORLD, &nproc);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank == root)
    rootbuf = (int *) malloc(nproc * 100 *
     ↪  sizeof(int));
// matrix is initialized in root
MPI_Scatter(rootbuf, 100, MPI_INT, localbuf, 100,
 ↪  MPI_INT, root, comm);
// do work with data
MPI_Gather(localbuf, 100, MPI_INT, rootbuf, 100,
 ↪  MPI_INT, root, comm);
// results back in root
```

In the example above, the root thread will have (supposedly) initialized a matrix, with 100 elements for each thread. These elements, stored in the `rootbuf`, are distributed, hundred by hundred, into the other threads' `localbuf` variable. After some work on the data, the content of variable `localbuf` of all threads is stored back into the root thread's `rootbuf` variable.

MPI_ALLGATHER

```
int MPI_Allgather(void* sendbuf,       // in
                  int sendcount,       // in
                  MPI_Datatype sendtype, // in
                  void* recvbuf,       // out
                  int recvcount,       // in
                  MPI_Datatype recvtype, // in
                  MPI_Comm comm );     // in
```

Does the same as `MPI_Gather`, but all threads receive the resulting message.

Other variations

MPI_SCATTERV

Works the same way as `MPI_SCATTER`, but permits the user to define how many elements will go to each thread, through an integer array with size `nproc`.

MPI_GATHERV

The exact opposite of `MPI_SCATTERV`, all results are scattered into the root, but it allows to set how many of the elements will be received from each thread.

MPI_ALLGATHERV

Permits to define how many elements will be gathered from each thread, as does `MPI_GATHERV`, but all threads will receive the resulting message.

MPI_ALLTOALL

```
int MPI_Alltoall(void* sendbuf,       // in
                 int sendcount,       // in
                 MPI_Datatype sendtype, // in
                 void* recvbuf,       // out
                 int recvcount,       // in
                 MPI_Datatype recvtype, // in
                 MPI_Comm comm );     // in
```

Sends a distinct message from each process to every other process. The `j`th block of data sent from process `i` is received by process `j` and placed in the `i`th block of the buffer `recvbuf`.

MPI_ALLTOALLV

As does `MPI_Alltoall`, sends a message from each process to all processes, but allows to specify the amount of elements to send to each thread.

CPDS: CUDA cheatsheet

Devices, kernel definition and offloading

Device management

The application can query and select the GPUs.

- `cudaGetDeviceCount(int *count)`
- `cudaSetDevice(int device)`
- `cudaGetDeviceProperties(cudaDeviceProp *prop, int device)`
- `cudaGetDevice(int *device)`

Offloading kernel execution

A kernel is a highly parallel function that runs on the device and is called from host. It is defined using the keyword `__global__`.

```
__global__ void helloworld(void) {
}
```

The code shown below offloads the kernel on GPU device, this is, relegates the execution of the kernel function to the said device.

```
helloworld<<<1,1>>>();
```

The kernel launch is asynchronous, the host does not wait for the termination of the execution. However, the host may need to wait before continuing execution. Function `cudaThreadSynchronize()`[1] blocks the CPU until all CUDA kernels have completed.

```
__global__ void helloworld(void) {
}

int main(void) {
    helloworld<<<1,1>>>();
    cudaThreadSynchronize();
    printf("Hello, world!\n");
    return 0;
}
```

Offloading to multi-GPU nodes

Any host thread can access all GPUs in the node (if more than one available) using the `cudaSetDevice` call.

```
int main(void) {
    int numDevs;
    cudaGetDeviceCount(&numDevs);

    for (int d = 0; d < numDevs; d++) {
        cudaSetDevice(d);
        helloworld<<<1,1>>>();
    }
    for (int d = 0; d < numDevs; d++) {
        cudaSetDevice(d);
        cudaThreadSynchronize();
    }
    return 0;
}
```

[1] Deprecated, to be replaced by `cudaDeviceSynchronize()`

Blocks threads and indexing

Parallel kernel

Kernel replication to use multiple **streaming multiprocessors** (SM) using the arguments in the kernel launch:

```
// Replication in different SM
helloworld<<<N, 1>>>();
// Replication in different threads inside one SM
helloworld<<<1, N>>>();
```

Each kernel instance can be unequivocally identified, which can be used to make kernel instances cooperate. This can also be done at the threads level.

```
__global__ void helloWorld(char* str) {
    // Kernel instance level
    int idx = blockIdx.x;
    // Thread level
    int idx = threadIdx.x;

    str[idx] += idx;
}
```

Indexing vectors with blocks and threads

Consider indexing an array with one element per thread and M threads per block (e.g. 8). Then a unique index for each thread is given by `int index = threadIdx.x + blockIdx.x * M`.

Example: Vector addition

The sequential code for a vector addition would look like this:

```
// Compute vector sum C = A + B
void vecAdd(float* a, float* b, float* c, int N) {
    for (int i = 0; i < N; i++)
        c[i] = a[i] + b[i];
}

int main() {
    float a[N], b[N], c[N];
    // initialize a and b
    vecAdd(a, b, c, N);
    // display the results
    return 0;
}
```

The code for the vector addition with kernel definition and invocation would look like this:

```
// Each kernel invocation performs one pair-wise
↪  addition
__global__ void vecAddKernel(float* a_d, float* b_d,
↪  float* c_d, int N) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    c_d[i] = a_d[i] + b_d[i];
}

__host__ int vectAdd(float* a, float* b, float* c, int
↪  N) {
    vecAddKernel<<<ceil(N / 256), 256>>>(a, b, c, N);
}
```

In the example above, the vectors are split in blocks of 256 threads and equally distributed among as many blocks as needed. This means that, if the vectors are of size 512, two SM will run blocks of 256 threads.
The `__global__` function will be run on each of the threads, which will take care of just making the corresponding addition. The `__host__` function will be called on the host.

Function declarations in CUDA

- `__device__` functions are executed on the device and only callable from the device.
- `__global__` functions (also called kernel functions) are executed on the device and are only callable from the host. They can only return void.
- `__host__` functions are executed and callable only on the host.

Example: vector addition (revisited)

What if `blockDim.x` does not divide by N?

```
__global__ void vecAddKernel(float* a_d, float* b_d,
↪  float* c_d, int N) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i < n)
        c_d[i] = a_d[i] + b_d[i];
}
```

Identifying threads inside a grid

In general, the GPU offers a grid of threads, divided into thread blocks, and each block divided into threads. The grid of thread blocks can actually be partitioned into 1, 2, or 3 dimensions.
Each thread uses indices (1D, 2D or 3D) to decide what to do and what data to work on (data decomposition). These indices are `blockIdx` and `threadIdx`, defined inside a grid with `gridDim` blocks, each with `blockDim` threads.
This is, the CUDA grid will have dimensions `gridDim.x` times `gridDim.y`. Inside the grid, there will be thread blocks with `blockDim.x` times `blockDim.y` threads.

dim3 datatype

Data type to define variables to hold block and grid dimensionalities.

```
__host__ int vectAdd(float* a, float* b, float* c, int
↪  N) {
    // Run ceil(N/256) blocks of 256 threads each
    dim3 DimGrid(ceil(N/256), 1, 1);
    dim3 DimBlock(256, 1, 1);
    vecAddKernel<<<DimGrid, DimBlock>>>(a, b, c, N);
}
```

Thread scheduling and execution

Threads in a block are divided in 32-thread warps (scheduling units in SM). For example, if 3 blocks assigned to an SM, each with 256 threads, how many warps are there in an SM?

- Each block is divided into 256 (threads) / 32 (warp size) = 8 warps per block
- There are 8 (warps per block) × 3 (blocks) = 24 warps in total.

At any point in time, only one of the 24 warps will be selected for instruction fetch and execution (multithreading).

Accessing (global) memory

Partial view of the device memory

Device code can read and write thread registers and grid global memory.
Host code can allocate data in grid global memory and transfer data between host and grid global memories.

Basic device memory management

Global memory contents are visible to all threads. Variables in grid global memory can be declared using `__device__` attribute. It is also dynamically allocatable using `cudaMalloc` and `cudaFree`.

```
#define N 1000

// declared outside function and kernel bodies.
__device__ int A[N];

__global__ kernel() {
    int id = blockId.x * blockDim.x + threadId.x;
    A[tid]++;
}
```

- `cudaMalloc` allocates an object in the device global memory. It takes two parameters, pointer to the allocated object and its size in bytes.
- `cudaFree` frees an object from device global memory. It takes the pointer to the object as only parameter.

Example: vector addition (host code)

```
int main() {
    // Arrays in host memory
    float a[N], b [N], c[N];
    // Pointers to vectors dynamically allocated in
     ↪  global memory
    float *dev_a, *dev_b, *dev_c;

    cudaMalloc(&dev_a, N * sizeof(float));
    cudaMalloc(&dev_b, N * sizeof(float));
    cudaMalloc(&dev_c, N * sizeof(float));

    // ...

    // kernel invocation
    dim3 DimGrid(ceil(N/256, 1, 1);
    dim3 DimBlock(256, 1, 1);
    vecAddKernel<<<DimGrid,DimBlock>>>(dev_a, dev_b,
     ↪  dev_c, N);

    // ...

    // Free the memory allocated on device
    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);

    return 0;
}
```

Basic device memory management (cont.)

- `cudaMemcpy)` is a synchronous function for data transfer. It blocks the CPU until it is complete and the copy does not begin until all preceding CUDA calls have completed. It requires 4 parameters: pointer to destinations, pointer to source, number of bytes to be copied and type of transfer.
- `cudaMemcpyAsync` is the asynchronous version of the function, allowing the overlap of data transfer and computation. It also uses an additional `stream` parameter that, if not 0, can be used to synchronize.

Vector addition (host code)

```
int main() {
    // Arrays in host memory
    float a[N], b [N], c[N];
    // Pointers to vectors dynamically allocated in
     ↪  global memory
    float *dev_a, *dev_b, *dev_c;

    cudaMalloc(&dev_a, N * sizeof(float));
    cudaMalloc(&dev_b, N * sizeof(float));
    cudaMalloc(&dev_c, N * sizeof(float));

    // Initialize a and b in host memory (as
     ↪  sequential)

    // Copy a and b from host to device memory
    cudaMemcpy(dev_a, a, N * sizeof(float),
     ↪  cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, b, N * sizeof(float),
     ↪  cudaMemcpyHostToDevice);

    dim3 DimGrid(ceil(N/256), 1, 1);
    dim3 DimBlock(256, 1, 1);
    vecAddKernel<<<DimGrid,DimBlock>>>(dev_a, dev_b,
     ↪  dev_c, N);

    // Copy back c from device to host
    cudaMemcpy(c, dev_c, N * sizeof(float),
     ↪  cudaMempcyDeviceToHost);

    // Display results

    // Free the memory allocated on device
    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);

    return 0;
}
```

Sharing data between GPUs

There are three options to acomplish this.

- One is to make copies explicitly from the host code.
- Another option is the peer-to-peer memory access. This can be done with the direct copy from pointer on one device to pointer on another device, via the function `cudaMemcpyPeer(void* dst, int dstDevice, void* src, int srcDevice, size_t count)`.
- The last option is the zero-copy shared host array, allowing the direct access to host memory from the device (Not covered).

Cooperating threads and shared memory

Threads vs. blocks

Unlike blocks, threads have mechanisms to synchronize and communicate via memory. The synchronization can be achieved with the instruction `__syncthreads()`, which forces all warps (i.e. threads in a block) to wait until they reach the same point.

A more complete view of the device memory

Each thread can:

- R/W per-thread registers and local memory.
- R/W per-block shared memory (100 times faster than global).
- R/W per-grid global memory.
- Read only per-grid constant and texture memories.

Host code can R/W global, constant and texture memories.
Shared memory is shared and allocated within a block, used to share data among threads and is not visible to threads in other blocks. It is declared using `__shared__`.

```
#define N 1000

__global__ kernel() {
    __shared__ int A[N];
    int tid = threadId.x;
    A[tid]++;
}
```

Example: 1D stencil

Consider applying a stencil to a 1D array of elements. Each output element is the sum of input elements within a radius. Each thread processes one output element. Each input element is read $2 \times radius + 1$ times.
Read `blockDim.x + 2 * radius` input elements from global memory to shared memory. Compute `blockDim.x` output elements. Write `blockDim.x` output elements to global memory.