Computational Astrophysics: The practical side

today:

02: Parallel Programming Models

S.R. Knollmann, UAM

last updated: 16.11.2010

Overview

- Overview
- Short introduction to OpenMP
  - what is it
  - how to use it
- Short introduction to MPI
  - what is it
  - how to use it
- And even more...
  - Different models
  - Accelerators

Why parallel?

There are generally two (non-exclusive) reasons to search for parallel methods to solve a problem:

- it takes too long
  - expensive computations
  - deeply nested loops
  - many iterations
- it requires too many resources (→ RAM)
  - large data set
  - high resolution required

Other reasons may include:

- because we can
- because the problem is inherently parallel (*embarrassingly parallel*)

Why parallel?

There are generally two (non-exclusive) reasons to search for parallel methods to solve a problem:

- it takes too long
  - expensive computations
  - deeply nested loops
  - many iterations
- it requires too many resources ($\rightarrow$ RAM)
  - large data set
  - high resolution required

Other reasons may include:

- because we can
- because the problem is inherently parallel (*embarrassingly parallel*)

Why parallel?

There are generally two (non-exclusive) reasons to search for parallel methods to solve a problem:

- it takes too long
  - expensive computations
  - deeply nested loops
  - many iterations
- it requires too many resources (→ RAM)
  - large data set
  - high resolution required

Other reasons may include:

- because we can
- because the problem is inherently parallel (*embarrassingly parallel*)

Why parallel?

There are generally two (non-exclusive) reasons to search for parallel methods to solve a problem:

- it takes too long
  - expensive computations
  - deeply nested loops
  - many iterations

- it requires too many resources (→ RAM)
  - large data set
  - high resolution required

Other reasons may include:
- because we can
- because the problem is inherently parallel (*embarrassingly parallel*)

Why parallel?

There are generally two (non-exclusive) reasons to search for parallel methods to solve a problem:

- it takes too long
  - expensive computations
  - deeply nested loops
  - many iterations
- it requires too many resources (→ RAM)
  - large data set
  - high resolution required

Other reasons may include:

- because we can
- because the problem is inherently parallel (*embarrassingly parallel*)

Architectures

Machine hierarchy:

- core (simplified: can execute a machine instruction per step)
  - cache
- CPU (contains one to many cores)
  - cache
- node (contains one to many CPUs)
  - cache
  - main memory
  - node local hard disk
  - remote hard disks
- rack (contains one to many nodes)
- machine (contains one to many racks)

Architectures

Machine hierarchy:

- core (simplified: can execute a machine instruction per step)
  - cache
- CPU (contains one to many cores)
  - cache
- node (contains one to many CPUs)
  - cache
  - main memory
  - node local hard disk
  - remote hard disks
- rack (contains one to many nodes)
- machine (contains one to many racks)

Architectures

Machine hierarchy:

- core (simplified: can execute a machine instruction per step)
  - cache
- CPU (contains one to many cores)
  - cache
- node (contains one to many CPUs)
  - cache
  - main memory
  - node local hard disk
  - remote hard disks
- rack (contains one to many nodes)
- machine (contains one to many racks)

Architectures

Machine hierarchy:

- core (simplified: can execute a machine instruction per step)
  - cache
- CPU (contains one to many cores)
  - cache
- node (contains one to many CPUs)
  - cache
  - main memory
  - node local hard disk
  - remote hard disks
- rack (contains one to many nodes)
- machine (contains one to many racks)

Architectures

Machine hierarchy:

- core (simplified: can execute a machine instruction per step)
  - cache
- CPU (contains one to many cores)
  - cache
- node (contains one to many CPUs)
  - cache
  - main memory
  - node local hard disk
  - remote hard disks
- rack (contains one to many nodes)
- machine (contains one to many racks)

Architectures

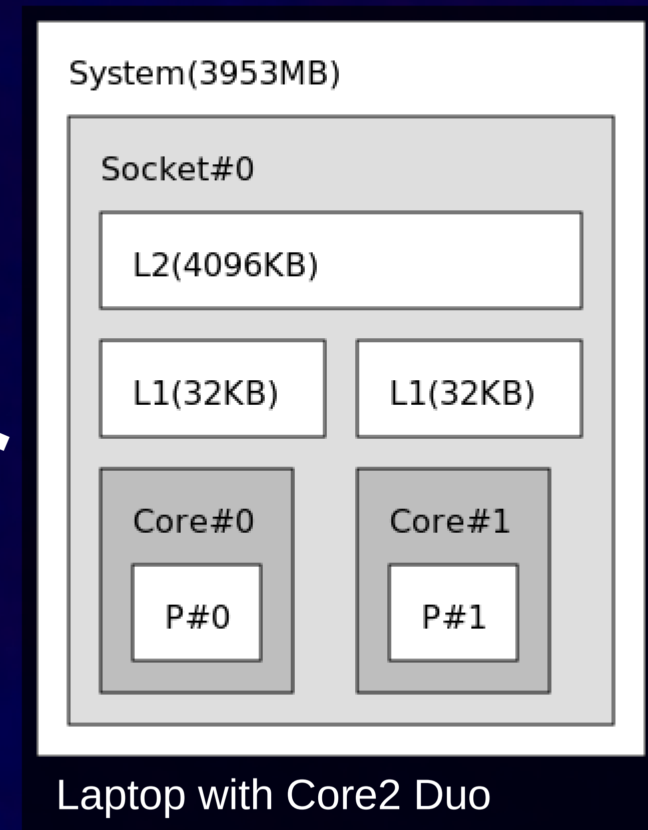
Machine hierarchy:

- core (simplified: can execute a machine instruction per step)
  - cache
- CPU (contains one to many cores)
  - cache
- node (contains one to many CPUs)
  - cache
  - main memory
  - node local hard disk
  - remote hard disks
- rack (contains one to many nodes)
- machine (contains one to many racks)

Architectures

Machine hierarchy:

- core (simplified: can execute a machine instruction per step)
  - cache
- CPU (contains one to many cores)
  - cache
- node (contains one to many CPUs)
  - cache
  - main memory
  - node local hard disk
  - remote hard disks
- rack (contains one to many nodes)
- machine (contains one to many racks)

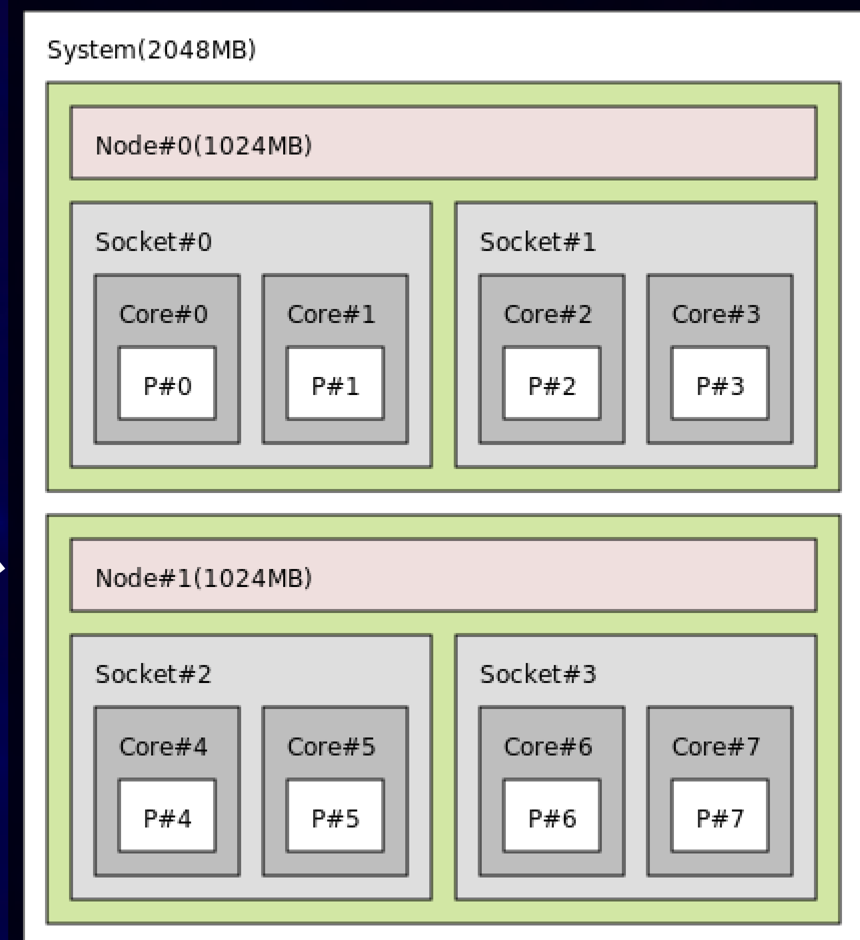


Laptop with Core2 Duo

Architectures

Machine hierarchy:

- core (simplified: can execute a machine instruction per step)
  - cache
- CPU (contains one to many cores)
  - cache
- node (contains one to many CPUs)
  - cache
  - main memory
  - node local hard disk
  - remote hard disks
- rack (contains one to many nodes)
- machine (contains one to many racks)



Rack with 2 nodes

Architectures

Machine hierarchy:

- core (simplified: can execute a machine instruction per step)
  - cache
- CPU (contains one to many cores)
  - cache
- node (contains one to many CPUs)
  - cache
  - main memory
  - node local hard disk
  - remote hard disks

local
address space

- rack (contains one to many nodes)
- machine (contains one to many racks)

Architectures

Machine hierarchy:

– core (simplified: can execute a machine instruction per step)
  – cache

– CPU (contains one to many cores)
  – cache

– node (contains one to many CPUs)
  – cache
  – main memory
  – node local hard disk
  – remote hard disks

local address space

– rack (contains one to many nodes)

– machine (contains one to many racks)

global address space

Architectures

Machine hierarchy:

- core (simplified: can execute a machine instruction per step)
  - cache
- CPU (contains one to many cores)
  - cache
- node (contains one to many CPUs)
  - cache
  - main memory
  - node local hard disk
  - remote hard disks
- rack (contains one to many nodes)
- machine (contains one to many racks)

local
address space

global
address space

Methods

We want:

- one execution path per core, to make full use of the computing capabilities
- the executions paths should be able to exchange information, either explicitly or implicitly

We can either:

- use one program which spawns multiple execution threads, i.e. every thread can see the whole memory of the program

or

- use one program (with its own independent address space) per core and have an inter-process communication method, i.e. we run the same program multiple times and those copies communicate with each other

Methods

We want:

- one execution path per core, to make full use of the computing capabilities
- the executions paths should be able to exchange information, either explicitly or implicitly

We can either:

- use one program which spawns multiple execution threads, i.e. every thread can see the whole memory of the program

or

- use one program (with its own independent address space) per core and have an inter-process communication method, i.e. we run the same program multiple times and those copies communicate with each other

Methods

We want:

- one execution path per core, to make full use of the computing capabilities
- the executions paths should be able to exchange information, either explicitly or implicitly

We can either:

- use one program which spawns multiple execution threads, i.e. every thread can see the whole memory of the program

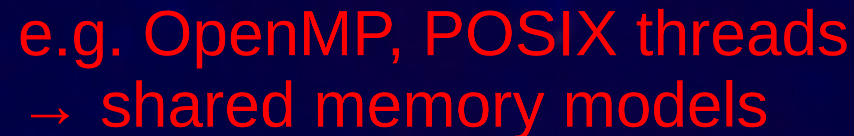
or

- use one program (with its own independent address space) per core and have an inter-process communication method, i.e. we run the same program multiple times and those copies communicate with each other

Methods

We want:

- one execution path per core, to make full use of the computing capabilities
- the executions paths should be able to exchange information, either explicitly or implicitly



We can either:

- use one program which spawns multiple execution threads, i.e. every thread can see the whole memory of the program

or

- use one program (with its own independent address space) per core and have an inter-process communication method, i.e. we run the same program multiple times and those copies communicate with each other

Methods

We want:

- one execution path per core, to make full use of the computing capabilities
- the executions paths should be able to exchange information, either explicitly or implicitly

e.g. OpenMP, POSIX threads
→ shared memory models

We can either:

- use one program which spawns multiple execution threads, i.e. every thread can see the whole memory of the program

or

- use one program (with its own independent address space) per core and have an inter-process communication method, i.e. we run the same program multiple times and those copies communicate with each other

e.g. MPI, Network sockets
→ distributed memory models

Limits

Shared memory:

- generally limited to work within one node
- cache-coherence issues limit scalability

Distributed memory:

- access to whole memory not directly possible and requires sophisticated communication schemes
- every connection requires some memory overhead for book-keeping, this can limit scalability

Limits

Shared memory:

- generally limited to work within one node
- cache-coherence issues limit scalability

Distributed memory:

- access to whole memory not directly possible and requires sophisticated communication schemes
- every connection requires some memory overhead for book-keeping, this can limit scalability

Limits

Shared memory:

- generally limited to work within one node
- cache-coherence issues limit scalability

Distributed memory:

- access to whole memory not directly possible and requires sophisticated communication schemes
- every connection requires some memory overhead for book-keeping, this can limit scalability

OpenMP: What is it?

– API extension to C, C++, and Fortran

– allows for shared memory programming

– based on compiler directives with a small runtime library

– is portable (gcc supports OpenMP)

– retains the sequential version of the code

– supports task and data parallelism

– Current Version: 3.0 (not yet supported in all compilers)

– Main web page: www.openmp.org

OpenMP: What is it?

- API extension to C, C++, and Fortran
- allows for shared memory programming
- based on compiler directives with a small runtime library
- is portable (gcc supports OpenMP)
- retains the sequential version of the code
- supports task and data parallelism
- Current Version: 3.0 (not yet supported in all compilers)
- Main web page: www.openmp.org

OpenMP: What is it?

- API extension to C, C++, and Fortran
- allows for shared memory programming
- based on compiler directives with a small runtime library
- is portable (gcc supports OpenMP)
- retains the sequential version of the code
- supports task and data parallelism
- Current Version: 3.0 (not yet supported in all compilers)
- Main web page: www.openmp.org

OpenMP: What is it?

- API extension to C, C++, and Fortran
- allows for shared memory programming
- based on compiler directives with a small runtime library
- is portable (gcc supports OpenMP)
- retains the sequential version of the code
- supports task and data parallelism
- Current Version: 3.0 (not yet supported in all compilers)
- Main web page: www.openmp.org

OpenMP: What is it?

- API extension to C, C++, and Fortran
- allows for shared memory programming
- based on compiler directives with a small runtime library
- is portable (gcc supports OpenMP)
- retains the sequential version of the code
- supports task and data parallelism
- Current Version: 3.0 (not yet supported in all compilers)
- Main web page: www.openmp.org

OpenMP: What is it?

- API extension to C, C++, and Fortran
- allows for shared memory programming
- based on compiler directives with a small runtime library
- is portable (gcc supports OpenMP)
- retains the sequential version of the code
- supports task and data parallelism
- Current Version: 3.0 (not yet supported in all compilers)
- Main web page: www.openmp.org

OpenMP: What is it?

- API extension to C, C++, and Fortran
- allows for shared memory programming
- based on compiler directives with a small runtime library
- is portable (gcc supports OpenMP)
- retains the sequential version of the code
- supports task and data parallelism
- Current Version: 3.0 (not yet supported in all compilers)
- Main web page: www.openmp.org

OpenMP: What is it?

- API extension to C, C++, and Fortran
- allows for shared memory programming
- based on compiler directives with a small runtime library
- is portable (gcc supports OpenMP)
- retains the sequential version of the code
- supports task and data parallelism
- Current Version: 3.0 (not yet supported in all compilers)
- Main web page: www.openmp.org

OpenMP: What is it?

- API extension to C, C++, and Fortran
- allows for shared memory programming
- based on compiler directives with a small runtime library
- is portable (gcc supports OpenMP)
- retains the sequential version of the code
- supports task and data parallelism
- Current Version: 3.0 (not yet supported in all compilers)
- Main web page: www.openmp.org

OpenMP: What is it?

- API extension to C, C++
- allows for shared memo
- based on compiler direct
- is portable (gcc supports
- retains the sequential ve
- supports task and data p
- Current Version: 3.0 (no
- Main web page: www.openmp.org

OpenMP home page

OpenMP™ The OpenMP API specification for parallel programming

Subscribe to the News Feed

»» OpenMP Specifications
»About OpenMP
»Compilers
»Resources
»Discussion Forum

Events
» SC09 - November 14-20 - Portland, OR

Input Register
Alert the OpenMP.org webmaster about new products, events, or updates and we'll post it here.
»webmaster@openmp.org

Search OpenMP.org
Search

Archives
- October 2009
- June 2009
- April 2009
- March 2009
- February 2009
- January 2009
- November 2008
- October 2008

OpenMP News

»OpenMP Will Be At SC09

SC Portland, OR 2009 Computing For A Changing World. November 14-20, 2009 Oregon Convention Center Portland, Oregon

The OpenMP team will be at **SC09** in Portland OR.

Visit us at **Booth #2985** and get your own C/C++ OpenMP 3.0 syntax cards.

- Sign up for the **"Hands On OpenMP" tutorial with Tim Mattson, Larry Meadows, and Mark Bull, Sunday 8.30-19:00**
- **BoF Session Wednesday 17:30-19:00 OpenMP: Evolving in an Age of Extreme Parallelism**

BoF Agenda:
1. Welcome and summary of OpenMP ARB news (5-10 minutes)
- Larry Meadows (Intel)
2. OpenMP language committee update (20-25 minutes)
- Bronis de Supinski (LLNL)
3. Announcements (5-10 minutes)
- IWOMP 2010
4. Panel (45 minutes): "*OpenMP and heterogeneous platforms (such as GPGPU, CPU/LRB, or CPU/Cell combinations)*"
- Moderator: Bronis de Supinski, LLNL
- Panelists: Tim Mattson, Intel; Barbara Chapman, Univ. of Houston; Michael Wolfe, The Portland Group; Yuan Lin, Nvidia
5. Wrap up (5 minutes)
- Larry Meadows

Come by the booth and meet with some of the designers of the OpenMP 3.0 specifications!

Posted on October 27, 2009

The OpenMP API supports multi-platform shared-memory parallel programming in C/C++ and Fortran. OpenMP is a portable, scalable model with a simple and flexible interface for developing parallel applications on platforms from the desktop to the supercomputer.
»*Read about OpenMP.org*

Get
»OpenMP specs

Use
»OpenMP Compilers

Learn

Using OpenMP

»*Using OpenMP* -- the book

How to use it

For directives:
- figure out the right compiler switch to allow for parsing of OpenMP pragmas (gcc: -fopenmp; icc: -openmp)
- add directives at the parts in your code that you want to parallelize

For library:
- use the same compiler switch (it will deal with linking as well)
- include <omp.h> in the source files in which you want to use OpenMP library functions

Running:
- run the code as you would have done
- you can influence the number of threads the code uses at (basically) two positions:
  - in the code itself (explicitly setting the number of threads)
  - at the command line through environment variables

Good to know:
- If a file is compiled by an OpenMP enabled compiler, the macro _OPENMP is defined, i.e. you can use that to conditionally compile your code

How to use it

For directives:

- figure out the right compiler switch to allow for parsing of OpenMP pragmas (gcc: -fopenmp; icc: -openmp)
- add directives at the parts in your code that you want to parallelize

For library:

- use the same compiler switch (it will deal with linking as well)
- include <omp.h> in the source files in which you want to use OpenMP library functions

Running:

- run the code as you would have done
- you can influence the number of threads the code uses at (basically) two positions:
  - in the code itself (explicitly setting the number of threads)
  - at the command line through environment variables

Good to know:

- If a file is compiled by an OpenMP enabled compiler, the macro _OPENMP is defined, i.e. you can use that to conditionally compile your code

How to use it

For directives:

- figure out the right compiler switch to allow for parsing of OpenMP pragmas (gcc: -fopenmp; icc: -openmp)
- add directives at the parts in your code that you want to parallelize

For library:

- use the same compiler switch (it will deal with linking as well)
- include <omp.h> in the source files in which you want to use OpenMP library functions

Running:

- run the code as you would have done
- you can influence the number of threads the code uses at (basically) two positions:
  - in the code itself (explicitly setting the number of threads)
  - at the command line through environment variables

Good to know:

- If a file is compiled by an OpenMP enabled compiler, the macro _OPENMP is defined, i.e. you can use that to conditionally compile your code

How to use it

For directives:

- figure out the right compiler switch to allow for parsing of OpenMP pragmas (gcc: -fopenmp; icc: -openmp)
- add directives at the parts in your code that you want to parallelize

For library:

- use the same compiler switch (it will deal with linking as well)
- include <omp.h> in the source files in which you want to use OpenMP library functions

Running:

- run the code as you would have done
- you can influence the number of threads the code uses at (basically) two positions:
  - in the code itself (explicitly setting the number of threads)
  - at the command line through environment variables

Good to know:

- If a file is compiled by an OpenMP enabled compiler, the macro _OPENMP is defined, i.e. you can use that to conditionally compile your code

How to use it

For directives:

- figure out the right compiler switch to allow for parsing of OpenMP pragmas (gcc: -fopenmp; icc: -openmp)
- add directives at the parts in your code that you want to parallelize

For library:

- use the same compiler switch (it will deal with linking as well)
- include <omp.h> in the source files in which you want to use OpenMP library functions

Running:

- run the code as you would have done
- you can influence the number of threads the code uses at (basically) two positions:
  - in the code itself (explicitly setting the number of threads)
  - at the command line through environment variables

Good to know:

- If a file is compiled by an OpenMP enabled compiler, the macro _OPENMP is defined, i.e. you can use that to conditionally compile your code

Directives

Standard way:

- start a parallel region
- perform things in parallel

Example:

- Square matrix multiplication

```
#define IDX(n, i, j) ((i) * (n) + (j))

extern void
matrix_mulQuad3(const double *a, const double *b,
                double *c, int n)
{




    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            double tmp = 0.0;
            for (int k = 0; k < n; k++) {
                tmp +=  a[IDX(n, i, k)]
                       *b[IDX(n, k, j)];
            }
            c[IDX(n, i, j)] = tmp;
        }
    }
}
```

Directives

Standard way:
- start a parallel region
- perform things in parallel

Example:
- Square matrix multiplication

```
#define IDX(n, i, j) ((i) * (n) + (j))

extern void
matrix_mulQuad3(const double *a, const double *b,
                double *c, int n)
{




    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            double tmp = 0.0;
            for (int k = 0; k < n; k++) {
                tmp +=  a[IDX(n, i, k)]
                       *b[IDX(n, k, j)];
            }
            c[IDX(n, i, j)] = tmp;
        }
    }
}
```

Directives

Standard way:

- start a parallel region
- perform things in parallel

Example:

- Square matrix multiplication

```
#ifdef _OPENMP
#include <omp.h>
#endif

#define IDX(n, i, j) ((i) * (n) + (j))

extern void
matrix_mulQuad3(const double *a, const double *b,
                double *c, int n)
{
#ifdef _OPENMP
#pragma omp parallel for \
    schedule(static)     \
    shared(a, b, c, n)
#endif
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            double tmp = 0.0;
            for (int k = 0; k < n; k++) {
                tmp +=  a[IDX(n, i, k)]
                       *b[IDX(n, k, j)];
            }
            c[IDX(n, i, j)] = tmp;
        }
    }
}
```

Directives: Constructs

The parallel construct forms a team of threads and starts parallel execution. The parallel execution ends at the end of the structured block.

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

Clauses:

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

Directives: Constructs

The loop constructs will cause the iteration to be split among the encountering team of threads.

```
#pragma omp for [clauses]
   structured block
```

Clauses:

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

Directives: Constructs

The section contains a set of structured blocks that will distributed among the encountering team of threads.

```
#pragma omp sections [clauses]
    {
        #pragma omp section
        structured block
        #pragma omp section
        structured block
    }
```

Clauses:

- `private (list)`
- `firstprivate (list)`
- `lastprivate (list)`
- `reduction (operator: list)`
- `nowait`

Directives: Constructs

Combined directives are a shortcut for defining a single workshare region with no further parallel parts.

```
#pragma omp parallel for [clauses]
   structured block

#pragma omp parallel sections [clauses]
   {
      #pragma omp section
      structured block
      #pragma omp section
      structured block
   }
```

Directives: Constructs

The task construct defines an explicit task.

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

Clauses:

- `if (expression)`
- `untied`
- `default (shared|none)`
- `private (list)`
- `firstprivate (list)`
- `shared (list)`

Directives: Constructs

The critical construct restricts the block to be executed by only one thread at a time.

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

The atomic construct ensures that a specific memory location is updated atomically, i.e. only one write access at a time.

```
#pragma omp atomic
   expression
      x = expr
      x++
      x--
      ++x
      --x
```

Directives: Constructs

The critical construct restricts the block to be executed by only one thread at a time.

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

The atomic construct ensures that a specific memory location is updated atomically, i.e. only one write access at a time.

```
#pragma omp atomic
   expression
      x = expr
      x++
      x--
      ++x
      --x
```

Directives: Clauses

Data sharing clauses

`default(shared|none)`
Controls the default data sharing of variables that are used in the construct.

`shared(list)`
Declares the variables in the list to be shared.

`private(list)`
Declares the variables in the list to be private.

`firstprivate(list)`
Declares the variables in the list to be private and initializes each of them with the value that the corresponding original item has when the construct is encountered.

`lastprivate(list)`
Declares the variables in the list to be private and causes the corresponding original item to be updated at the end of the region.

`reduction(operator:list)`
Declares accumulation into the list items using the indicated associative operator.

Directives: Clauses

Data sharing clauses

`default(shared|none)`
Controls the default data sharing of variables that are used in the construct.

`shared(list)`
Declares the variables in the list to be shared.

`private(list)`
Declares the variables in the list to be private.

`firstprivate(list)`
Declares the variables in the list to be private and initializes each of them with the value that the corresponding original item has when the construct is encountered.

`lastprivate(list)`
Declares the variables in the list to be private and causes the corresponding original item to be updated at the end of the region.

`reduction(operator:list)`
Declares accumulation into the list items using the indicated associative operator.

Directives: Clauses

Data sharing clauses

`default(shared|none)`
Controls the default data sharing of variables that are used in the construct.

`shared(list)`
Declares the variables in the list to be shared.

`private(list)`
Declares the variables in the list to be private.

`firstprivate(list)`
Declares the variables in the list to be private and initializes each of them with the value that the corresponding original item has when the construct is encountered.

`lastprivate(list)`
Declares the variables in the list to be private and causes the corresponding original item to be updated at the end of the region.

`reduction(operator:list)`
Declares accumulation into the list items using the indicated associative operator.

Directives: Clauses

Data sharing clauses

`default(shared|none)`
Controls the default data sharing of variables that are used in the construct.

`shared(list)`
Declares the variables in the list to be shared.

`private(list)`
Declares the variables in the list to be private.

`firstprivate(list)`
Declares the variables in the list to be private and initializes each of them with the value that the corresponding original item has when the construct is encountered.

`lastprivate(list)`
Declares the variables in the list to be private and causes the corresponding original item to be updated at the end of the region.

`reduction(operator:list)`
Declares accumulation into the list items using the indicated associative operator.

Directives: Clauses

Data sharing clauses

`default(shared|none)`
Controls the default data sharing of variables that are used in the construct.

`shared(list)`
Declares the variables in the list to be shared.

`private(list)`
Declares the variables in the list to be private.

`firstprivate(list)`
Declares the variables in the list to be private and initializes each of them with the value that the corresponding original item has when the construct is encountered.

`lastprivate(list)`
Declares the variables in the list to be private and causes the corresponding original item to be updated at the end of the region.

`reduction(operator:list)`
Declares accumulation into the list items using the indicated associative operator.

Directives: Clauses

Data sharing clauses

`default(shared|none)`
: Controls the default data sharing of variables that are used in the construct.

`shared(list)`
: Declares the variables in the list to be shared.

`private(list)`
: Declares the variables in the list to be private.

`firstprivate(list)`
: Declares the variables in the list to be private and initializes each of them with the value that the corresponding original item has when the construct is encountered.

`lastprivate(list)`
: Declares the variables in the list to be private and causes the corresponding original item to be updated at the end of the region.

`reduction(operator:list)`
: Declares accumulation into the list items using the indicated associative operator.

Directives: Clauses

Data copying clauses

`copyin (list)`

Copies the values of the master thread's threadprivate variable to the threadprivate variable of each other member of the team.

`copyprivate(list)`

Broadcasts a value from the data environment of one implicit task to the data environment of the other implicit tasks belonging to the parallel region.

Directives: Clauses

Data copying clauses

`copyin (list)`

Copies the values of the master thread's threadprivate variable to the threadprivate variable of each other member of the team.

`copyprivate(list)`

Broadcasts a value from the data environment of one implicit task to the data environment of the other implicit tasks belonging to the parallel region.

Directives: Clauses

Schedule types for the loop construct

`static`
: Iterations are divided into chunks of chunk_size and the chunks are assigned to the tasks in a round-robin fashion in order of the thread number.

`dynamic`
: Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain.

`guided`
: Each thread executes a chunk of iterations, then requests another chunk until no chunks remain. The chunk size starts large and shrinks to the indicated chunk_size as chunks are scheduled.

`auto`
: The decision regarding the scheduling is up to the compiler/runtime.

`runtime`
: The scheduler is set via environment variables.

Directives: Clauses

Schedule types for the loop construct

`static`
: Iterations are divided into chunks of chunk_size and the chunks are assigned to the tasks in a round-robin fashion in order of the thread number.

`dynamic`
: Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain.

`guided`
: Each thread executes a chunk of iterations, then requests another chunk until no chunks remain. The chunk size starts large and shrinks to the indicated chunk_size as chunks are scheduled.

`auto`
: The decision regarding the scheduling is up to the compiler/runtime.

`runtime`
: The scheduler is set via environment variables.

Directives: Clauses

Schedule types for the loop construct

`static`
: Iterations are divided into chunks of chunk_size and the chunks are assigned to the tasks in a round-robin fashion in order of the thread number.

`dynamic`
: Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain.

`guided`
: Each thread executes a chunk of iterations, then requests another chunk until no chunks remain. The chunk size starts large and shrinks to the indicated chunk_size as chunks are scheduled.

`auto`
: The decision regarding the scheduling is up to the compiler/runtime.

`runtime`
: The scheduler is set via environment variables.

Directives: Clauses

Schedule types for the loop construct

`static`
: Iterations are divided into chunks of chunk_size and the chunks are assigned to the tasks in a round-robin fashion in order of the thread number.

`dynamic`
: Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain.

`guided`
: Each thread executes a chunk of iterations, then requests another chunk until no chunks remain. The chunk size starts large and shrinks to the indicated chunk_size as chunks are scheduled.

`auto`
: The decision regarding the scheduling is up to the compiler/runtime.

`runtime`
: The scheduler is set via environment variables.

Directives: Clauses

Schedule types for the loop construct

`static`
: Iterations are divided into chunks of chunk_size and the chunks are assigned to the tasks in a round-robin fashion in order of the thread number.

`dynamic`
: Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain.

`guided`
: Each thread executes a chunk of iterations, then requests another chunk until no chunks remain. The chunk size starts large and shrinks to the indicated chunk_size as chunks are scheduled.

`auto`
: The decision regarding the scheduling is up to the compiler/runtime.

`runtime`
: The scheduler is set via environment variables.

Library

Useful library functions:

```
int omp_get_num_threads(void);
```
Returns the number of threads in the current team.

```
int omp_get_thread_num(void);
```
Returns the ID of the encountering threads. IDs range from 0..size-1.

```
int omp_get_num_procs(void);
```
Returns the number of processors available to the program.

```
double omp_get_wtime(void);
```
Returns elapsed wall clock time in seconds.

```
double omp_get_wtick(void);
```
Returns the precision of the timer used by omp_get_wtime.

Library

Useful library functions:

```
int omp_get_num_threads(void);
```
Returns the number of threads in the current team.

```
int omp_get_thread_num(void);
```
Returns the ID of the encountering threads. IDs range from 0..size-1.

```
int omp_get_num_procs(void);
```
Returns the number of processors available to the program.

```
double omp_get_wtime(void);
```
Returns elapsed wall clock time in seconds.

```
double omp_get_wtick(void);
```
Returns the precision of the timer used by omp_get_wtime.

Library

Useful library functions:

```
int omp_get_num_threads(void);
```

Returns the number of threads in the current team.

```
int omp_get_thread_num(void);
```

Returns the ID of the encountering threads. IDs range from 0..size-1.

```
int omp_get_num_procs(void);
```

Returns the number of processors available to the program.

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

Returns elapsed wall clock time in seconds.

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

Returns the precision of the timer used by omp_get_wtime.

Library

Useful library functions:

```
int omp_get_num_threads(void);
```
Returns the number of threads in the current team.

```
int omp_get_thread_num(void);
```
Returns the ID of the encountering threads. IDs range from 0..size-1.

```
int omp_get_num_procs(void);
```
Returns the number of processors available to the program.

```
double omp_get_wtime(void);
```
Returns elapsed wall clock time in seconds.

```
double omp_get_wtick(void);
```
Returns the precision of the timer used by omp_get_wtime.

Library

Useful library functions:

```
int omp_get_num_threads(void);
```

Returns the number of threads in the current team.

```
int omp_get_thread_num(void);
```

Returns the ID of the encountering threads. IDs range from 0..size-1.

```
int omp_get_num_procs(void);
```

Returns the number of processors available to the program.

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

Returns elapsed wall clock time in seconds.

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

Returns the precision of the timer used by omp_get_wtime.

MPI: What is it?

– Is a *Message Passing Interface* standard

– Library for C (C++) and Fortran

– allows for *message passing* between programs connected in a communication group

– provides means for
  – point to point communication
  – collective communication
  – one sided communication
  – data abstraction

– is portable and available on all relevant clusters

– can run on shared memory machines as well (i.e. on your laptop)

– Different implementations, try www.openmpi.org

MPI: What is it?

- Is a *Message Passing Interface* standard
- Library for C (C++) and Fortran
- allows for *message passing* between programs connected in a communication group
- provides means for
  - point to point communication
  - collective communication
  - one sided communication
  - data abstraction
- is portable and available on all relevant clusters
- can run on shared memory machines as well (i.e. on your laptop)
- Different implementations, try www.openmpi.org

MPI: What is it?

– Is a *Message Passing Interface* standard

– Library for C (C++) and Fortran

– allows for *message passing* between programs connected in a communication group

– provides means for
  – point to point communication
  – collective communication
  – one sided communication
  – data abstraction

– is portable and available on all relevant clusters

– can run on shared memory machines as well (i.e. on your laptop)

– Different implementations, try www.openmpi.org

MPI: What is it?

– Is a *Message Passing Interface* standard

– Library for C (C++) and Fortran

– allows for *message passing* between programs connected in a communication group

– provides means for
  – point to point communication
  – collective communication
  – one sided communication
  – data abstraction

– is portable and available on all relevant clusters

– can run on shared memory machines as well (i.e. on your laptop)

– Different implementations, try www.openmpi.org

MPI: What is it?

- Is a *Message Passing Interface* standard
- Library for C (C++) and Fortran
- allows for *message passing* between programs connected in a communication group
- provides means for
  - point to point communication
  - collective communication
  - one sided communication
  - data abstraction
- is portable and available on all relevant clusters
- can run on shared memory machines as well (i.e. on your laptop)
- Different implementations, try www.openmpi.org

MPI: What is it?

- Is a *Message Passing Interface* standard
- Library for C (C++) and Fortran
- allows for *message passing* between programs connected in a communication group
- provides means for
  - point to point communication
  - collective communication
  - one sided communication
  - data abstraction
- is portable and available on all relevant clusters
- can run on shared memory machines as well (i.e. on your laptop)
- Different implementations, try www.openmpi.org

MPI: What is it?

- Is a *Message Passing Interface* standard
- Library for C (C++) and Fortran
- allows for *message passing* between programs connected in a communication group
- provides means for
  - point to point communication
  - collective communication
  - one sided communication
  - data abstraction
- is portable and available on all relevant clusters
- can run on shared memory machines as well (i.e. on your laptop)
- Different implementations, try www.openmpi.org

MPI: What is it?

- Is a *Message Passing Interface* standard
- Library for C (C++) and Fortran
- allows for *message passing* between programs connected in a communication group
- provides means for
  - point to point communication
  - collective communication
  - one sided communication
  - data abstraction
- is portable and available on all relevant clusters
- can run on shared memory machines as well (i.e. on your laptop)
- Different implementations, try www.openmpi.org

How to use it

- Include <mpi.h> to use the library functions
- Use mpicc (as your CC) as the compiler/linker, it will deal with the right flags to find the headers and libraries
- You need to initialize the library in your code, before you can use it
- Run your code with mpiexec (it will start multiple copies of your executable and set up the communication layer between them)

How to use it

- Include <mpi.h> to use the library functions
- Use mpicc (as your CC) as the compiler/linker, it will deal with the right flags to find the headers and libraries
- You need to initialize the library in your code, before you can use it
- Run your code with mpiexec (it will start multiple copies of your executable and set up the communication layer between them)

How to use it

- Include <mpi.h> to use the library functions
- Use mpicc (as your CC) as the compiler/linker, it will deal with the right flags to find the headers and libraries
- You need to initialize the library in your code, before you can use it

```
#include <mpi.h>

int main(int argc, char **argv)
{
    MPI_Init(argc, argv);
    // Do things
    MPI_Finalize();
    return EXIT_SUCCESS;
}
```

- Run your code with mpiexec (it will start multiple copies of your executable and set up the communication layer between them)

How to use it

- Include <mpi.h> to use the library functions
- Use mpicc (as your CC) as the compiler/linker, it will deal with the right flags to find the headers and libraries
- You need to initialize the library in your code, before you can use it

```
#include <mpi.h>

int main(int argc, char **argv)
{
    MPI_Init(argc, argv);
    // Do things
    MPI_Finalize();
    return EXIT_SUCCESS;
}
```

- Run your code with mpiexec (it will start multiple copies of your executable and set up the communication layer between them)

```
mpiexec -n 4 ./myCode myParamater1 myParamter2
```

How to use it: Blocking vs Non-blocking communication

- Communication generally come in two forms: blocking and non-blocking

- blocking communications
  - start a data transfer and do not return until the transfer has been completed
- non-blocking communications
  - initiate a communication but return immediately and you need to check yourself when the data has been transferred

- This mechanism is meant to
  - allow overlapping of communication and computation
  - prevent dead-locks

How to use it: Blocking vs Non-blocking communication

- Communication generally come in two forms: blocking and non-blocking

- blocking communications
  - start a data transfer and do not return until the transfer has been completed
- non-blocking communications
  - initiate a communication but return immediately and you need to check yourself when the data has been transferred

- This mechanism is meant to
  - allow overlapping of communication and computation
  - prevent dead-locks

How to use it: Blocking vs Non-blocking communication

- Communication generally come in two forms: blocking and non-blocking

- blocking communications
  - start a data transfer and do not return until the transfer has been completed
- non-blocking communications
  - initiate a communication but return immediately and you need to check yourself when the data has been transferred

- This mechanism is meant to
  - allow overlapping of communication and computation
  - prevent dead-locks

How to use it: Blocking vs Non-blocking communication

- Communication generally come in two forms: blocking and non-blocking

- blocking communications
  - start a data transfer and do not return until the transfer has been completed
- non-blocking communications
  - initiate a communication but return immediately and you need to check yourself when the data has been transferred

- This mechanism is meant to
  - allow overlapping of communication and computation
  - prevent dead-locks

How to use it: Blocking vs Non-blocking communication

- Communication generally come in two forms: blocking and non-blocking

- blocking communications
  - start a data transfer and do not return until the transfer has been completed
- non-blocking communications
  - initiate a communication but return immediately and you need to check yourself when the data has been transferred

- This mechanism is meant to
  - allow overlapping of communication and computation
  - prevent dead-locks

```
#include <mpi.h>

#define TAG 0

int main(int argc, char **argv)
{
    MPI_Status  status;
    int         rank, sendTo, recvFrom, recvData;

    MPI_Init(argc, argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    recvFrom = sendTo = (rank + 1) % 2;

    // This will produce a dead-lock
    MPI_Send(&rank, 1, MPI_INT, sendTo, TAG,
             MPI_COMM_WORLD);
    MPI_Recv(&recvData, 1, MPI_INT, recvFrom, TAG,
             MPI_COMM_WORLD, &status);

    MPI_Finalize();
    return EXIT_SUCCESS;
}
```

How to use it: Blocking vs Non-blocking communication

- Communication generally come in two forms: blocking and non-blocking

- blocking communications
  - start a data transfer and do not return until the transfer has been completed
- non-blocking communications
  - initiate a communication but return immediately and you need to check yourself when the data has been transferred

- This mechanism is meant to
  - allow overlapping of communication and computation
  - prevent dead-locks

```
#include <mpi.h>

#define TAG 0

int main(int argc, char **argv)
{
    MPI_Request request;
    MPI_Status  status;
    int         rank, sendTo, recvFrom, recvData;

    MPI_Init(argc, argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    recvFrom = sendTo = (rank + 1) % 2;

    MPI_IRecv(&recvData, 1, MPI_INT, recvFrom, TAG,
              MPI_COMM_WORLD, &request);
    MPI_Send(&rank, 1, MPI_INT, sendTo, TAG,
              MPI_COMM_WORLD);
    MPI_Wait(&request, &status);

    MPI_Finalize();
    return EXIT_SUCCESS;
}
```

Collective communications

- Collective communications involve all processes in a communicator (hence all have to call the collective routine)
- Can be used to
  - broadcast values
  - reduce values
  - synchronize the execution
- Depending on the runtime, collective calls may be optimised (e.g. tree pattern in communication) and thus faster than their point-to-point equivalents.

Collective communications

- Collective communications involve all processes in a communicator (hence all have to call the collective routine)

- Can be used to
  - broadcast values
  - reduce values
  - synchronize the execution

- Depending on the runtime, collective calls may be optimised (e.g. tree pattern in communication) and thus faster than their point-to-point equivalents.

Collective communications

- Collective communications involve all processes in a communicator (hence all have to call the collective routine)
- Can be used to
  - broadcast values
  - reduce values
  - synchronize the execution
- Depending on the runtime, collective calls may be optimised (e.g. tree pattern in communication) and thus faster than their point-to-point equivalents.

Collective communications: Broadcasts

Prototype:

```
int
MPI_BCast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);
```

Example:

Collective communications: Broadcasts

Prototype:

```
int
MPI_BCast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);
```

Example:

```
{
    int n;

    // Code

    if (rank == 0)
        n = getDimension();

    MPI_BCast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // More Code
}
```

Collective communications: Reductions

Prototype:

```
int
MPI_Reduce(void *sendBuf, void *recvBuf, int count, MPI_Datatype datatype,
           MPI_Op op, int root, MPI_Comm comm);
```

Example:

Collective communications: Reductions

Prototype:

```
int
MPI_Reduce(void *sendBuf, void *recvBuf, int count, MPI_Datatype datatype,
           MPI_Op op, int root, MPI_Comm comm);
```

Example:

```
{
    double myRand, sum;

    // Code

    myRand = getRandomNumber();
    sum    = 0.0;

    MPI_Reduce(&myRand, &sum, 1, MPI_DOUBLE,
               MPI_SUM, 0, MPI_COMM_WORLD);

    // More Code
}
```

Collective communications: Synchronizing

Prototype:

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

Example:

Collective communications: Synchronizing

Prototype:

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

Example:

```
{
    // Complicated computation

    // Wait here until all tasks arrive at that point
    MPI_Barrier(MPI_COMM_WORLD);

    // Continue doing things
}
```

Point-to-point communications

- Point-to-point communications involve pairs of processes
  - one sends
  - one receives
  - every send needs a receive and every receive needs a send
  - beware of dead-locks

- Can be used to
  - exchange boundary values
  - pass notes
  - realize complicated communication patterns

Point-to-point communications

- Point-to-point communications involve pairs of processes
  - one sends
  - one receives
  - every send needs a receive and every receive needs a send
  - beware of dead-locks
- Can be used to
  - exchange boundary values
  - pass notes
  - realize complicated communication patterns

Point-to-point communications

- Point-to-point communications involve pairs of processes
  - one sends
  - one receives
  - every send needs a receive and every receive needs a send
  - beware of dead-locks
- Can be used to
  - exchange boundary values
  - pass notes
  - realize complicated communication patterns

Point-to-point communications

- Point-to-point communications involve pairs of processes
  - one sends
  - one receives
  - every send needs a receive and every receive needs a send
  - beware of dead-locks

- Can be used to
  - exchange boundary values
  - pass notes
  - realize complicated communication patterns

Point-to-point communications

- Point-to-point communications involve pairs of processes
  - one sends
  - one receives
  - every send needs a receive and every receive needs a send
  - beware of dead-locks

- Can be used to
  - exchange boundary values
  - pass notes
  - realize complicated communication patterns

Point-to-point communications

- Point-to-point communications involve pairs of processes
  - one sends
  - one receives
  - every send needs a receive and every receive needs a send
  - beware of dead-locks

- Can be used to
  - exchange boundary values
  - pass notes
  - realize complicated communication patterns

Point-to-point communications

- Point-to-point communications involve pairs of processes
  - one sends
  - one receives
  - every send needs a receive and every receive needs a send
  - beware of dead-locks

- Can be used to
  - exchange boundary values
  - pass notes
  - realize complicated communication patterns

Point-to-point communications: Sending

Prototype:

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

int
MPI_ISend(void *sendBuf, int count, MPI_Datatype datatype, int destination,
          int tag, MPI_Comm comm, MPI_Request *request);
```

Point-to-point communications: Receiving

Prototype:

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

int
MPI_IRecv(void *recvBuf, int count, MPI_Datatype datatype, int source,
          int tag, MPI_Comm comm, MPI_Request *request);
```

Point-to-point communications: Example

```
inline static void
local_startReceiving(commScheme_t scheme)
{
    int numBuffersRecv;

    numBuffersRecv       = varArr_getLength(scheme->buffersRecv);
    scheme->requestsRecv = xmalloc(sizeof(MPI_Request) * numBuffersRecv);

    for (int i = 0; i < numBuffersRecv; i++) {
        commSchemeBuffer_t buf;
        buf = varArr_getElementHandle(scheme->buffersRecv, i);
        MPI_Irecv(buf->buf, buf->count, buf->datatype, buf->rank,
                  scheme->tag, scheme->comm, scheme->requestsRecv + i);
    }
}
```

Point-to-point communications: Example

```
inline static void
local_startSending(commScheme_t scheme)
{
    int                firstSendBuf = 0;
    int                numBuffersSend;
    commSchemeBuffer_t buf;

    numBuffersSend       = varArr_getLength(scheme->buffersSend);
    scheme->requestsSend = xmalloc(sizeof(MPI_Request) * numBuffersSend);

    while (firstSendBuf < numBuffersSend) {
        buf = varArr_getElementHandle(scheme->buffersSend, firstSendBuf);
        if (buf->rank > scheme->rank)
            break;
        firstSendBuf++;
    }
    firstSendBuf %= numBuffersSend;

    for (int i = firstSendBuf; i < numBuffersSend; i++) {
        buf = varArr_getElementHandle(scheme->buffersSend, i);
        MPI_Isend(buf->buf, buf->count, buf->datatype, buf->rank,
                  scheme->tag, scheme->comm, scheme->requestsSend + i);
    }

    for (int i = 0; i < firstSendBuf; i++) {
        buf = varArr_getElementHandle(scheme->buffersSend, i);
        MPI_Isend(buf->buf, buf->count, buf->datatype, buf->rank,
                  scheme->tag, scheme->comm, scheme->requestsSend + i);
    }
}
```

Was that all?

...not by far!

- This was only a rough and quick introduction to what OpenMP and MPI can do
- For more details and tutorials, check the Internet
- You can combine MPI and OpenMP in one code
- Instead of using OpenMP you could use POSIX threads (or whatever you local machine provides) to do it 'by hand'.
- Instead of MPI you could manually program a network library that is tailored for your code.
- ...and there are things beyond MPI and OpenMP.

Was that all?

...not by far!

- This was only a rough and quick introduction to what OpenMP and MPI can do
- For more details and tutorials, check the Internet
- You can combine MPI and OpenMP in one code
- Instead of using OpenMP you could use POSIX threads (or whatever you local machine provides) to do it 'by hand'.
- Instead of MPI you could manually program a network library that is tailored for your code.
- ...and there are things beyond MPI and OpenMP.

Was that all?

...not by far!

- This was only a rough and quick introduction to what OpenMP and MPI can do
- For more details and tutorials, check the Internet
- You can combine MPI and OpenMP in one code
- Instead of using OpenMP you could use POSIX threads (or whatever you local machine provides) to do it 'by hand'.
- Instead of MPI you could manually program a network library that is tailored for your code.
- ...and there are things beyond MPI and OpenMP.

Was that all?

...not by far!

- This was only a rough and quick introduction to what OpenMP and MPI can do
- For more details and tutorials, check the Internet
- You can combine MPI and OpenMP in one code
- Instead of using OpenMP you could use POSIX threads (or whatever you local machine provides) to do it 'by hand'.
- Instead of MPI you could manually program a network library that is tailored for your code.
- ...and there are things beyond MPI and OpenMP.

Was that all?

...not by far!

- This was only a rough and quick introduction to what OpenMP and MPI can do
- For more details and tutorials, check the Internet
- You can combine MPI and OpenMP in one code
- Instead of using OpenMP you could use POSIX threads (or whatever you local machine provides) to do it 'by hand'.
- Instead of MPI you could manually program a network library that is tailored for your code.
- ...and there are things beyond MPI and OpenMP.

Was that all?

...not by far!

- This was only a rough and quick introduction to what OpenMP and MPI can do
- For more details and tutorials, check the Internet
- You can combine MPI and OpenMP in one code
- Instead of using OpenMP you could use POSIX threads (or whatever you local machine provides) to do it 'by hand'.
- Instead of MPI you could manually program a network library that is tailored for your code.
- ...and there are things beyond MPI and OpenMP.

Was that all?

...not by far!

- This was only a rough and quick introduction to what OpenMP and MPI can do
- For more details and tutorials, check the Internet
- You can combine MPI and OpenMP in one code
- Instead of using OpenMP you could use POSIX threads (or whatever you local machine provides) to do it 'by hand'.
- Instead of MPI you could manually program a network library that is tailored for your code.
- ...and there are things beyond MPI and OpenMP.

Was that all?

...not by far!

- This was only a rough and quick introduction to what OpenMP and MPI can do
- For more details and tutorials, check the Internet
- You can combine MPI and OpenMP in one code
- Instead of using OpenMP you could use POSIX threads (or whatever you local machine provides) to do it 'by hand'.
- Instead of MPI you could manually program a network library that is tailored for your code.
- ...and there are things beyond MPI and OpenMP.

PGAS – Partitioned Global Address Space

- PGAS languages provide a way to denote that a memory location is remote in the language.
- As such, PGAS languages are generally extensions to existing languages
  - UPC (Unified Parallel C)
  - CAF (CoArray Fortran)
- Can simplify the code significantly for complex communication patterns.
  Note that the communication still has to take place, you just don't have to write it explicitly anymore
- Very promising concept, but not yet mature.

PGAS – Partitioned Global Address Space

- PGAS languages provide a way to denote that a memory location is remote in the language.
- As such, PGAS languages are generally extensions to existing languages
  - UPC (Unified Parallel C)
  - CAF (CoArray Fortran)
- Can simplify the code significantly for complex communication patterns.
  Note that the communication still has to take place, you just don't have to write it explicitly anymore
- Very promising concept, but not yet mature.

PGAS – Partitioned Global Address Space

- PGAS languages provide a way to denote that a memory location is remote in the language.
- As such, PGAS languages are generally extensions to existing languages
  - UPC (Unified Parallel C)
  - CAF (CoArray Fortran)
- Can simplify the code significantly for complex communication patterns. Note that the communication still has to take place, you just don't have to write it explicitly anymore
- Very promising concept, but not yet mature.

PGAS – Partitioned Global Address Space

- PGAS languages provide a way to denote that a memory location is remote in the language.
- As such, PGAS languages are generally extensions to existing languages
  - UPC (Unified Parallel C)
  - CAF (CoArray Fortran)
- Can simplify the code significantly for complex communication patterns.
  Note that the communication still has to take place, you just don't have to write it explicitly anymore
- Very promising concept, but not yet mature.

PGAS – Partitioned Global Address Space

- PGAS languages provide a way to denote that a memory location is remote in the language.
- As such, PGAS languages are generally extensions to existing languages
  - UPC (Unified Parallel C)
  - CAF (CoArray Fortran)
- Can simplify the code significantly for complex communication patterns.
  Note that the communication still has to take place, you just don't have to write it explicitly anymore
- Very promising concept, but not yet mature.

Accelerators

- Accelerators are specialized hardware to perform certain task at raw compute speed way beyond what a single (general purpose) CPU can provide
- Most common accelerators are graphic cards, other options are GRAPE, Cell, ClearSpeed, FPGAs...
- The top notch supercomputers rely on accelerators to achieved their speed (at manageable energy costs)
- Not all problems can benefit from accelerators
- Currently the most interesting ones are indeed graphic cards, they are
  - massively parallel (hence, fast)
  - cheap
  - relatively easy (in comparison) to program (CUDA, better OpenCL)

Accelerators

- Accelerators are specialized hardware to perform certain task at raw compute speed way beyond what a single (general purpose) CPU can provide
- Most common accelerators are graphic cards, other options are GRAPE, Cell, ClearSpeed, FPGAs...
- The top notch supercomputers rely on accelerators to achieved their speed (at manageable energy costs)
- Not all problems can benefit from accelerators
- Currently the most interesting ones are indeed graphic cards, they are
  - massively parallel (hence, fast)
  - cheap
  - relatively easy (in comparison) to program (CUDA, better OpenCL)

Accelerators

- Accelerators are specialized hardware to perform certain task at raw compute speed way beyond what a single (general purpose) CPU can provide
- Most common accelerators are graphic cards, other options are GRAPE, Cell, ClearSpeed, FPGAs...
- The top notch supercomputers rely on accelerators to achieved their speed (at manageable energy costs)
- Not all problems can benefit from accelerators
- Currently the most interesting ones are indeed graphic cards, they are
  - massively parallel (hence, fast)
  - cheap
  - relatively easy (in comparison) to program (CUDA, better OpenCL)

Accelerators

- Accelerators are specialized hardware to perform certain task at raw compute speed way beyond what a single (general purpose) CPU can provide
- Most common accelerators are graphic cards, other options are GRAPE, Cell, ClearSpeed, FPGAs...
- The top notch supercomputers rely on accelerators to achieved their speed (at manageable energy costs)
- Not all problems can benefit from accelerators
- Currently the most interesting ones are indeed graphic cards, they are
  - massively parallel (hence, fast)
  - cheap
  - relatively easy (in comparison) to program (CUDA, better OpenCL)

Accelerators

- Accelerators are specialized hardware to perform certain task at raw compute speed way beyond what a single (general purpose) CPU can provide
- Most common accelerators are graphic cards, other options are GRAPE, Cell, ClearSpeed, FPGAs...
- The top notch supercomputers rely on accelerators to achieved their speed (at manageable energy costs)
- Not all problems can benefit from accelerators
- Currently the most interesting ones are indeed graphic cards, they are
  - massively parallel (hence, fast)
  - cheap
  - relatively easy (in comparison) to program (CUDA, better OpenCL)

Accelerators

- Accelerators are specialized hardware to perform certain task at raw compute speed way beyond what a single (general purpose) CPU can provide
- Most common accelerators are graphic cards, other options are GRAPE, Cell, ClearSpeed, FPGAs...
- The top notch supercomputers rely on accelerators to achieved their speed (at manageable energy costs)
- Not all problems can benefit from accelerators
- Currently the most interesting ones are indeed graphic cards, they are
  - massively parallel (hence, fast)
  - cheap
  - relatively easy (in comparison) to program (CUDA, better OpenCL)