Introduction to Parallel Programming

Stéphane Zuckerman

Laboratoire ETIS
Université Paris-Seine, Université de Cergy-Pontoise, ENSEA, CNRS
F95000, Cergy, France

January 11, 2022

Outline

1. **Resources**
2. **Parallel Architectures Today**
3. **A short history of parallel programming and why we need it**
4. **Message-passing execution models**
   - Standard Features & Execution Model
   - MPI Basics: Features, Compilation, Execution
   - Non-Blocking Communications
   - Collective operations
5. **Shared-memory execution models**
   - Introduction to OpenMP
   - OpenMP Basics
   - Learning More About Shared-Memory Models
6. **References**

Resources

Resources I

Standards and specifications

- OpenMP: [Dagum and Menon 1998; Duran et al. 2011; Ayguade et al. 2009]
  - Useful books: *Using OpenMP* [Chapman, Jost, and Van Der Pas 2008]
  - `http://www.openmp.org`
- MPI: [Graham 2009; Dongarra et al. 1995; Forum 1994]
  - Useful books:
    - *Using MPI* [Gropp, Lusk, and Skjellum 1999; Gropp, Lusk, and Skjellum 2014] (latest: 3^{rd} edition),
    - *Using MPI-2* [Gropp, Lusk, and Thakur 1999]
    - *Using Advanced MPI* [Gropp, Hoefler, et al. 2014]
  - `http://mpi-forum.org`
  - `http://www.mcs.anl.gov/mpi`
- PGAS: `http://www.pgas.org`
- Accelerator programming:
  - Cuda: `https://developer.nvidia.com/cuda-zone`
  - OpenCL: `https://www.khronos.org`, in particular `https://www.khronos.org/opencl/`
  - OpenACC: `https://www.openacc.org`

Resources II

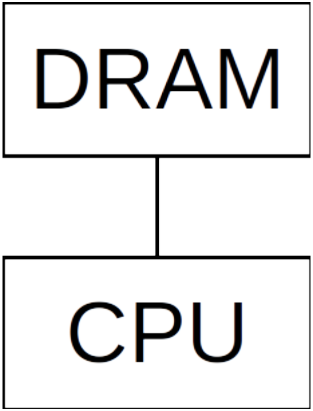
Available implementations

- OpenMP: Clang, GCC since v4.2 (proprietary implementations include Intel's ICC, IBM XL C; *etc.*)
  - Note: GCC's OpenMP runtime is more of a reference implementation than anything.
  - Intel's runtime implementation of OpenMP is free software, and used by Clang. You can also download it and link it to GCC.
- MPI: Mpich-2, Open-MPI
  - Most supercomputer vendors provide their own implementation, often derived from either Mpich or Open-MPI
  - Personal preference: Open-MPI (more modular; more recent–benefits from design issues found in Mpich)
- OpenACC: GCC since v5 (the proprietary PGI compiler also implements it)
- OpenCL: `libclc` on LLVM (Clang/LLVM)

Parallel Architectures Today

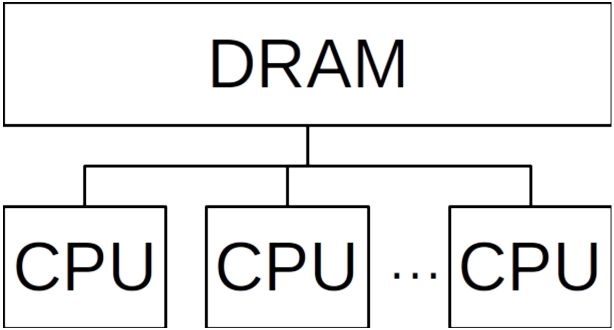
General Purpose Architectures I
An Overview

Figure: Single CPU and a single DRAM bank.



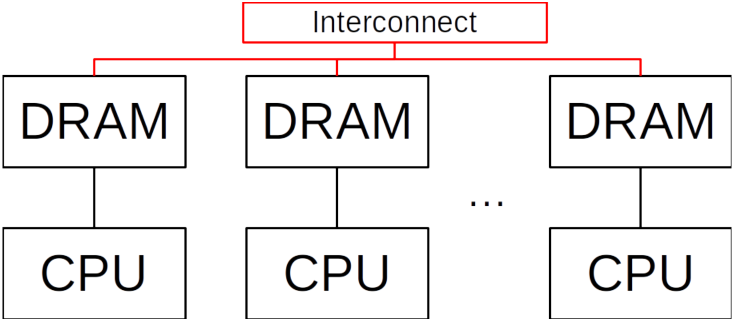
General Purpose Architectures II
An Overview

Figure: Symmetric Multi-Processor (SMP) system; single DRAM bank.



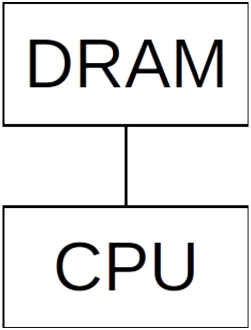
General Purpose Architectures III
An Overview

Figure: Symmetric Multi-Processor (SMP) system with Non-Uniform Memory Access (NUMA); multiple DRAM banks.



General Purpose Architectures IV
An Overview

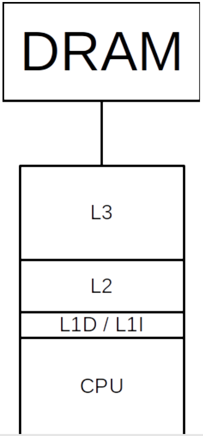
Figure: Single CPU and a single DRAM bank.



General Purpose Architectures V

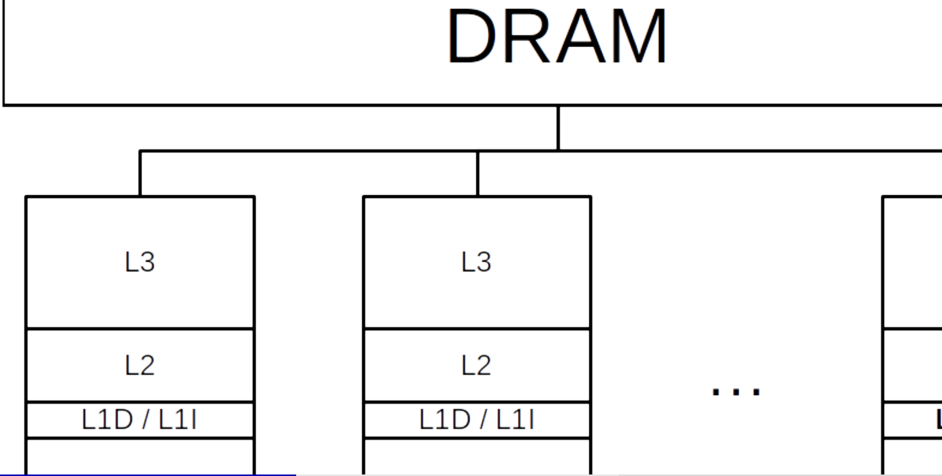
An Overview

Figure: Single CPU: CPU, L1 data cache (L1D), L1 Instruction cache (L1I), L2 unified cache (L2), L3 Unified cache (L3); single DRAM bank.



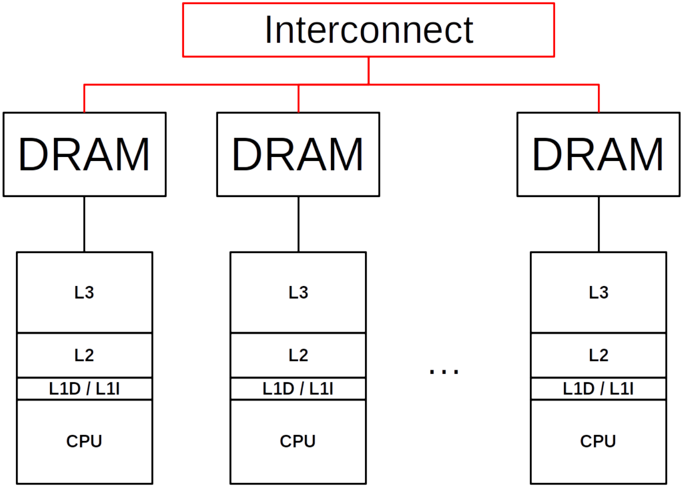
General Purpose Architectures VI
An Overview

Figure: Single CPU+cache hierarchy; single DRAM bank.

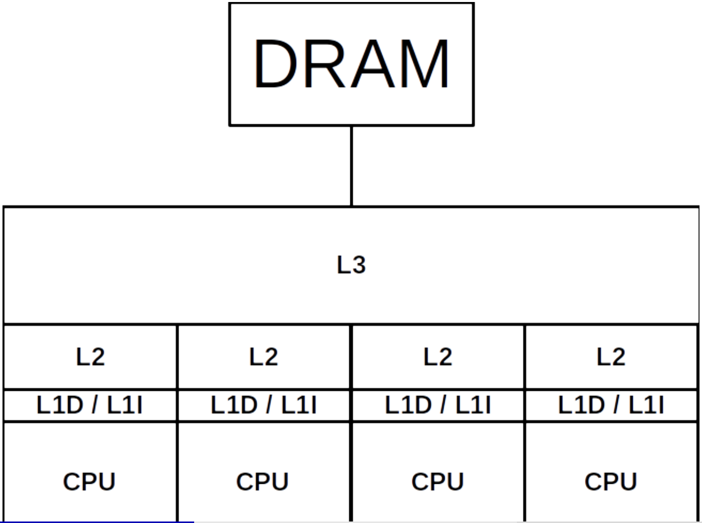


General Purpose Architectures VII

An Overview

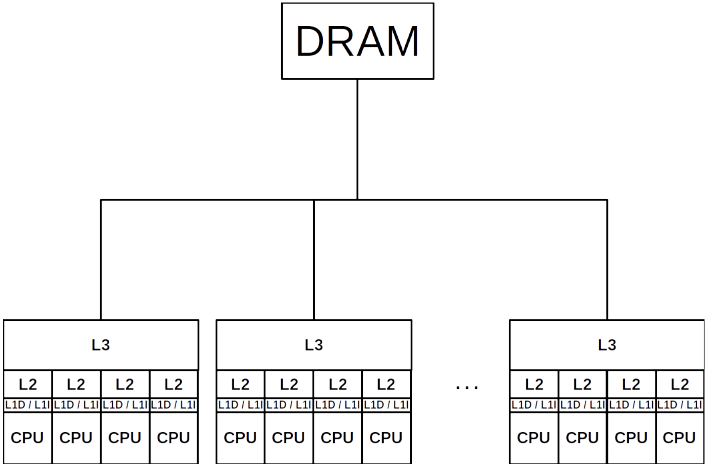


General Purpose Architectures VIII
An Overview

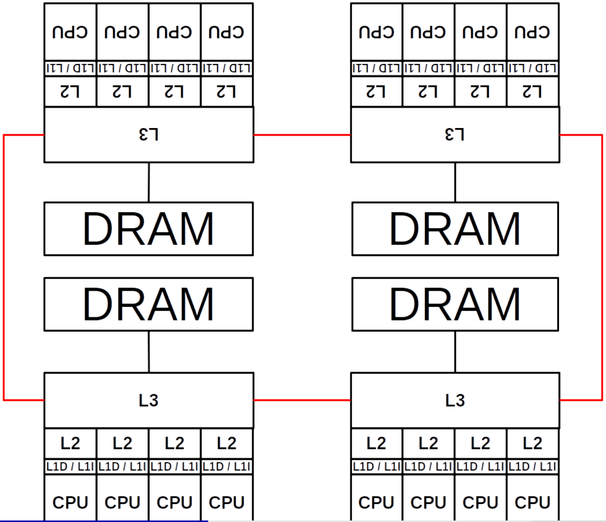


General Purpose Architectures IX

An Overview



General Purpose Architectures X
An Overview



A short history of parallel programming and why we need it

In the beginning... I

Why parallel computing?

- Parallel programming appears very early in the life of computing.
- For every generation of high-end processor, some computations hog all of the available resources.
- Solution: duplicate computing resources.

In the beginning. . . II

Two (non-exclusive) approaches

- Design a parallel processor/computer architecture, *i.e.*, duplicate functional units, provide vector units, . . . :
  - Cray I vector supercomputer
  - Connection Machine
- Design a computer made of multiple computing nodes (also called compute nodes):
  - Custom clusters: SGI Altix, IBM BlueGene, . . .
  - Commodity supercomputers: Beowulf clusters

In the beginning... III

Toward standardization of parallel computing

Until the early/mid 1990s, each supercomputer vendor provides their own parallel computing tools.

- Each new parallel computer requires to learn a new or updated environment to get good performance
- Terrible for portability and productivity

The time was ripe for a standardization effort for all types of parallelism.

In the beginning... IV

Toward standardization of parallel models of computation

- Distributed memory models:
  - PVM (1991)
  - MPI standard (1992)
- Shared memory models:
  - POSIX.1c / IEEE Std 1003.1c-1995 — *a.k.a.* PTHREAD library
  - OpenMP standard (1997)

The obvious benefits are portability and productivity, although *performance portability* is not guaranteed (only correctness).

Why should we care about these parallel models? I

Hardware context

- End of Dennard's scaling
- Moore's law now used to add more computing units on a single chip (instead of going to higher frequencies)
- Programming chip multiprocessors (CMP) is not just for scientific/high-performance computing anymore
  - Embedded chips require programming models and execution models to efficiently exploit all of the hardware

Why should we care about these parallel models? II

Software context – embedded systems

A lot of embedded systems are mainstream and *general purpose* nowadays

- *e.g.*, Raspberry PI, Beagle, *etc.*
- They feature CMPs such as ARM multicore chips
- Even on more specialized platforms, MPI, OpenMP, or OpenCL implementations exist:
  - Xilinx proposes a way to synthesize circuits in FPGAs with OpenCL
  - Adapteva's Parallella board: Zynq-7000 = dual core Cortex A9+FPGA SoC + Epiphany co-processor (16 cores with scratchpads).
  - There are OpenMP and MPI implementations for both ARM and Epiphany boards.
  - Bottom line: "general purpose" parallelism is made available to all parallel platforms nowadays

Why should we care about these parallel models? III

Advantages of traditional programming and execution models

- Because these models are standardized, they are made available in mainstream programming tools
  - OpenMP and MPI both have free/open source implementations available to all
  - Same with PGAS languages
  - Same with GPU-oriented languages
  - Performance goes from "acceptable" to "pretty good"
    - . . . but proprietary implementations tend to be faster because they have better/exclusive knowledge of underlying system software and/or hardware
    - . . . but not always!

Message-passing execution models

An introduction to MPI
Execution Model

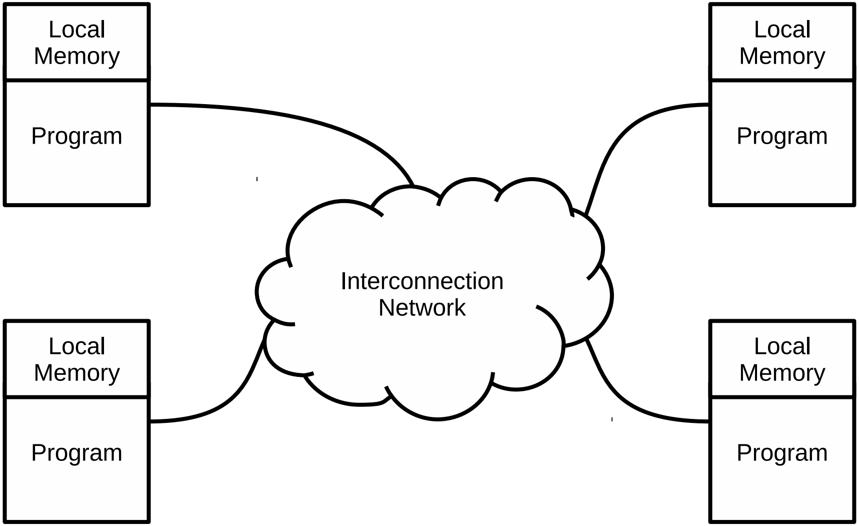
Execution Model

- Relies on the notion of distributed memory
- All data transfers between MPI *processes* are explicit
- Processes can also be synchronized with each other
- Achieved using a library API

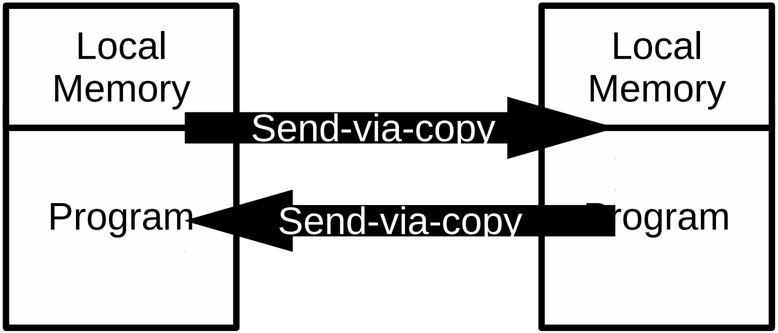
MPI process $\neq$ UNIX or Windows process.

- A process $=$ a program counter $+$ a (separate) address space
- An MPI process could be implemented as a thread [Huang, Lawlor, and Kale 2003]

MPI execution model in a nutshell I



MPI execution model in a nutshell II



MPI-1

- Basic I/O communication functions (100+)
- Blocking send and receive operations
- Nonblocking send and receive operations
- Collective communications
    - Broadcast, scatter, gather, *etc.*
    - Important for performance
- Datatypes to describe data layouet
- Process topologies (use of communicators, tags)
- C, C++, Fortran bindings
- Error codes and classes

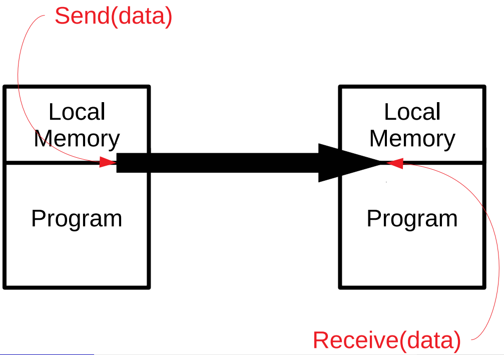
MPI-2 and beyond

- MPI-2 (2000):
  - Thread support
  - MPI-I/O, R-DMA
- MPI-2.1 (2008) and MPI-2.2 (2009):
  - Corrections to standard, small additional features
- MPI-3 (2012):
  - Lots of new features to standard (briefly discussed at the end)

MPI Basics

Stuff needed by the MPI implementation from application

- How to compile and run MPI programs
- How to identify processes
- How to describe the data



Compiling and running MPI programs

MPI is a library

Need to use function calls, to leverage MPI features.

Compilation

- Regular compilation: use of `cc`, *e.g.*, `gcc -o test test.c`
- MPI compilation: `mpicc -o test test.c`

Execution

- Regular execution: `./text`
- MPI execution: `mpiexec -np 16 ./test`

MPI process identification

MPI groups

- Each MPI process belongs to one or more groups
- Each MPI process is given one or more colors
- Group+color = *communicator*
- All MPI processes belong to `MPI_COMM_WORLD` when the program starts

Identifying individual processes: ranks

- If a process belongs to two different communicators, its rank may be different from the point of view of each communicator.

Most basic MPI program

Hello World

```
#include <mpi.h> // required to use MPI functions
#include <stdio.h>

int main(int argc, char* argv[]) {
    int rank, size;

//  must ALWAYS be called to run an MPI program
    MPI_Init(&argc, &argv);
//  get process rank/id
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
//  get total number of processes in communicator
    MPI_Comm_size(MPI_COMM_WOLRD, &size);

    printf("I␣am␣process␣#%d/%d\n", rank, size);

//  must ALWAYS be called to run an MPI program
    MPI_Finalize();
    return 0;
}
```

Basic data transfers

MPI_Send

Syntax: `int MPI_Send (const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`

- `buf`: the buffer from where to read the data to send
- `count`: the number of elements to send over the network link
- `datatype`: the type of data that is being sent, *e.g.*, `MPI_CHAR`, `MPI_INT`, `MPI_DOUBLE`, *etc.*
- `dest`: which process is meant to receive the data (identified by its rank)
- `tag`: a way to discriminate between various messages sent to the same process rank
- `comm`: the communicator (or "group of tasks") to target

Basic data transfers

MPI_Recv

Syntax: `int MPI_Recv (const void *buf, int count, MPI_Datatype datatype, int src, int tag, MPI_Comm comm, MPI_Status *status)`

- `buf`: the buffer from where to write the data to be read
- `count`: the number of elements to receive over the network link
  - count can be bigger than what was received in practice (the real count can be obtained using `MPI_Get_count`)
  - If count is smaller than what is being sent, an error occurs
- `datatype`: the type of data that is being received, *e.g.*, `MPI_CHAR`, `MPI_INT`, `MPI_DOUBLE`, *etc.*
- `dest`: from which process the data originates (identified by its rank)
- `tag`: a way to discriminate between various messages sent to the same receiving process rank
- `comm`: the communicator (or "group of tasks") to target
- `status`: contains the source of the message, the tag, how many elements were sent.

Basic data transfers I
Wildcards & status

Receive wildcards

- `MPI_ANY_SOURCE`: accepts data from any sender
- `MPI_ANY_TAG`: accepts data with any tag (as long as the receiver is a valid target)

Basic data transfers II
Wildcards & status

Status object

Objects of type `MPI_Status` have the following accessible fields (assume our object name is `status`):

- `MPI_SOURCE`: the rank of the process which sent the message (useful when using `MPI_ANY_SOURCE`)
- `MPI_TAG`: the tag used to identify the received message (useful when using `MPI_ANY_TAG`)
- `MPI_ERROR`: the error status (assuming the MPI program does not crash when an error is detected—which is the behavior by default).

To get the number of elements received, the user can query `status` using the `MPI_Get_count` function.

A simple example to send and receive data

```
#include <mpi.h> // required to use MPI functions
#include <stdio.h>

int main(int argc, char* argv[]) {
  int rank, data[100];

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

  if (rank == 0)
    MPI_Send(data,100,MPI_INT,1,0,MPI_COMM_WORLD);
  else
    MPI_Recv(data,100,MPI_INT,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);

  MPI_Finalize();
  return 0;
}
```

MPI is simple

- MPI_Init
- MPI_Comm_rank
- MPI_Comm_size
- MPI_Send
- MPI_Recv
- MPI_Finalize

...are enough to write any application using message passing.
However, to be *productive* and ensure reasonable *performance portability*, other functions are required.

MPI pragmatics
Building and running MPI programs

Building MPI programs

- C: `mpicc`
- C++: `mpicxx`
- Fortran: `mpif77` (Fortran 77) or `mpif90` (Fortran 90)

Running MPI programs

- `mpiexec -np 16 ./test`
  - ...will run the program `test` on 16 MPI processes.
- `mpiexec -host` *`h1,h2,...`* `-np 16 ./test`
  - ...will run the program `test` on the various hosts specified on the command line in a round-robin fashion
  - In our example, host `h1` will receive MPI processes 0,2,4,6.

Note: `mpiexec` or `mpirun` can be used interchangeably (they are aliases).

Non-blocking communications

Limits to `MPI_Send` and `MPI_Recv`

`MPI_Send` and `MPI_Recv` are blocking communication calls

- The sending process must wait until the data it is sending has been received
- The receiving process must block once it has initiated the receiving operation
- Consequence: data sent or received through blocking communications is *safe to (re)use*
- However, this can severely hamper the overall performance of an application

Non-blocking variants: `MPI_Isend` and `MPI_Irecv`

- Routine returns immediately – completion has to be tested separately
- Primarily used to overlap computation and communication

Non-blocking communications I
Syntax

API

- `int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)`
- `int MPI_Irecv(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)`
- `int MPI_Wait(MPI_Request *request, MPI_Status *status)`

Non-blocking communications II

Syntax

Properties

- Non-blocking operations allow overlapping of computation and communication
- Completion can be tested using `MPI_Test(MPI_Request *request, int flag, MPI_Status *status)`
- Anywhere one uses `MPI_Send` or `MPI_Recv`, one can use `MPI_Isend`/`MPI_Wait` or `MPI_Irecv`/`MPI_Wait` pairs instead
- Combinations of blocking and non-blocking sends/receives can be used to synchronize execution instead of barriers

Non-blocking communications III
Syntax

Multiple completions

- `int MPI_Waitall(int count, MPI_Request *array_of_requests, MPI_Status *array_of_statuses)`
- `int MPI_Waitany(int count, MPI_Request *array_of_requests, int *index, MPI_Status *status)`
- `int MPI_Waitsome(int count, MPI_Request *array_of_requests, int *array_of_indices, MPI_Status *array_of_status)`

There are corresponding versions of `MPI_Test` for each of those.

A simple example to use non-blocking communications

```
#include <mpi.h>
// ...
int main(int argc, char* argv[]) {
//     ...
  if (rank == 0) {
    for (i=0; i< 100; i++) {
      /* Compute each data element and send it out */
      data[i] = compute(i);
      MPI_ISend(&data[i], 1, MPI_INT, 1, 0, MPI_COMM_WORLD,
                &request[i]);
    }
    MPI_Waitall(100, request, MPI_STATUSES_IGNORE)
  } else {
    for (i = 0; i < 100; i++)
      MPI_Recv(&data[i], 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
               MPI_STATUS_IGNORE);
  }
//   ...
}
```

Collective operations

Introduction

- Collective operations are called by all processes belonging to the same communicator
- `MPI_Bcast` distributes data from one process (the root) to all others in a communicator
- `MPI_Reduce` combines data from all processes in the communicator and returns it to one process
- In many (numerical) algorithms, send/receive pairs can be replaced by broadcast/reduce ones
  - Simpler and more efficient

Properties

- Tags are not used; only communicators matter.
- Non-blocking collective operations were added in MPI-3
- Three classes of operations: synchronization, data movement, collective computation

Synchronization

```
int MPI_Barrier(MPI_Comm *comm)
```

- Blocks until all processes belonging to communicator `comm` call it
- No process can get out of the barrier unless all the other processes have reached it

Collective data movements

- MPI_Bcast
- MPI_Scatter
- MPI_Gather
- MPI_Allgather
- MPI_Alltoall

Collective computations

- `MPI_Reduce`
- `MPI_Scan`

Other MPI Collective routines

- Other useful collective operations
  - `MPI_Allgatherv`
  - `MPI_Alltoallv`
  - `MPI_Gatherv`
  - `MPI_Scatterv`
  - `MPI_Reducescatter`
- "All" versions deliver results to all participating processes
- "v" versions (stands for *vector*) allow the chunks to have different sizes
  - Important when the number of processes involved in the computation is no a multiple of the number of data elements
- `MPI_Allreduce`, `MPI_Reduce`, `MPI_Reducescatter`, and `MPI_Scan` take both built-in and user-defined combiner functions.

Built-in collective computation operations

MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_PROD	Product
MPI_SUM	Sum
MPI_LAND	Logical and
MPI_LOR	Logical or
MPI_LXOR	Logical exclusive or
MPI_BAND	Bitwise and
MPI_BOR	Bitwise or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and location
MPI_MINLOC	Minimum and location

Example using collective operations I

```
#include <mpi.h>
#include <math.h>
int main(int argc, char* argv[]) {
  const double g_PI25DT = 3.141592653589793238462643;
  double   mypi, pi, h, sum, x, a;
  int      n, myid, numprocs, i, ierr;

  xMPI_Init(&argc, &argv);
  xMPI_Comm_rank(MPI_COMM_WORLD, &myid);
  xMPI_Comm_size(MPI_COMM_WORLD, &numprocs);

  for (;;) {
    if (myid == 0) {
      printf("Enter␣the␣number␣of␣intervals:␣(0␣quits)␣");
      fflush(stdout);
      scanf("%f", &n);
      if (n <= 0) break;
    }

    xMPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if (n < 0) { xMPI_Finalize(); exit(0); }
```

Example using collective operations II

```
    h = 1.0/n; sum = 0.0;
    for (int i = myid+1; i < n; i+=numprocs) {
      x = h * ((double)i-0.5);
      sum += (4.0 / (1.0 * x*x));
    }
    mypi = h*sum;

    xMPI_Reduce(&mypi,&pi,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);

    if (myid == 0)
      printf("pi␣is␣%f.␣Error␣is␣%f\n", pi, fabs(pi-g_PI25DT));
  }
  xMPI_Finalize();
  return 0;
}
```

Shared-memory execution models

Introduction to OpenMP I

The OpenMP Framework

- Stands for Open MultiProcessing
- Three languages supported: C, C++, Fortran
- Supported on multiple platforms: UNIX, Linux, Windows, *etc.*
    - Very portable
- Many compilers provide OpenMP capabilities:
    - The GNU Compiler Collection (gcc) – OpenMP 3.1
    - Intel C/C++ Compiler (icc) – OpenMP 3.1 (and partial support of OpenMP 4.0)
    - Oracle C/C++ – OpenMP 3.1
    - IBM XL C/C++ – OpenMP 3.0
    - Microsoft Visual C++ – OpenMP 2.0
    - *etc.*

Introduction to OpenMP II

OpenMP's Main Components

- Compiler directives
- A functions library
- Environment variables

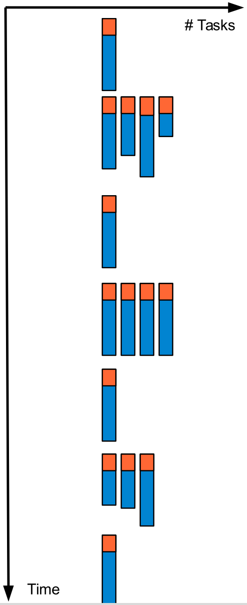
The OpenMP Model

- An OpenMP program is executed using a unique process
- Threads are activated when entering a parallel region
- Each thread executes a task composed of a pool of instructions
- While executing, a variable can be read and written in memory:
  - It can be defined in the stack of a thread: the variable is private
  - It can be stored somewhere in the heap: the variable is shared by all threads

Running OpenMP Programs: Execution Overview

OpenMP: Program Execution

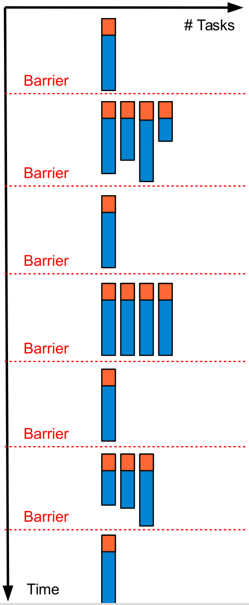
- An OpenMP program is a sequence of serial and parallel regions
- A sequential region is always executed by the master thread: Thread 0
- A parallel region can be executed by multiple tasks at a time
- Tasks can share work contained within the parallel region



Running OpenMP Programs: Execution Overview

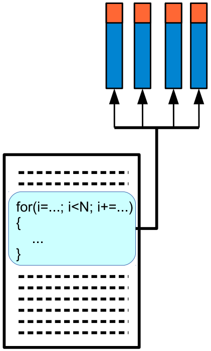
OpenMP: Program Execution

- An OpenMP program is a sequence of serial and parallel regions
- A sequential region is always executed by the master thread: Thread 0
- A parallel region can be executed by multiple tasks at a time
- Tasks can share work contained within the parallel region



OpenMP Parallel Structures

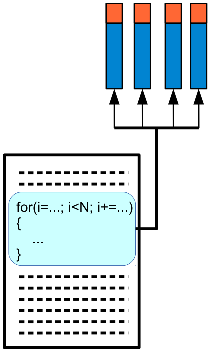
- Parallel loops



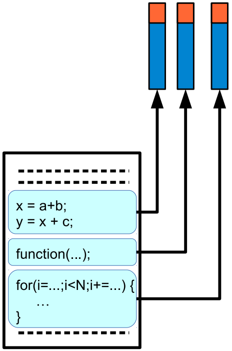
Parallel For

OpenMP Parallel Structures

- Parallel loops
- Sections



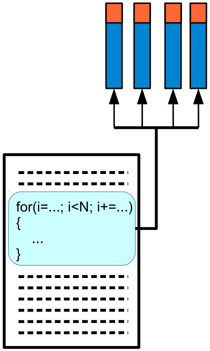
Parallel For



Sections

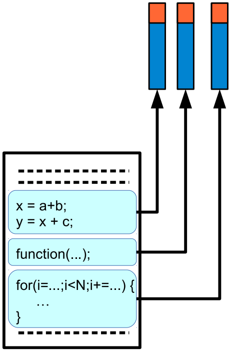
OpenMP Parallel Structures

- Parallel loops
- Sections
- Procedures through orphaning



```
for(i=...; i<N; i+=...)
{
    ...
}
```

Parallel For

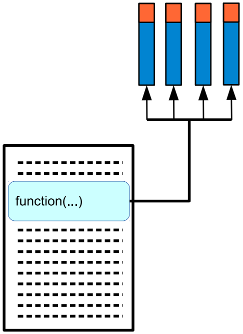


```
x = a+b;
y = x + c;
```

```
function(...);
```

```
for(i=...;i<N;i+=...) {
    ...
}
```

Sections

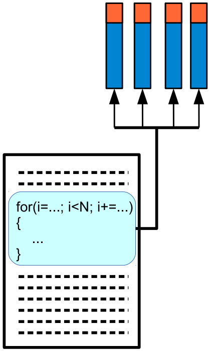


```
function(...)
```

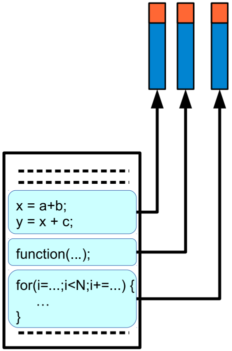
Orphan Procedures

OpenMP Parallel Structures

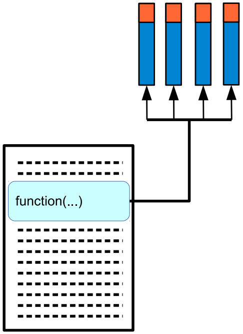
- Parallel loops
- Sections
- Procedures through orphaning
- Tasks



Parallel For



Sections



Orphan Procedures

OpenMP Structure I

Compilation Directives and Clauses

They define how to:

- Share work
- Synchronize
- Share data

They are processed as comments unless the right compiler option is specified on the command line.

Fonctions and Subroutines

They are part of a library loaded at link time

OpenMP Structure II

Environment Variables

Once set, their values are taken into account at execution time

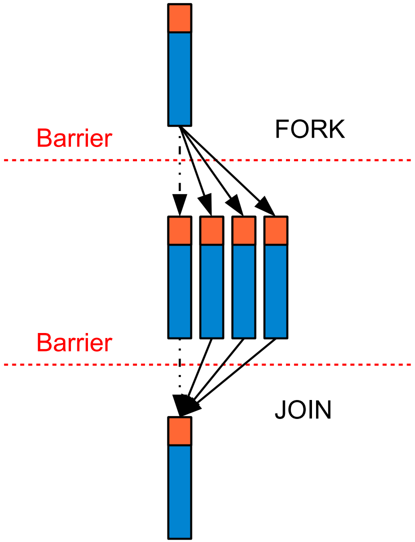
OpenMP vs. MPI I

These two programming models are complementary:

- Both OpenMP and MPI can interface using C, C++, and Fortran
- MPI is a multi-process environment whose communication mode is explicit (the user is in charge of handling communications)
- OpenMP is a multi-tasking environment whose communication between tasks is implicit (the compiler is in charge of handling communications)
- In general, MPI is used on multiprocessor machines using distributed memory
- OpenMP is used on multiprocessor machines using shared memory
- On a cluster of independent shared memory machines, combining two levels of parallelism can significantly speed up a parallel program's execution.

OpenMP: Principles

- The developer is in charge of introducing OpenMP directives
- When executing, the OpenMP runtime system builds a parallel region relying on the "fork-join" model
- When entering a parallel region, the master task spawns ("forks") children tasks which disappear or go to sleep when the parallel region ends
- Only the master task remains active after a parallel region is done



Principal Directives I

Creating a Parallel Region: the `parallel` Directive

```
#pragma omp parallel
{
        /* Parallel region code */
}
```

Principal Directives II

Data Sharing Clauses

- `shared(...)`: Comma-separated list of all variables that are to be shared by all OpenMP tasks
- `private(...)`: Comma-separated list of all variables that are to be visible only by their task.
  - Variables that are declared private are “duplicated:” their content is unspecified when entering the parallel region, and when leaving the region, the privatized variable retains the content it had *before* entering the parallel region
- `firstprivate(...)`: Comma-separated list of variables whose content must be copied (and not just allocated) when entering the parallel region.
  - The value when leaving the parallel remains the one from before entering it.
- `default(none|shared|private)`: Default policy w.r.t. sharing variables. If not specified, defaults to “shared”

A First Example: Hello World

```
szuckerm@evans201g:examples$ gcc -std=c99 -Wall -Wextra -pedantic \
                              -O3 -o omp_hello omp_hello.c
```

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#ifndef _OPENMP
#define omp_get_thread_num() 0
#endif

int main(void)
{
    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        printf("[%d]\tHello,␣World!\n", tid);
    }

    return EXIT_SUCCESS;
}
```

```
examples$ ./hello
[0] Hello, World!
[3] Hello, World!
[1] Hello, World!
[2] Hello, World!
```

Figure: omp_hello.c

Example: Privatizing Variables

```
examples:$ gcc -std=c99 -Wall -Wextra -pedantic -O3 \
               -o omp_private omp_private.c
omp_private.c: In function ‘main._omp_fn.0’:
omp_private.c:8:11: warning: ‘a’ is used  uninitialized
in this function [-Wuninitialized]
         a = a + 716.;
           ^
omp_private.c:4:11: note: ‘a’ was declared here
     float a = 1900.0;
```

```c
#include <stdio.h>
#include <omp.h>
int  main() {
    float a = 1900.0;
    #pragma omp parallel default(none) private(a)
    {
        a = a + 716.;
        printf("[%d]\ta␣=␣%.2f\n",omp_get_thread_num(), a);
    }
    printf("[%d]\ta␣=␣%.2f\n",omp_get_thread_num(), a);
    return 0;
}
```

```
[2]  a = 716.00
[1]  a = 716.00
[0]  a = 716.00
[3]  a = 716.00
[0]  a = 1900.00
```

Sharing Data Between Threads

```
examples:$ gcc -std=c99 -Wall -Wextra -pedantic -O3 \
               -o omp_hello2 omp_hello2.c
```

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#ifndef _OPENMP
#define omp_get_thread_num() 0
#endif

int main(void)
{
    int ids[] = {0, 1, 2, 3, 4, 5, 6, 7};
    #pragma omp parallel default(none) shared(ids)
    {
        printf("[%d]\tHello,␣World!\n", ids[omp_get_thread_num()]);
    }

    return EXIT_SUCCESS;
}
```

```
examples$ ./hello2
[0] Hello, World!
[3] Hello, World!
[1] Hello, World!
[2] Hello, World!
```

Figure: hello2.c

Capturing Privatized Variables' Initial Values

```
szuckerm@evans201g:examples$ gcc -std=c99 -Wall -Wextra -pedantic -O3\
                             -o omp_firstprivate omp_firstprivate.c
```

```
#include <stdio.h>
#include <omp.h>
int main() {
    float a = 1900.0;

    #pragma omp parallel \
      default(none) firstprivate(a)
    {
        a = a + 716.;
        printf("a␣=␣%f\n",a);
    }

    printf("a␣=␣%f\n",a);

    return 0;
}
```

```
examples$ ./omp_firstprivate
a = 19716.000000
a = 19716.000000
a = 19716.000000
a = 19716.000000
a = 19000.000000
```

Figure: omp_firstprivate.c

Scope of OpenMP Parallel Regions

When calling functions from a parallel region, local and automatic variables are implicitly private to each task (they belong to their respective task's stack). Example:

```
#include <stdio.h>
#include <omp.h>
void sub(void);
int main(void) {
    #pragma omp parallel default(shared)
    {
        sub();
    }
    return 0;
}
void sub(void) {
    int a  = 19716;
    a     += omp_get_thread_num();
    printf("a␣=␣%d\n", a);
}
```

Parallel Loops

```
szuckerm@evans201g:examples$ gcc -std=c99 -Wall -Wextra -pedantic -O3\
                             -o omp_for parallel_for.c
```

```
#include <stdio.h>
#include <omp.h>

int
main(void)
{
    #pragma omp parallel
    {
        int n_threads = omp_get_num_threads();
        #pragma omp for
        for (int i = 0; i < n_threads; ++i) {
            printf("[%d]\tHellow,␣World!\n", i);
        }
    }
}
```

```
examples$ ./omp_for
[1] Hellow, World!
[0] Hellow, World!
[3] Hellow, World!
[2] Hellow, World!
```

Figure: parallel_for.c

Parallel Loops: A Few Things to Remember

1. The iterator of a `omp for` loop must use additions/substractions to get to the next iteration (no `i *= 10` in the postcondition)
2. The iterator of the outermost loop (which directly succeeds to the `omp for` directive) is always private, but *not* the ones in other nested loops!
3. There is an implicit barrier at the end of the loop. You can remove it by adding the clause `nowait` on the same line: `#pragma omp for nowait`
4. How the iterations are distributed among threads can be defined using the `schedule` clause.

Parallel Loops I
Specifying the Schedule Mode

The syntax to define a scheduling policy is `schedule(ScheduleType, chunksize)`.
The final line should like this:

```
#pragma omp parallel default(none) \
                     shared(...) private(...) firstprivate(...)
{
    #pragma omp for schedule(...) lastprivate(...)
    for (int i = InitVal; ConditionOn(i); i += Stride)
    { /* loop body */ }
}

// or, all in one directive:

#pragma omp parallel for default(none) shared(...) private(...) \
                         firstprivate(...) lastprivate(...)
    for (int i = InitVal; ConditionOn(i); i += Stride) {
      /* loop body */
    }
```

Parallel Loops II
Specifying the Schedule Mode

The number of iterations in a loop is computed as follows:

$$NumIterations = \left\lfloor \frac{|FinalVal - InitVal|}{Stride} \right\rfloor + |FinalVal - InitVal| \bmod Stride$$

The number of *iteration chunks* is thus computed like this:

$$NumChunks = \left\lfloor \frac{NumIterations}{ChunkSize} \right\rfloor + NumIterations \bmod ChunkSize$$

Parallel Loops III
Specifying the Schedule Mode

Static Scheduling

`schedule(static,chunksize)` distributes the iteration chunks across threads in a round-robin fashion

- Guarantee: if two loops with the same "header" (precondition, condition, postcondition, and chunksize for the `parallel for` directive) succeed to each other, the threads will be assigned the *same* iteration chunks
- By default, `chunksize` is equal to `OMP_NUM_THREADS`
- Very useful when iterations take roughly the same time to perform (*e.g.*, dense linear algebra routines)

Dynamic Scheduling

`schedule(dynamic,chunksize)` divides the iteration space according to `chunksize`, and creates an "abstract" queue of iteration chunks. If a thread is done processing its chunk, it dequeues the next one from the queue. By default, `chunksize` is 1.
Very useful if the time to process individual iterations varies.

Parallel Loops IV

Specifying the Schedule Mode

Guided Scheduling

`guided,chunksize` Same behavior as `dynamic`, but the chunksize is divided by two each time a threads dequeues a new chunk. The minimum size is one, and so is the default.
Very useful if the time to process individual iterations varies, *and* the amount of work has a "trail"

Parallel Loops
Specifying the Schedule Mode I

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
const double MAX = 100000.;

double sum(const int n) {
    const int id = omp_get_thread_num();
    double f  = 0.0;
    const int bound = id == 0 ? n*1001 : n;

    for (int i = 0; i < bound; ++i)
      f += i;

    return f;
}
```

Parallel Loops
Specifying the Schedule Mode II

```c
int main(void) {
  printf("MAX␣=␣%.2f\n",MAX);
  double acc = 0.0;
  int* sum_until = malloc(MAX*sizeof(int));
  if (!sum_until) perror("malloc"), exit( EXIT_FAILURE );
  for (int i = 0; i < (int)MAX; ++i)  sum_until[i] = rand () % 100;
  #pragma omp parallel default(none) \
              shared(sum_until) firstprivate(acc)
  {  /* Use the OMP_SCHEDULE environment variable on the command
      * line to specify the type of scheduling you want, e.g.:
      * export OMP_SCHEDULE="static" or OMP_SCHEDULE="dynamic,10"
      * or OMP_SCHEDULE="guided,100"; ./omp_schedule
      */
    #pragma omp for schedule(runtime)
    for (int i = 0; i < bound; i+=1) {
        acc += sum( sum_until[i] );
    }
    printf ("[%d]\tMy␣sum␣=␣%.2f\n", omp_get_thread_num(), acc);
  }
  free(sum_until);
  return 0;
}
```

Figure: omp_for_schedule.c

Parallel Loops

Specifying the Schedule Mode: Outputs I

```
szuckerm@evans201g:examples$ gcc -std=c99 -Wall -Wextra -pedantic \
                             -O3 -o omp_schedule omp_for_schedule.c
szuckerm@evans201g:examples$ export OMP_NUM_THREADS=4 OMP_PROC_BIND=true
```

```
szuckerm@evans201g:examples$ export OMP_SCHEDULE="static"
szuckerm@evans201g:examples$ time ./omp_schedule
MAX = 100000.00
[0] My sum = 41299239778797.00
[1] My sum = 40564464.00
[3] My sum = 40174472.00
[2] My sum = 40502412.00

real    0m11.911s
user    0m11.930s
sys 0m0.004s
```

Parallel Loops

Specifying the Schedule Mode: Outputs I

```
szuckerm@evans201g:examples$ gcc -std=c99 -Wall -Wextra -pedantic \
                             -O3 -o omp_schedule omp_for_schedule.c
szuckerm@evans201g:examples$ export OMP_NUM_THREADS=4 OMP_PROC_BIND=true
```

```
szuckerm@evans201g:examples$ export OMP_SCHEDULE="static"
szuckerm@evans201g:examples$ time ./omp_schedule
MAX = 100000.00
[0] My sum = 41299239778797.00
[1] My sum = 40564464.00
[3] My sum = 40174472.00
[2] My sum = 40502412.00

real    0m11.911s
user    0m11.930s
sys 0m0.004s
```

```
szuckerm@evans201g:examples$ export OMP_SCHEDULE="static,1"
szuckerm@evans201g:examples$ time ./omp_schedule
MAX = 100000.00
[0] My sum = 41487115603934.00
[1] My sum = 40266669.00
[3] My sum = 40319644.00
[2] My sum = 40468898.00

real    0m11.312s
user    0m11.356s
sys 0m0.004s
```

Parallel Loops

Specifying the Schedule Mode: Outputs II

```
szuckerm@evans201g:examples$ export OMP_SCHEDULE="dynamic,1000"
szuckerm@evans201g:examples$ time ./omp_schedule
MAX = 100000.00
[0] My sum = 1661647855868.00
[1] My sum = 55011312.00
[2] My sum = 46974801.00
[3] My sum = 58218664.00

real    0m0.546s
user    0m0.576s
sys 0m0.004s
```

Parallel Loops

Specifying the Schedule Mode: Outputs II

```
szuckerm@evans201g:examples$ export OMP_SCHEDULE="dynamic,1000"
szuckerm@evans201g:examples$ time ./omp_schedule
MAX = 100000.00
[0] My sum = 1661647855868.00
[1] My sum = 55011312.00
[2] My sum = 46974801.00
[3] My sum = 58218664.00

real    0m0.546s
user    0m0.576s
sys 0m0.004s
```

```
szuckerm@evans201g:examples$ export OMP_SCHEDULE="dynamic,1"
szuckerm@evans201g:examples$ time ./omp_schedule
MAX = 100000.00
[1] My sum = 57886783.00
[0] My sum = 76809786053.00
[2] My sum = 47423265.00
[3] My sum = 56452544.00

real    0m0.023s
user    0m0.059s
sys 0m0.004s
```

Parallel Loops

Specifying the Schedule Mode: Outputs III

```
szuckerm@evans201g:examples$ export OMP_SCHEDULE="guided,1000"
szuckerm@evans201g:examples$ time ./omp_schedule
MAX = 100000.00
[0] My sum = 30922668944167.00
[3] My sum = 44855495.00
[2] My sum = 45989686.00
[1] My sum = 40596797.00

real    0m8.437s
user    0m8.452s
sys 0m0.008s
```

Parallel Loops

Specifying the Schedule Mode: Outputs III

```
szuckerm@evans201g:examples$ export OMP_SCHEDULE="guided,1000"
szuckerm@evans201g:examples$ time ./omp_schedule
MAX = 100000.00
[0] My sum = 30922668944167.00
[3] My sum = 44855495.00
[2] My sum = 45989686.00
[1] My sum = 40596797.00

real    0m8.437s
user    0m8.452s
sys 0m0.008s
```

```
szuckerm@evans201g:examples$ export OMP_SCHEDULE="guided,1"
szuckerm@evans201g:examples$ time ./omp_schedule
MAX = 100000.00
[0] My sum = 17508269385607.00
[1] My sum = 49603788.00
[2] My sum = 40584346.00
[3] My sum = 54438904.00

real    0m5.401s
user    0m5.438s
sys 0m0.008s
```

Parallel Loops

The lastprivate Clause

```
int main(void) {
    double acc = 0.0; const int bound = MAX;
    printf("[%d]\tMAX␣=␣%.2f\n",omp_get_thread_num(),MAX);
    int* sum_until = smalloc(MAX*sizeof(int));
    for (int i = 0; i < bound; ++i)
        sum_until[i] = rand () % 100;
    #pragma omp parallel for default(none) shared(sum_until) \
                  schedule(runtime) firstprivate(acc) lastprivate(acc)
    for (int i = 0; i < bound; i+=1)
        acc += sum( sum_until[i] );
    printf("Value␣of␣the␣last␣thread␣to␣write␣to␣acc␣=␣%.2f\n",acc);
    free(sum_until);
    return EXIT_SUCCESS;
}
```

Figure: omp_for_lastprivate.c

Incrementing a Global Counter I

Racy OpenMP Version

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
unsigned long g_COUNTER = 0;

int
main(void)
{
  int n_threads = 1;
  #pragma omp parallel default(none) \
              shared(n_threads,stdout,g_COUNTER)
  {
    #pragma omp master
    {
      n_threads = omp_get_num_threads();
      printf("n_threads␣=␣%d\t",n_threads); fflush(stdout);
    }

    ++g_COUNTER;
  }
  printf("g_COUNTER␣=␣%lu\n",g_COUNTER);
```

Incrementing a Global Counter II
Racy OpenMP Version

```
    return EXIT_FAILURE;
}
```

```
szuckerm@evans201g:examples$ for i in $(seq 100)
> do ./global_counter ;done|sort|uniq
n_threads = 4   g_COUNTER = 2
n_threads = 4   g_COUNTER = 3
n_threads = 4   g_COUNTER = 4
```

Incrementing a Global Counter

Using a Critical Section

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
unsigned long g_COUNTER = 0;

int main(void) {
  int n_threads = 1;
  #pragma omp parallel default(none) \
              shared(n_threads,stdout,g_COUNTER)
  {
    #pragma omp master
    {
      n_threads = omp_get_num_threads();
      printf("n_threads␣=␣%d\t",n_threads); fflush(stdout);
    }

    #pragma omp critical
    { ++g_COUNTER; }
  }
  printf("g_COUNTER␣=␣%lu\n",g_COUNTER);
  return EXIT_FAILURE;
}
```

Incrementing a Global Counter

Using an Atomic Section

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
unsigned long g_COUNTER = 0;

int main(void) {
  int n_threads = 1;
  #pragma omp parallel default(none) \
              shared(n_threads,stdout,g_COUNTER)
  {
    #pragma omp master
    {
      n_threads = omp_get_num_threads();
      printf("n_threads␣=␣%d\t",n_threads); fflush(stdout);
    }

    #pragma omp atomic
    ++g_COUNTER;
  }
  printf("g_COUNTER␣=␣%lu\n",g_COUNTER);
  return EXIT_FAILURE;
}
```

Synchronization in OpenMP I

`critical` Directive

`#pragma omp critical` *[(name)]*

Guarantees that only one thread can access the sequence of instructions contained in the (named) critical section. If no name is specified, an "anonymous" name is automatically generated.

`atomic` Directive

`#pragma omp atomic`

Guarantees the atomicity of the *single* arithmetic instruction that follows. On architectures that support atomic instructions, the compiler can generate a low-level instruction to ensure the atomicity of the operation. Otherwise, `atomic` is equivalent to `critical`.

Synchronization in OpenMP II

`barrier` Directive

`#pragma omp barrier`

All threads from a given parallel region must wait at the barrier. All `parallel` regions have an implicit barrier. All `omp for` loops do too. So do `single` regions.

`single` Directive

Guarantees that a single thread will execute the sequence of instructions located in the `single` region, and the region will be executed only once. There is an implicit barrier at the end of the region.

Synchronization in OpenMP III

`master` Directive

Guarantees that only the master thread (with $ID = 0$) will execute the sequence of instructions located in the `single` region, and the region will be executed only once. There is NO implicit barrier at the end of the region.

`nowait` Clause

`nowait` can be used on `omp for`, `single`, and `critical` directives to remove the implicit barrier they feature.

Tasking in OpenMP

OpenMP 3.x brings a new way to express parallelism: tasks.

- Tasks must be created from within a `single` region
- A task is spawned by using the directive `#pragma omp task`
- Tasks synchronize with their siblings (*i.e.*, tasks spawned by the same parent task) using `#pragma omp taskwait`

Case Study: Fibonacci Sequence

We'll use the Fibonacci numbers example to illustrate the use of tasks:

```c
/**
 * \brief Computes Fibonacci numbers
 * \param n the Fibonacci number to compute
 */
u64 xfib(u64 n) {
    return n < 2 ? // base case?
           n     : // fib(0) = 0, fib(1) = 1
           xfib(n-1) + xfib(n-2);
}
```

	Average Time (cycles)
Sequential - Recursive	196051726.08

Table: Fibonacci(37), 50 repetitions, on Intel i7-2640M CPU @ 2.80GHz

Computing Fibonacci Numbers: Headers I

utils.h, common.h, and mt.h

```
#ifndef UTILS_H_GUARD
#define UTILS_H_GUARD
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <stdint.h>
#include "rdtsc.h"

static inline void fatal(const char* msg) {
    perror(msg), exit(errno);
}
static inline void sfree(void* p) {
    if (p) { *(char*)p = 0;}  free(p);
}
static inline void* scalloc(size_t nmemb, size_t size) {
  void* p = calloc(nmemb,size);
  if (!p) { fatal("calloc"); }
  return p;
}
static inline void* smalloc(size_t size) {
```

Computing Fibonacci Numbers: Headers II

utils.h, common.h, and mt.h

```
  void* p = malloc(size);
  if (!p) { fatal("malloc"); }
  return p;
}
static inline void usage(const char* progname) {
  printf("USAGE:␣%s␣positive_number\n", progname);
  exit(0);
}
void u64_measure(u64 (*func)(u64),  u64 n,
                 u64 n_reps, const char* msg);
void u64func_time(u64 (*func)(u64), u64 n,
                  const char* msg);
#endif // UTILS_H_GUARD
```

Computing Fibonacci Numbers: Headers III

utils.h, common.h, and mt.h

```
#ifndef COMMON_H_GUARD
#define COMMON_H_GUARD
#include "utils.h" // for smalloc(), sfree(), fatal(), scalloc(), ...
#define FIB_THRESHOLD 20

typedef uint64_t u64; typedef uint32_t u32; typedef uint16_t u16;
typedef uint8_t   u8; typedef  int64_t s64; typedef  int32_t s32;
typedef  int16_t s16; typedef  int8_t   s8;

u64   xfib(u64);     u64   trfib(u64,u64,u64);
u64   trFib(u64);    u64   sfib(u64);
u64   memoFib(u64); u64   memofib(u64,u64*);

void* mt_memofib(void*); u64   mt_memoFib(u64);
void* mtfib(void*);      u64   mtFib(u64);

u64   oFib(u64);       u64   ofib(u64);
u64   o_memoFib(u64); u64   o_memofib(u64,u64*);
```

Computing Fibonacci Numbers: Headers IV

`utils.h`, `common.h`, and `mt.h`

```
#endif /* COMMON_H_GUARD  */
#ifndef MT_H_GUARD
#define MT_H_GUARD
#include <pthread.h>
typedef struct fib_s     { u64 *up,        n; }     fib_t;
typedef struct memofib_s { u64 *up, *vals, n; } memofib_t;
static inline pthread_t* spawn(void* (*func)(void*), void* data) {
    pthread_t* thread = smalloc(sizeof(pthread_t)); int error = 0;
    do {
      errno = error = pthread_create(thread,NULL,func,data);
    } while (error == EAGAIN);
    if (error) fatal("pthread_create");
    return thread;
}
static inline void sync(pthread_t* thread) {
    int error = 0; void* retval = NULL;
    if ( (errno = ( error = pthread_join(*thread, &retval) ) ) )
        fatal("pthread_join");
    sfree(thread);
```

Computing Fibonacci Numbers: Headers V

`utils.h`, `common.h`, and `mt.h`

```
}
#endif // MT_H_GUARD
```

Computing Fibonacci Numbers: Code I

Naïve Pthread and OpenMP

```c
#include "common.h"
#include "mt.h"
void* mtfib(void* frame) {
    fib_t* f = (fib_t*) frame;
    u64 n  = f->n, *up = f->up;
    if (n < FIB_THRESHOLD)
        *up = sfib(n), pthread_exit(NULL);
    u64 n1 = 0, n2 = 0;
    fib_t frame1 = { .up = &n1, .n = f->n-1 },
          frame2 = { .up = &n2, .n = f->n-2 };
    pthread_t *thd1 = spawn(mtfib,&frame1),
              *thd2 = spawn(mtfib,&frame2);
    sync(thd1); sync(thd2);
    *up = n1+n2;
    return NULL;
}
u64  mtFib(u64 n) {
    u64 result = 0; fib_t f = { .up = &result, .n = n };
    (void)mtfib(&f);
    return result;
}
```

Computing Fibonacci Numbers: Code II

Naïve Pthread and OpenMP

```
#include "common.h"
#include <omp.h>

u64 ofib(u64 n) { u64 n1, n2;
    if (n < FIB_THRESHOLD)  return sfib(n);
#   pragma omp task shared(n1)
    n1 = ofib(n-1);
#   pragma omp task shared(n2)
    n2 = ofib(n-2);
#   pragma omp taskwait
    return n1 + n2;
}

u64 oFib(u64 n) { u64 result = 0;
#   pragma omp parallel
    {
#       pragma omp single nowait
        { result = ofib(n); }
    } // parallel
    return result;
}
```

Computing Fibonacci Numbers: Code III

Naïve Pthread and OpenMP

	Average Time (cycles)
Sequential - Recursive	196051726.08
Parallel - PThread	2837871164.24
Parallel - OpenMP	17707012.14

Table: Fibonacci(37), 50 repetitions, on Intel i7-2640M CPU @ 2.80GHz

Computing Fibonacci Numbers: Code I

Memoization using Serial, Pthread and OpenMP

```
#include "common.h"
#include "mt.h"
void* mt_memofib(void* frame) { memofib_t* f = (memofib_t*) frame;
  u64 n = f->n *vals = f->vals, *up = f->up;
  if (n < FIB_THRESHOLD)  *up = vals[n] = sfib(n), pthread_exit(NULL)
  if (vals[n] == 0) { u64 n1 = 0, n2 = 0;
    memofib_t frame1 = {.up=&n1,.n=f->n-1,.vals=vals},
              frame2 = {.up=&n2,.n=f->n-2,.vals=vals};
    pthread_t *thd1  = spawn(mt_memofib,&frame1),
              *thd2  = spawn(mt_memofib,&frame2);
    sync(thd1); sync(thd2);
    vals[n] = n1 + n2; }
  *up = vals[n], pthread_exit(NULL);
}
u64 mt_memoFib(u64 n) { u64 result = 0;
    u64* fibvals = scalloc(n+1,sizeof(u64));
    fibvals[1]=1; memofib_t f={.up=&result,.n=n,.vals=fibvals};
    (void)mt_memofib(&f);
    return result;
}
```

Computing Fibonacci Numbers: Code II

Memoization using Serial, Pthread and OpenMP

```
#include "common.h"
#include <omp.h>
u64 o_memofib(u64 n, u64* vals) {
  if (n < FIB_THRESHOLD) return sfib(n);
  if (vals[n] == 0) { u64 n1 = 0, n2 = 1;
#   pragma omp task shared(n1,vals)
    n1 = o_memofib(n-1,vals);
#   pragma omp task shared(n2,vals)
    n2 = o_memofib(n-2,vals);
#   pragma omp taskwait
    vals[n] = n1 + n2;
  }
  return vals[n];
}
u64 o_memoFib(u64 n) {
  u64 result=0, *fibvals=calloc(n+1,sizeof(u64));
# pragma omp parallel
  {
#   pragma omp single nowait
    { fibvals[1] = 1; result = o_memofib(n,fibvals); }
  }
  return result; }
```

Computing Fibonacci Numbers: Code III

Memoization using Serial, Pthread and OpenMP

```
#include "common.h"

u64 memofib(u64 n, u64* vals) {
    if (n < 2)
        return n;
    if (vals[n] == 0)
        vals[n] = memofib(n-1,vals) + memofib(n-2,vals);
    return vals[n];
}

u64  memoFib(u64 n) {
    u64* fibvals = calloc(n+1,sizeof(u64));
    fibvals[0] = 0; fibvals[1] = 1;
    u64 result = memofib(n,fibvals);
    sfree(fibvals);
    return result;
}
```

Computing Fibonacci Numbers: Code IV

Memoization using Serial, Pthread and OpenMP

	Average Time (cycles)
Sequential - Recursive	196051726.08
Parallel - PThread	2837871164.24
Parallel - OpenMP	17707012.14
Parallel - PThread - Memoization	2031161888.56
Parallel - OpenMP - Memoization	85899.58
Sequential - Memoization	789.70

Table: Fibonacci(37), 50 repetitions, on Intel i7-2640M CPU @ 2.80GHz

Computing Fibonacci Numbers: Code I

When Serial is MUCH Faster Than Parallel

```
#include "common.h"
u64 trfib(u64 n, u64 acc1, u64 acc2) {
    return n < 2 ?
           acc2  :
           trfib( n-1, acc2, acc1+acc2);
}
u64 trFib(u64 n) {  return trfib(n, 0, 1);  }
```

```
#include "common.h"
u64  sfib(u64 n) {
    u64 n1 = 0, n2 = 1, r = 1;
    for (u64 i = 2; i < n; ++i) {
        n1 = n2;
        n2 = r;
        r  = n1 + n2;
    }
    return r;
}
```

Computing Fibonacci Numbers: Code II

When Serial is MUCH Faster Than Parallel

	Average Time (cycles)
Sequential - Recursive	196051726.08
Parallel - PThread	2837871164.24
Parallel - OpenMP	17707012.14
Parallel - PThread - Memoization	2031161888.56
Parallel - OpenMP - Memoization	85899.58
Sequential - Memoization	789.70
Sequential - Tail Recursive	110.78
Sequential - Iterative	115.02

Table: Fibonacci(37), 50 repetitions, on Intel i7-2640M CPU @ 2.80GHz

Learning More About Multi-Threading and OpenMP

Books (from most theoretical to most practical)

- Herlihy and Shavit 2008
- Rünger and Rauber 2013
- Kumar 2002
- Chapman, Jost, and Pas 2007

Internet Resources

- "The OpenMP® API specification for parallel programming" at openmp.org
  - Provides all the specifications for OpenMP, in particular OpenMP 3.1 and 4.0
  - Lots of tutorials (see http://openmp.org/wp/resources/#Tutorials)
- The Wikipedia article at http://en.wikipedia.org/wiki/OpenMP

Food for Thoughts

- Sutter 2005 (available at http://www.gotw.ca/publications/concurrency-ddj.htm)
- Lee 2006 (available at http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf)
- Boehm 2005 (available at http://www.hpl.hp.com/techreports/2004/HPL-2004-209.pdf)

References

Bibliography I

References

Forum, Message Passing (1994). *MPI: A Message-Passing Interface Standard*. Tech. rep. Knoxville, TN, USA.

Dongarra, Jack J. et al. (1995). *An Introduction to the MPI Standard*. Tech. rep. Knoxville, TN, USA.

Dagum, Leonardo and Ramesh Menon (1998). “OpenMP: an industry standard API for shared-memory programming”. In: *IEEE computational science and engineering* 5.1, pp. 46–55.

Gropp, William, Ewing Lusk, and Anthony Skjellum (1999). *Using MPI: portable parallel programming with the message-passing interface*. Vol. 1. MIT press.

Bibliography II

Gropp, William, Ewing Lusk, and Rajeev Thakur (1999). *Using MPI-2: Advanced Features of the Message-Passing Interface*. Cambridge, MA, USA: MIT Press. ISBN: 0262571331.

Kumar, Vipin (2002). *Introduction to Parallel Computing*. 2nd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0201648652.

Huang, Chao, Orion Lawlor, and Laxmikant V Kale (2003). "Adaptive mpi". In: *International workshop on languages and compilers for parallel computing*. Springer, pp. 306–322.

Boehm, Hans-J. (June 2005). "Threads Cannot Be Implemented As a Library". In: *SIGPLAN Not.* 40.6, pp. 261–268. ISSN: 0362-1340. DOI: 10.1145/1064978.1065042. URL: http://doi.acm.org/10.1145/1064978.1065042.

Bibliography III

Sutter, Herb (2005). "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software". In: *Dr. Dobb's Journal* 30.3.

Lee, Edward A. (May 2006). "The Problem with Threads". In: *Computer* 39.5, pp. 33–42. ISSN: 0018-9162. DOI: 10.1109/MC.2006.180. URL: http://dx.doi.org/10.1109/MC.2006.180.

Chapman, Barbara, Gabriele Jost, and Ruud van der Pas (2007). *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press. ISBN: 0262533022, 9780262533027.

Chapman, Barbara, Gabriele Jost, and Ruud Van Der Pas (2008). *Using OpenMP: portable shared memory parallel programming*. Vol. 10. MIT press.

Bibliography IV

Herlihy, Maurice and Nir Shavit (2008). *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN: 0123705916, 9780123705914.

Ayguade, E. et al. (Mar. 2009). "The Design of OpenMP Tasks". In: *IEEE Transactions on Parallel and Distributed Systems* 20.3, pp. 404–418. ISSN: 1045-9219. DOI: 10.1109/TPDS.2008.105.

Graham, Richard L. (2009). "The MPI 2.2 Standard and the Emerging MPI 3 Standard". In: *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Espoo, Finland: Springer-Verlag, pp. 2–2. ISBN: 978-3-642-03769-6. DOI: 10.1007/978-3-642-03770-2_2. URL: http://dx.doi.org/10.1007/978-3-642-03770-2_2.

Bibliography V

Duran, Alejandro et al. (2011). “Ompss: a proposal for programming heterogeneous multi-core architectures”. In: *Parallel Processing Letters* 21.02, pp. 173–193.

Rünger, Gudula and Thomas Rauber (2013). *Parallel Programming - for Multicore and Cluster Systems; 2nd Edition*. Springer. ISBN: 978-3-642-37800-3. DOI: 10.1007/978-3-642-37801-0. URL: http://dx.doi.org/10.1007/978-3-642-37801-0.

Gropp, William, Torsten Hoefler, et al. (2014). *Using Advanced MPI: Modern Features of the Message-Passing Interface*. The MIT Press. ISBN: 0262527634, 9780262527637.

Gropp, William, Ewing Lusk, and Anthony Skjellum (2014). *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press. ISBN: 0262527391, 9780262527392.