Simulation and Scientific Computing 2
Seminar

Kristina Pickl, Dominik Bartuschat, Christoph Rettinger, Ulrich Rüde
SS 2015
Chair for System Simulation (LSS)

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Outline

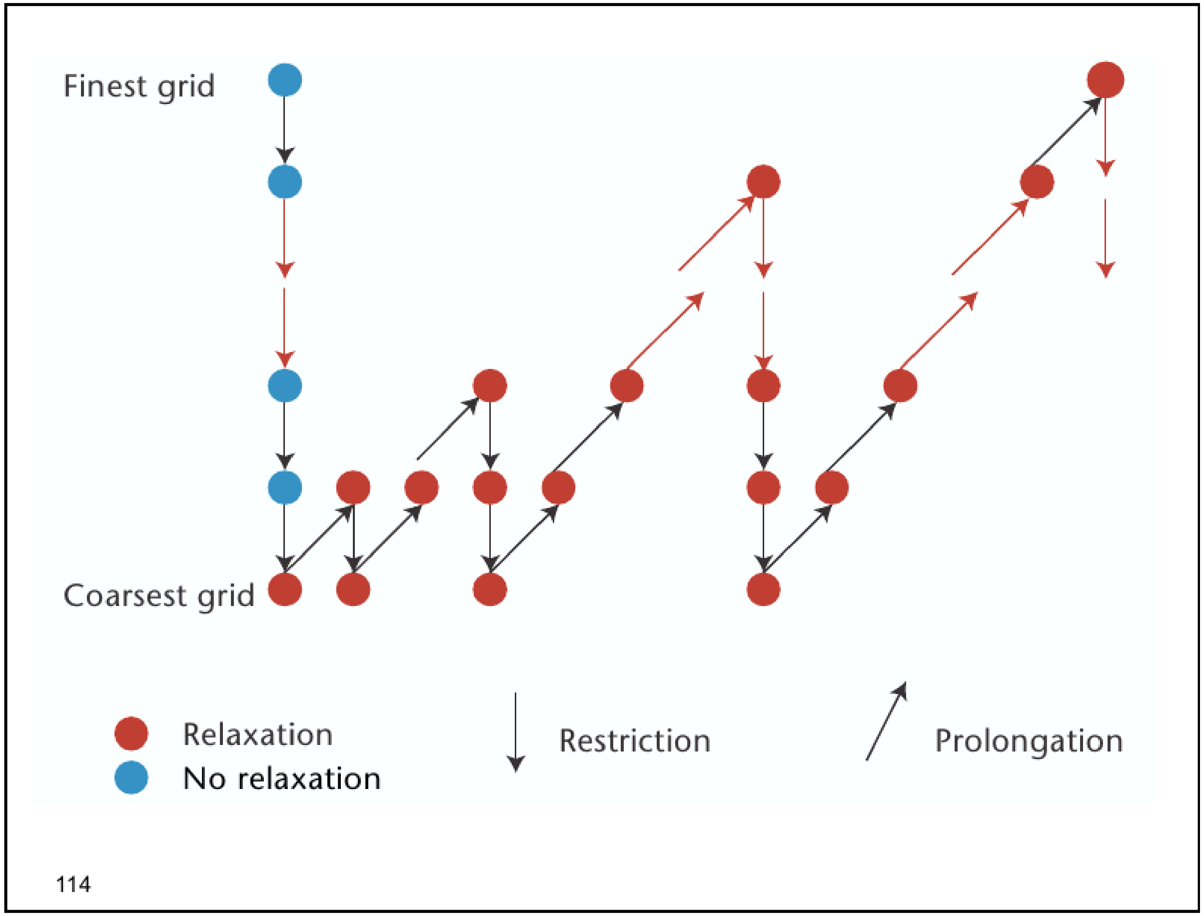
Full Multi-Grid (FMG)

Introduction to Parallel Programming

- Distributed vs. Shared Memory
- OpenMP
- MPI-Message Passing Interface

Full Multi-Grid (FMG)

Finite State Machine

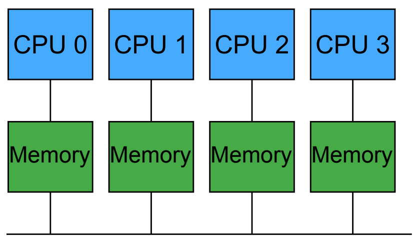


Introduction to Parallel Programming

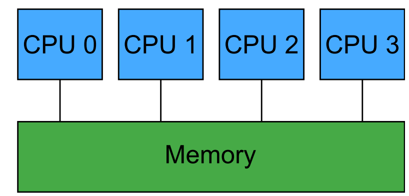
FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Distributed vs. Shared Memory



- Hardware
  - same program on each processor/machine
  - explicit programming required
- Programming
  - all variables process-local, no implicit knowledge of data on other processors
  - send/receive messages of suitable library
- Languages
  - e.g. MPI
  - `http://www.mpi-forum.org/`



- Hardware
  - single program on single machine
  - workload distributed among threads
- Programming
  - all variables either shared among all threads or duplicated for each thread
  - threads communicate by sharing variables
- Languages
  - e.g. OpenMP
  - `http://www.openmp.org/`

OpenMP Programming Model

- based on the `#pragma` compiler directives

```
#pragma omp directive [clause list]
```

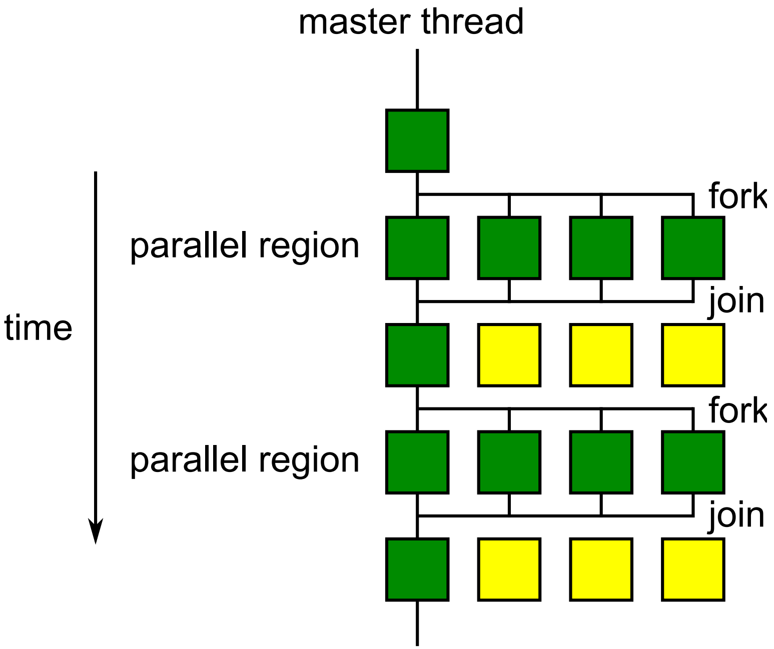
- OpenMP programs execute serially until they encounter `parallel` directive

```
/* serial code segment

#pragma omp parallel [clause list]
{
  /* parallel code segment
}

/* rest of serial code segment */
```

- each thread executes structured block specified by the parallel directive
- the `clause list` specifies the conditional parallelization, number of threads and data handling

Execution Model

Fork-Join Model



Data Handling

- `private ( variable list )`: indicates that the set of variables is local to each thread (i.e. each thread has its own copy of each variable in the list)
- `shared ( variable list )`: indicates that all variables in the list are shared across all the threads, i.e., there is only one copy

```
#include <omp.h>
int main()
{
  int i=5; // a shared variable

  #pragma omp parallel
  {
    int c; // a variable private to each thread
    c = omp_get_thread_num();
    std::cout << "c: " << c << ", i: " << i << std::endl;
  }
}
```

Data Handling

- `reduction ( operator list )`: specifies how multiple local copies of a variable are combined into a single copy at the master when threads exit
- possible operators: +, *, -, &, |, &&, ||

```
#include <omp.h>
int main()
{
  double sum ( 0.0 );

  #pragma omp parallel reduction (+: sum)
  {
    /* compute local sums here */
  }
  /* sum here contains sum of all local instances of sum */
}
```

Synchronization Constructs

- `#pragma omp barrier`
  - synchronizes all threads in the team
- `#pragma omp single`
  - only executed by exactly one thread
  - all other threads skip this region
  - implicit barrier at end of single construct
- `#pragma omp master`
  - only executed by the master thread
  - all other threads skip this region
  - no implicit barrier associated
- `#pragma omp critical`
  - executed by all threads
  - but only one thread at a time

Runtime Library Functions

- Setting total number of threads
  - at runtime: via `omp_set_num_threads`

    ```
    #include <omp.h>
    void omp_set_num_threads( int num_threads )
    ```

  - via environment variable `OMP_NUM_THREADS`

    ```
    export OMP_NUM_THREADS=4
    ```

- getting total number of threads

  ```
  #include <omp.h>
  void omp_get_num_threads( void )
  ```

- getting ID of specific thread

  ```
  #include <omp.h>
  void omp_get_thread_num( void )
  ```

MPI Programming Model

- each processor runs a sub-program
- variables of each sub-program have
  - the same name
  - but different locations and different data
  - i.e. all variables are private
- communicate via special send and receive routines

Hello World

```
#include <mpi.h>
int main( int argc,char **argv )
{
  // Definition of the variables
  int size; //The total number of processes
  int rank; //The rank/number of this process

  // MPI initialization
  MPI_Init( &argc, &argv );

  // Determining the number of CPUs and the rank for each CPU
  MPI_Comm_size( MPI_COMM_WORLD, &size );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  ...
```

Hello World

```
  ...
  // 'Hello World' output for CPU 0
  if( rank == 0 )
    std::cout << "Hello World" << std::endl;

  // Output of the own rank and size of each CPU
  std::cout << "I am CPU " << rank << " of " << size << " CPUs" << std::endl;

  // MPI finalizations
  MPI_Finalize();
  return 0;
}
```

Blocking Operations

- some operations may block until another process acts:
  - synchronous send operation blocks until receive is posted
  - receive operation blocks until message is sent
- send buffer may be reused after `MPI_Send` returns
- MPI call returns after completion of the corresponding send/receive operation

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

Non-Blocking Operations

- return immediately and allow the sub-program to perform other work
- at some later time the sub-program must test or wait for the completion of non-blocking operation
- all non-blocking operations must have matching wait (or test) operations
- a non-blocking operation immediately followed by a matching wait is equivalent to a blocking operation

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

Collective Communications

- Broadcast operation: a one-to-many communication

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

- Reduction operation: combine data from several processes to produce single result

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

- Barriers: synchronizes processes, blocks until all processes in the communicator have reached this routine

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

- Wait: waits for an MPI send or receive to complete

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

Thank you for your attention!

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

References

- MPI:
  - `http://www.mpi-forum.org/`
  - `https://www10.informatik.uni-erlangen.de/Teaching/Courses/WS2014/SiWiR/exerciseSheets/ex03/mpi.pdf`
- OpenMP:
  - `http://www.openmp.org/`
  - `https://computing.llnl.gov/tutorials/openMP/`
  - `https://www10.informatik.uni-erlangen.de/Teaching/Courses/WS2014/SiWiR/exerciseSheets/ex02/ex02_1.pdf`