Parallel and Distributed Programming

MPI

Message Passing Interface

- ”send-receive” paradigm
  - sending the message:
    - send (target, id, data)
  - receiving the message:
    - receive (source, id, data)
- Versatility of the model
- High efficiency and scalability of calculations
- MPMD or SPMD programs
- Specific programming environment with tools to run programs (often also compile and debug)

Message Passing Interface

- Programming interface that defines the relationship with C, C + + and Fortran
- Standardization and extension of the previous programming solutions for the transmission of messages (PVM, P4, Chameleon, Express, Linda) to obtain:
  - portability of parallel programs
  - completeness of the interface
  - high-performance computation
  - ease of parallel programming
- Developed in years: 1992-1995 MPI-1 and MPI-2 1995-1997

Message Passing Interface

- Basic concepts:
  - communicator (predefined communicator MPI_COMM_WORLD)
  - rank of the process
- Basic procedures:
  - int MPI_Init(int *pargc, char ***pargv)
  - int MPI_Comm_size(MPI_Comm comm, int *psize)
  - int MPI_Comm_rank(MPI_Comm comm, int *prank)
  - (0 <= *prank < *psize)
  - int MPI_Finalize(void)

Message Passing Interface

```
#include "mpi.h"
int main( int argc, char** argv )
{
    int rank, size, source, dest, tag, i;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf("My number is %d, in a group of %d processes\n", rank, size);
    MPI_Finalize();
    return(0);
}
```

Message Passing Interface

- The procedures for the two-point (point-to-point) messaging:
  - MPI guarantees progress in the implementation of the message (after the correct initialization of send-receive pair at least one of them will be completed)
  - MPI guarantees the order of receiving (in order of send) for messages from the same source, with the same ID and within the same communicator
  - MPI does not guarantee fairness when receiving messages from a different sources
  - During the realizations of procedures for the message passing error can occur associated with crossing the limits of available system resources

Message Passing Interface

- The procedures for the two-point messaging:
  - Blocking - the procedure waits until communication operations have been completed and you can safely use buffers (variables) that are operands
  - int MPI_Send (void * buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm)
  - int MPI_Recv (void * buf, int count, MPI_Datatype dtype, int src, int tag, comm MPI_Comm, MPI_Status * stat)

Message Passing Interface

```
#include "mpi.h"
int main( int argc, char** argv )
{
    int rank, ranksent, size, source, dest, tag, i; MPI_Status status;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    if( rank != 0 ){ dest=0; tag=0;
        MPI_Send( &rank, 1, MPI_INT, dest, tag, MPI_COMM_WORLD );
    } else {
        for( i=1; i<size; i++ ) { MPI_Recv( &ranksent, 1, MPI_INT, MPI_ANY_SOURCE,
                                  MPI_ANY_TAG, MPI_COMM_WORLD, &status );
        printf("Data from process number %d (%d)\n", ranksent, status.MPI_SOURCE );
        }
    }
    MPI_Finalize(); return(0);
}
```

Message Passing Interface

- The procedures for the two-point messaging:
  - Non-blocking - the procedure immediately transfers control to further instructions of the program
    - MPI_Isend int (void * buf, int count, MPI_Datatype dtype, int dest, int tag, comm MPI_Comm, MPI_Request * req)
    - MPI_Irecv int (void * buf, int count, MPI_Datatype dtype, int src, int tag, comm MPI_Comm, MPI_Request * req)
  - The role of the variable *req
  - As a part of the send-receive pairs one can connect any combination of blocking and non-blocking procedures

Message Passing Interface

- Procedures associated with the non-blocking message passing:
  - int MPI_Wait (MPI_Request * Preq, MPI_Status * pstat)
  - int MPI_Test (MPI_Request * Preq, int * pflag, MPI_Status * pstat) (result of the test in *pflag variable)
  - Additional variants of the above: MPI_Waitany, MPI_Waitall, MPI_Testany, MPI_Testall
- Procedures for testing the arrival of messages (without answering) - two types - blocking and non-blocking
  - MPI_Probe int (int src, int tag, comm MPI_Comm, MPI_Status *stat)
  - MPI_Iprobe int (int src, int tag, MPI_Comm comm, int * flag, MPI_Status *stat)
- And many others (MPI_Send_init, MPI_Start, MPI_Sendrecv, MPI_Cancel, etc.)

Message Passing Interface

- The essence of the example is to overlap computation and communication, which in some cases can significantly improve the performance of the program

```
MPI_Request request1, request2; MPI_Status status1, status2;
MPI_Irecv( &datarecv, num, MPI_INT, source, tag, MPI_COMM_WORLD, &request1 );
// computations which does not require received data
MPI_Isend( &datasent, num, MPI_INT, dest, tag, MPI_COMM_WORLD, &request2 );
// computations that does not change data sent
MPI_Wait( &request1, &status1 );
printf("Data form process rank: %d (%d)\n", source, status.MPI_SOURCE );
MPI_Wait( &request2, &status2 );
printf(„Data sent to process rank: %d \n", dest );
```

Overlapping example

- Memory areas for process: A - private data (updated), C – received data, B – sent data (updated using received data C)
- Case 1: naive operation in the order C, A, B –

  proc.1 - recv C, proc.2 - recv C -> deadlock
- Case 2: in order to avoid deadlocks we operate:

  update A, send B, receive C,

  update B - it turns out that we have to wait
- Case 3: overlap computation and communication - Isend B, Irecv C, update A, wait B, wait C, updating B.