# Combining OpenMP and Message Passing: Hybrid Programming

Michael Hanke

Summer School on High Performance Computing

# Outline

# What You Already Know

- Parallelizing programs on shared memory computers using OpenMP
- Parallelizing programs on distributed memory computers using MPI

# What Is Covered in This Module?

- Multithreaded MPI
  - Support provided by the implementation
  - How to use it
- Programming recommendations

# Mainstream HPC Architecture

- General purpose processors are not getting (very much) faster
- The optimal (price/performance) HPC hardware goes to massively parallel computers (MPPs):
  - many compute nodes coupled by high-speed interconnects
  - each compute node is a multi-socket shared memory node
  - each socket holds a multi-core processor
- Hybrid architectures:
  - Coupling of "standard" compute nodes with highly specilized computes nodes (cell processors, general purpose GPUs)

# Programming Models

**OpenMP** Shared memory, symmetric multi-processors

- Corresponds to one multi-core processor

**MPI** Distributed memory, processing nodes coupled via comunication channels

- Corresponds to MPPs

These are logical programming models and can be mapped to any hardware!

## Problem

Can one combine these two models in order to obtain faster codes by using the architecture specifics?

# Reasons to Combine MPI and OpenMP

1. This software approach matches the hardware trend.
2. Some applications expose two levels of parallelism: coarse-grained (suitable for MPI), and fine-grained (best suited for OpenMP)
3. Application requirements or system restrictions may limit the number of MPI processes. Thus, OpenMP can offer an additional amount of parallelism.
4. Some application show unbalanced workload at the MPI level. OpenMP can be used to address this issue by assigning a different number of threads to each MPI process.

# Reasons Not to Combine MPI and OpenMP

1. Introducing OpenMP into an existing MPI code also means introducing the drawbacks of OpenMP, such as the following:
   - Limitations when it comes to control of work distribution and synchronization
   - Overhead introduced by thread creation and synchronization
   - Dependence on quality of compiler and runtime support for OpenMP
   - Shared memory issues (ccNUMA architectures)
2. The interaction of MPI and OpenMP runtime libraries may have negative side effects on the program's performance.
3. Some appliations naturally expose only one level of parallelism, and there may be no benefit in introducing a hierarchical parallelism.

# Development Steps

1. Start from existing sequential code.
2. Decompose the problem for MPI parallelization.
3. Add OpenMP directives later.
   - Only the master thread will perform communication between MPI tasks (limitations to efficiency!)
   - Least error-prone: Use MPI only outside of parallel regions
   - Is the MPI runtime system thread-safe??

# Hybrid Programming Models

1. No overlapping communication and computation
   - 1.1 MPI is called only outside parallel regions and on the master thread
   - 1.2 MPI is called by several threads
2. Communication and computation overlap: while some of the threads communicate, the rest are executing an application
   - 2.1 MPI is called only be the master thread
   - 2.2 Communication is carried out with several threads
   - 2.3 Each thread handles its own communication demands

# MPI Support For Threading

- MPI standard defines four levels of support

  `MPI_THREAD_SINGLE` Only one thread allowed

  `MPI_THREAD_FUNNELED` Only master thread allowed to make MPI calls

  `MPI_THREAD_SERIALIZED` All threads allowed to make MPI calls, but not concurrently

  `MPI_THREAD_MULTIPLE` No restrictions

- Some implementations support an additional model:
  - 1.5: MPI calls are allowed only outside of parallel regions
  - Returns `MPI_THREAD_SINGLE`

# MPI Initialization

## C

```
int MPI_Init_thread(int *argc, char **argv[],
        int required, int *provided)
```

## Fortran

```
MPI_INIT_THREAD(required, provided, ierror)
INTEGER required, provided, ierror
```

- `required` - required level of threading support
- `provided` - implementation provided best match for `required`

# Support on Ferlin

Code snippet:

```
MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE,
                &provided)
printf(“Supports level %d of %d %d %d %d\n”,
       provided,
       MPI_THREAD_SINGLE,
       MPI_THREAD_FUNNELED,
       MPI_THREAD_SERIALIZED,
       MPI_THREAD_MULTIPLE);
```

Output on Ferlin using openmpi 1.3:

```
Supports level 3 of 0 1 2 3
```

Note: Certain implementation dependent restrictions apply.

# Hello World (C)

```
int main(int argc, char *argv[]) {
  int rank, omp_rank, mpisupport;
  MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE,
                  &mpisupport)
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
#pragma omp parallel private(omp_rank)
  {
    omp_rank = omp_get_thread_num();
    printf(“Hello. This is process %d, thread %d\n”,
           rank, omp_rank);
  }
  MPI_Finalize();
}
```

# Hello World (F77)

```
      PROGRAM hello
      IMPLICIT NONE
      INCLUDE 'mpif.h'
      INCLUDE 'omp_lib.h'
      INTEGER rank, omp_rank, mpisupport, ierror
C
      CALL mpi_init_thread(MPI_THREAD_MULTIPLE,
     &                     mpisupport, ierror)
      CALL mpi_comm_rank(MPI_COMM_WORLD, rank, ierror)
!$OMP parallel private(omp_rank)
      omp_rank = omp_get_thread_num()
      WRITE (*,1) rank, omp_rank
1     FORMAT ('Hello. This is process ',I2,',
     &       thread ',I2)
!$OMP end parallel
C
      CALL mpi_finalize(ierror)
      END
```

# Remarks

- Neither the MPI standard nor the OpenMP standard contain any provisions for placement of tasks or threads.
- Task placement must be controlled by the user, e.g.,
  - in openmpi, use `mpirun -bynode <remaining options> <program>`
- For efficient behavior of OpenMP, e.g., thread migration must be avoided.
  - In GNU gcc, use `GOMP_CPU_AFFINITY`
  - In Intel icpc, use `KMP_AFFINITY`
  - On linux, use `numactl`, `taskset`

## Fact

*The careful use of these tools is essential in order to obtain the desired efficiency of hybrid programming.*

# NAS Parallel Benchmark

- Simulated 3D CFD simulation
- Uses ADI in 3D to solve the discrete Navier-Stokes equations
- Parallelization by domain decomposition
- Consider the multi-zone benchmarks
- Zone size varies widely, therefore, a load-balancing algorithm is used

# NPB on Ranger at TACC

- DDR infiniband network
- 3936 compute blades
- Each node: 4 2.3GHz AMD "Barcelona" quad core
- Peak performance: 579 TFlops
- MVAPICH, numactl
- PGI F90 7.1
- Class E benchmark: 4224 x 3456 x 92 points in 4096 zones
- Equally-sized zones (SP) and zones of varyimg size (BT)

# NPB on Ranger, Results

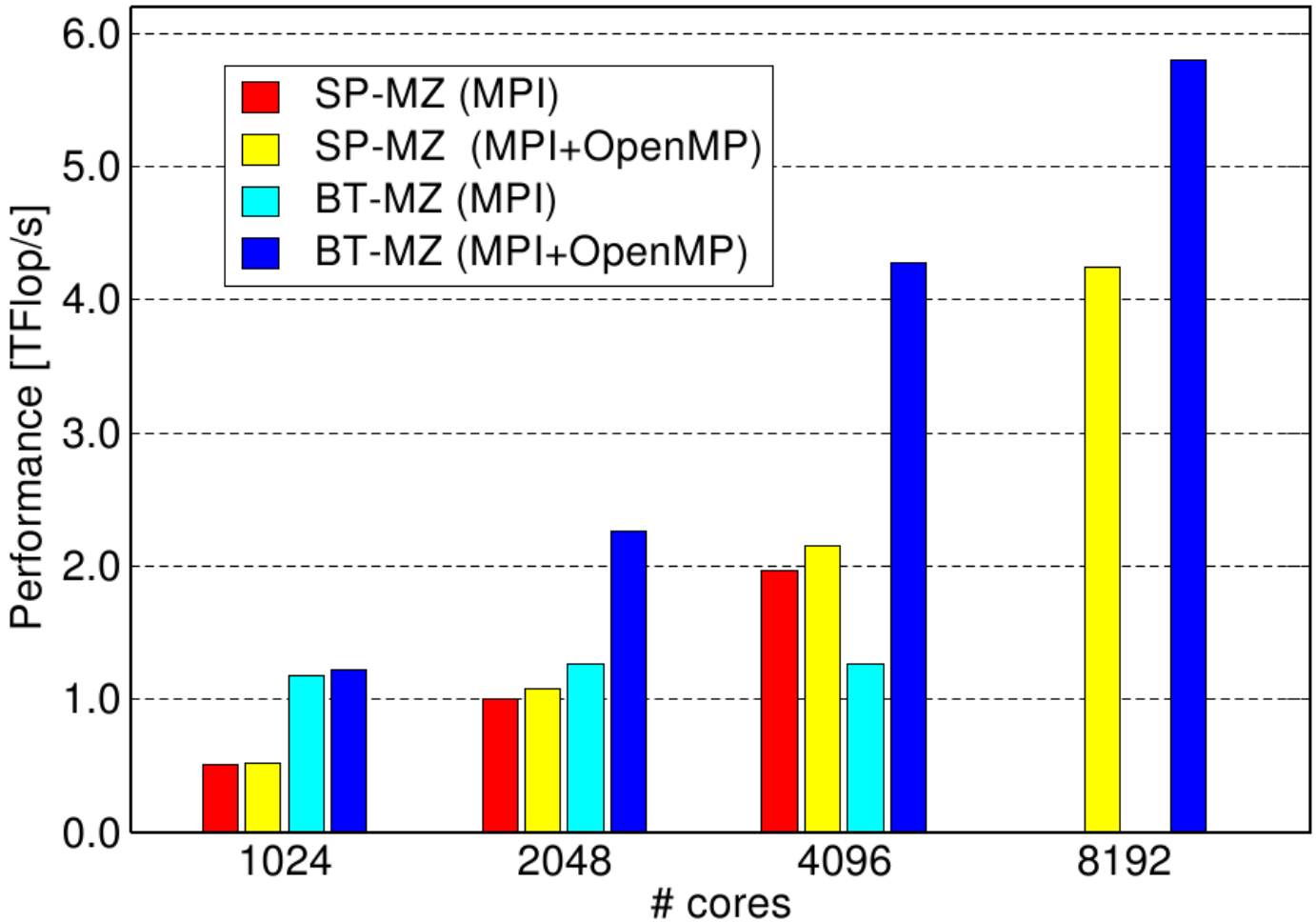


Figure from: Ralf Rabenseifner, Georg Hager, Gabriele Jost: Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes

# Conclusions

- Pure OpenMP performs better than pure MPI within node is a necessity to have hybrid code better than pure MPI across node.

  But not sufficient...
- It is often very hard to make hybrid programs running faster than pure MPI solutions.
- One needs to use hardware specific tuning.
- Hybrid programming includes both programming techniques and system tools.