MPI on Hybrid Systems

MPI + OpenMP and other models on clusters of SMP nodes

Rolf Rabenseifner

University of Stuttgart
High-Performance Computing-Center Stuttgart (HLRS)
www.hlrs.de

Outline

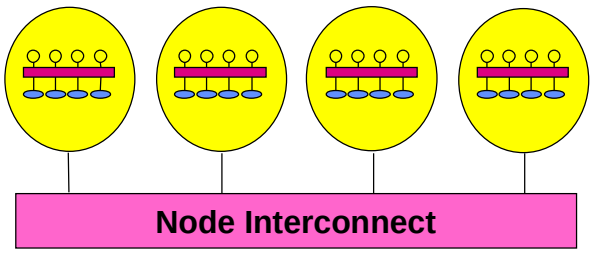
- Programming Models / an Overview
- Why should I use hybrid programming models?
- The technical question – How can I program hybrid programming models?
  - MPI-MPP
  - MPI + OpenMP *and* MPI + Automatic SMP-parallelization
  - Examples
  - Rules for MPI+OpenMP and MPI+Automatic SMP-parallelization
  - MPI+OpenMP and MPI+Compas on Hitachi SR 8000
- Which model is the best for my application?
  - Advantages and Disadvantages
- Summary

The Programming Models

- MPI
  - **standardized distributed memory parallelism with message passing**
  - **process-based**
  - **the application processes are calling MPI library-routines**
  - **compiler generates normal sequential code**
- OpenMP
  - **standardized shared memory parallelism**
  - **thread-based**
  - **mainly loop-parallelism via OpenMP-directives**
  - **compiler translates OpenMP directives into thread-handling**
- Automatic SMP-Parallelization
  - **e.g., Compas (Hitachi), Autotasking (NEC)**
  - **thread based shared memory parallelism**
  - **with directives (same programming model as with OpenMP)**
  - **supports automatic parallelization of loops that are vectorized by the compiler // (pseudo-)vectorization**
- Vectorization

Hybrid Systems

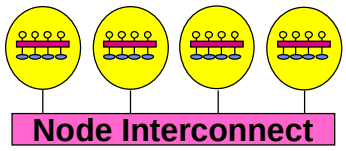
- Most modern high-performance computing (HPC) systems are clusters of SMP nodes



- DMP (distributed memory parallelization) on the node inteconnect
- SMP (symmetric multi-processing) inside of each node

Programming Models on Hybrid Systems

- MPI based:
  - the MPP model
    - **massively parallel processing**
    - **each CPU = one MPI process**
  - MPI + OpenMP
    - **each SMP node = one MPI process**
    - **MPI communication on the node interconnect**
    - **OpenMP inside of each SMP node**
    - **DMP with MPI & SMP with OpenMP**
  - MPI + automatic parallelization
    - **Compas on Hitachi, Autotasking on NEC, ...**
    - **same model as MPI+OpenMP**
- Other models:
  - HPF, MLP, ...
    - **[not part of this talk]**



Questions

<u>Why</u> should I use hybrid programming models?

<u>How</u> can I program hybrid programming models?

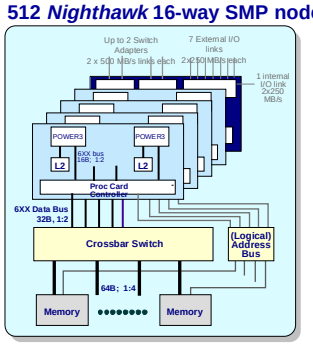
<u>Which</u> model is the best for my application?
(Advantages and Disadvantages)

Why MPI + OpenMP / Automatic SMP-parallelization ? (2)

- Programming SMP nodes only with MPI causes **additional message overhead** (instead of simply accessing the data via the shared memory)!

- i.e., use the hardware you bought!

ASCI "White":
512 *Nighthawk* 16-way SMP nodes (Power 3+)



- 4 processor cards
  - 8-way SMP
- Up to 4 Memory cards
  - Up to 16 GB, 32 GB future
- Switched Data
  - 5 (32B 1:2) X 4 (64B 1:4)
  - 3.2 GB/s per port with 200 MHz processors (B/F=2)
- SP Switch adapters directly into Node Crossbar (5th card)
- Address Rebroadcast for performance
- 200 MHz CPU
- 6.4 GF/node

MPI + OpenMP is really used!

- Projects at LLNL are combining MPI and OpenMP:

Fortran

- **TETON**
  - 2-, 3-D Radiation Transport
- **CRETIN**
  - 1-, 2-, 3-D Non-LTE Atomic Kinetics / Radiation Transport
- **LASNEX**
  - 2-D ICF Simulations
- **MCXP and MPHOT**
  - 2-, 3-D Monte Carlo Transport
- **SPPM**
  - Turbulence

C/C++

- **HYDRA**
  - 3-D Hydrodynamics
- **KULL**
  - 3-D Radiation / Hydrodynamics

Target machine: IBM Blue
3 sectors
488 nodes / sectors
4 processors / node
Now White with 16/node

Source: **Euro-Par 2000 Tutorial** © **Koniges, Keyes, Rabenseifner, Heroux**
Extreme ! Scientific Parallel Computing

Questions

Why should I use hybrid programming models?

How can I program hybrid programming models?

Which model is the best for my application?
(Advantages and Disadvantages)

How can I program these programming models (1)

- **The MPI-MPP model**
  e.g., on the Hitachi SR8000
  - Compilation + Linkage:
    - **mpif90 -OSS -noparallel -o *my_appl* *my_appl*.f or *my_appl*.f90**
    - **mpicc -O4 -pvec +Op -noparallel -o *my_appl* *my_appl*.c**
  - NQS queue and interactive partition:
    - **qsub -q multi -N #nodes jobscript**
    - **mpiexec -p multi -N #nodes -n #processes my_appl my_options**
  - 8-way SMP ==> **#processes** $\leq$ 8 * **#nodes**
  - alignment of ranks in MPI_COMM_WORLD to real CPUs:
    - **MPIR_ALIGN_NO_ROUND=yes — sequential [default at HLRS/DLR]:**
      ==> ranks 0-7 on 1st node,
      ranks 8-15 on 2nd node, ...
    - **with MPIR_ALIGN_NO_ROUND=no — round robin:**
      ==> rank 0 on node 0, rank 1 on node 1, ...
      rank (#nodes) on node 0, rank (#nodes+1) on node 1, ... □

How can I program these programming models (2)

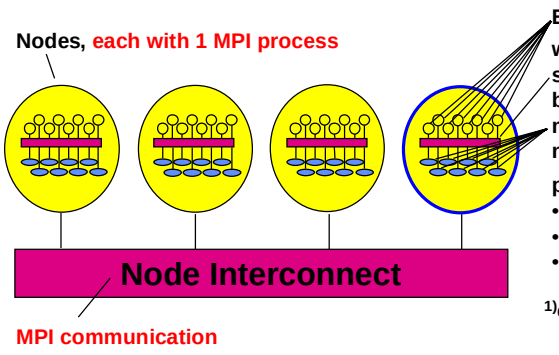
- The MPI-MPP model
- **The MPI + OpenMP model**
- **The MPI + SMP-auto-parallelization model**

Outline of the next slides:

- Programming Techniques
- Examples
- Rules for MPI+OpenMP and MPI+Automatic SMP-parallelization
- MPI+OpenMP and MPI+Compas on Hitachi SR 8000

Hybrid Programming Technique (1)

- each MPI process is now multi-threaded
- the multi-threading inside of each MPI process is done with OpenMP



Each node: 8 threads[1)] with shared memory access by all threads to memory boards on the node:

programmed with
- **(Pseudo-)vectorization,**
- **OpenMP,**
- **[or MPI (MPP-model)]**

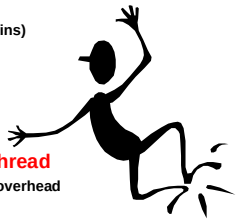
[1)]depends on the platform
SR8000: 8 processors
SX-4: up to 32
SX-5: up to 16
ccNUMA: up to 64 (ASCI: 1024)

Hybrid Programming Technique (2)

- coming from a MPI program:
  - OpenMP directives for loops, ...
  - MPI communication
    - **simplest way: outside of parallel regions**
    - **with thread-safe MPI: also inside of parallel regions — be careful!!!!**
- coming from an OpenMP/Compas program
  - MPI parallelization = same job as for sequential programs
- **Understanding memory usage is critical to correct parallelization**

But results may surprise!

- Example code - HYDRA
- Domain-decomposed hydrodynamics
  - (almost) independent mesh domains with ghost cells on boundaries
  - ghost cells communicate boundary information ~40-50 times per cycle
- Parallelism model: single level
  - MPI divides domains among compute nodes
  - OpenMP further subdivides domains among processors
  - domain size set for cache efficiency
    - **minimizes memory usage, maximizes efficiency**
    - **scales to very large problem sizes (>10^7 zones, >10^3 domains)**
- Results:
  - **MPI** (256 proc.) **~20% faster than MPI / OpenMP** (64 nodes x 4 proc./node)
  - domain-domain communication not threaded, i.e., **MPI communication is done only by main thread**
    - **accounts for ~10% speed difference, remainder in thread overhead**



The heat example — MPI+OpenMP

```
c iteration
      do it=1,itmax
         dphimax=0.
!$OMP PARALLEL PRIVATE(i,k,dphi,dphimax_priv)
         dphimax0=dphimax
!$OMP DO
         do k=ks+b1,ke-b1
          do i=is+b1,ie-b1
           dphi=(phi(i+1,k)+phi(i-1,k)-2.*phi(i,k))*dy2i
   &            +(phi(i,k+1)+phi(i,k-1)-2.*phi(i,k))*dx2i
           dphi=dphi*dt
           dphimax_priv=max(dphimax_priv,dphi)
           phi_new(i,k)=phi(i,k)+dphi
          enddo
         enddo
!$OMP END DO
!$OMP CRITICAL
         dphimax=max(dphimax,dphimax_priv)
!$OMP END CRITICAL
c        save values
!$OMP DO
         do k=ks+b1,ke-b1
          do i=is+b1,ie-b1
           phi(i,k)=phi_new(i,k)
          enddo
         enddo
!$OMP END DO
!$OMP END PARALLEL
         dphimaxpartial = dphimax
         call MPI_ALLREDUCE(dphimaxpartial, dphimax, 1,
   &      MPI_DOUBLE_PRECISION,MPI_MAX,comm,ierror)
         if(dphimax.lt.eps) goto 10
c send and receive to/from upper/lower
         call MPI_IRECV(phi(is+b1,ks),1,horizontal_border,
   &     ...          lower,MPI_ANY_TAG,comm, req(1),ierror)
c send and receive to/from left/right
         ...
         call MPI_WAITALL(4, req, statuses, ierror)
      enddo
10    continue
```

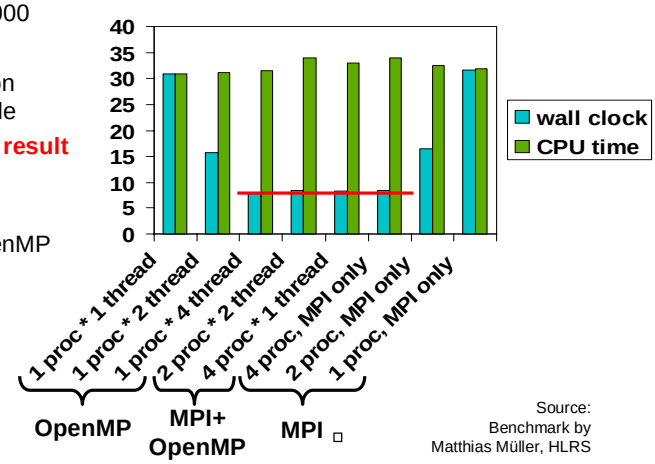

The heat example — MPI+Compas (Hitachi)

```
c iteration
      do it=1,itmax
         dphimax=0.
         dphimax0=dphimax
         do k=ks+b1,ke-b1
          do i=is+b1,ie-b1
           dphi=(phi(i+1,k)+phi(i-1,k)-2.*phi(i,k))*dy2i
   &            +(phi(i,k+1)+phi(i,k-1)-2.*phi(i,k))*dx2i
           dphi=dphi*dt
           dphimax =max(dphimax ,dphi)
           phi_new(i,k)=phi(i,k)+dphi
          enddo
         enddo
c        save values
         do k=ks+b1,ke-b1
          do i=is+b1,ie-b1
           phi(i,k)=phi_new(i,k)
          enddo
         enddo
         dphimaxpartial = dphimax
         call MPI_ALLREDUCE(dphimaxpartial, dphimax, 1,
   &      MPI_DOUBLE_PRECISION,MPI_MAX,comm,ierror)
         if(dphimax.lt.eps) goto 10
c send and receive to/from upper/lower
         call MPI_IRECV(phi(is+b1,ks),1,horizontal_border,
   &     ...          lower,MPI_ANY_TAG,comm, req(1),ierror)
c send and receive to/from left/right
         ...
         call MPI_WAITALL(4, req, statuses, ierror)
      enddo
10    continue
```

Compas: The compiler tries to make the work for you !

Benchmarking example

- heat 2000x2000
- on HP V2250
- comparison on one SMP node
- **nearly same result** with
  - OpenMP
  - MPI+OpenMP
  - MPI only



Source: Benchmark by Matthias Müller, HLRS

MPI rules with OpenMP / Automatic SMP-parallelization (1)

- Is the MPI-library thread-safe?
- **Problems:**
  - **MPI may use internally *own* threads!**
  - **MPI calls library routines (e.g., malloc) and these may be not thread-safe**
- Where can I call MPI-routines inside of an OpenMP program?
- Where can I use OpenMP parallel regions inside of an MPI-program?

- Who gives the answers?
  - MPI_Init_thread(.... ***provided***)
  - vendor documentation
  - asking the developers of MPI and OpenMP

MPI rules with OpenMP / Automatic SMP-parallelization (2)

- Special MPI-2 Init for multi-threaded MPI processes:

```
int MPI_Init_threads(int * argc, char *((*argv)[ ]), int required, int* provided)
MPI_INIT_THREAD(REQUIRED, PROVIDED, IERROR )
```

- REQUIRED values (increasing order):
  - **MPI_THREAD_SINGLE:** **Only one thread will execute**
  - **THREAD_SIMPLE: (virtual value, not part of the standard)** **MPI processes may be multi-threaded, but only master thread will make MPI-calls AND only while other threads are sleeping**
  - **MPI_THREAD_FUNNELED:** **Only master thread will make MPI-calls**
  - **MPI_THREAD_SERIALIZED:** **Multiple threads may make MPI-calls, but only one at a time**
  - **MPI_THREAD_MULTIPLE:** **Multiple threads may call MPI, with no restrictions**
- returned ***PROVIDED*** may be less than REQUIRED by the application

MPI rules with OpenMP / Automatic SMP-parallelization (3)

- ***PROVIDED*** may depend on external MPI options / environment values / which MPI library is used
- **MPI_Init(...)** may be equivalent to
  - MPI_INIT_THREAD(MPI_THREAD_SINGLE,...), or
    - **usually on MPP systems**
  - MPI_INIT_THREAD(**MPI_THREAD_FUNNELED**,...)
    - **usually on hybrid systems**
    - **only main thread (=master thread) can make MPI calls**
    - **but options may be required**
      - e.g., on NEC SX-5, MPI/SX 10.1: the environment variable MULTITASKMIX must be set to ON

==> On most hybrid systems, SMP multi-threading (autotasking, OpenMP, ...) can be used without modifying the MPI calls and structure!

Programming method / „*provided*“

Hybrid MPI/OpenMP programming method allowed with:	MPI_THREAD_SINGLE	THREAD_SIMPLE	MPI_THREAD_FUNNELED	MPI_THREAD_SERIALIZED	MPI_THREAD_MULTIPLE
No hybrid programming	+	+	+	+	+
(H1) Calling MPI only outside of parallel regions		+	+	+	+
(H2) Calling MPI also inside of OMP MASTER			+	+	+
(H3) Calling MPI also inside of OMP SINGLE				+	+
(H4) Calling MPI everywhere					+

[1)] **MPI_Init_threads() can only return MPI_THREAD_SINGLE**

Rules with MPI+OpenMP / Compas on Hitachi SR8000

- MPI_Init_threads returns always MPI_THREAD_SINGLE !
- With MPI+Compas, the compiler has to guarantee, that there are no problems!
- Therefore also no problems with THREAD_SIMPLE and OpenMP!
- No problem if you use old MPI_Init(...)

(H1) Calling MPI only outside of parallel regions

- **The simplest MPI+OpenMP model**
- Requires only the „*non-standard value*“ THREAD_SIMPLE
- MPI routines executed only by one thread
- All other threads are sleeping
- With dedicated processors: sleeping = idle time
  - ➔ all MPI communication time must be weighted by number of threads per node
  - ➔ e.g. on SR8000 (8 processors per node):
    - **Node-to-node bandwidth reduced from 1 GB/s to 125 MB/s**
    - **Latency increases from 10 µs to 80 µs**

(H1) Calling MPI only outside of parallel regions – example

(remember: implicit barrier at the end of each parallel region)

```
!$OMP PARALLEL DO
    do i=1,10000
        a(i)=b(i)+f*d(i)
    enddo
!$OMP END PARALLEL DO
    call MPI_Xxx(...)
!$OMP PARALLEL DO
    do i=1,10000
        x(i)=a(i)+f*b(i)
    enddo
!$OMP END PARALLEL DO
```

```
#pragma omp parallel for
        for (i=0; i<10000; i++)
        {   a[i]=b[i]+f*d[i];
        }
/* end omp parallel for */
        MPI_Xxx(...);
#pragma omp parallel for
        for (i=0; i<10000; i++)
        {   x[i]=a[i]+f*b[i];
        }
/* end omp parallel for */
```

(H2) Calling MPI also inside of OMP MASTER

- Inside of a parallel region, with "**OMP MASTER**"
- Requires MPI_THREAD_FUNNELED,
  i.e., only master thread will make MPI-calls
- **Caution:** There isn't any synchronization with "OMP MASTER"! Therefore, "**OMP BARRIER**" normally necessary to guarantee, that data or buffer space from/for other threads is available before/after the MPI call!

```
!$OMP BARRIER
!$OMP MASTER
        call MPI_Xxx(...)
!$OMP END MASTER
!$OMP BARRIER
```

```
#pragma omp barrier
#pragma omp master
        MPI_Xxx(...);

#pragma omp barrier
```

- But this implies that all other threads are sleeping!
- The additional barrier implies also the necessary cache flush!

(H2) … the barrier is necessary – example with MPI_Recv

```
!$OMP PARALLEL
!$OMP DO
        do i=1,1000
           a(i) = buf(i)
        end do
!$OMP END DO NOWAIT
!$OMP BARRIER
!$OMP MASTER
        call MPI_RECV(buf,...)
!$OMP END MASTER
!$OMP BARRIER
!$OMP DO
        do i=1,1000
           c(i) = buf(i)
        end do
!$OMP END DO NOWAIT
!$OMP END PARALLEL
```

```
#pragma parallel
{
#pragma for nowait
      for (i=0; i<1000; i++)
            a[i] = buf[i];

#pragma omp barrier
#pragma omp master
            MPI_Recv(buf,...);
#pragma omp barrier

#pragma for nowait
      for (i=0; i<1000; i++)
            c[i] = buf[i];

}
#pragma end parallel
```

(H2) … the barrier is necessary – example with MPI_Send

```
!$OMP PARALLEL
!$OMP DO
        do i=1,1000
           buf(i) = a(i)
        end do
!$OMP END DO NOWAIT  *)
!$OMP BARRIER        *)
!$OMP MASTER
        call MPI_SEND(buf,...)
!$OMP END MASTER
!$OMP BARRIER
!$OMP DO
        do i=1,1000
           buf(i) = c(i)
        end do
!$OMP END DO NOWAIT
!$OMP END PARALLEL
```

```
#pragma omp parallel
{
#pragma omp for nowait  *)
      for (i=0; i<1000; i++)
           buf[i] = a[i];
#pragma omp barrier      *)
#pragma omp master
           MPI_Send(buf,...);
#pragma omp barrier
#pragma omp for nowait
for (i=0; i<1000; i++)
      buf[i] = c[i];
}
#pragma omp end parallel
```

*) NOWAIT & BARRIER
or normal waiting OMP do/for

(H3) Calling MPI also inside of OMP SINGLE

- Inside of a parallel region, with "**OMP SINGLE**"
- Requires MPI_THREAD_SERIALIZED,
  i.e., any thread my call MPI routines,
  but only one at a time in each MPI process
- **Caution:** – "OMP SINGLE" synchronizes only at the end!
  Therefore, "**OMP BARRIER**" normally necessary
  before "OMP SINGLE"!
  – MPI_THREAD_SERIALIZED may be not guaranteed
  with MPI_Init()

```
!$OMP BARRIER
!$OMP SINGLE
        call MPI_Xxx(...)
!$OMP END SINGLE
```

```
#pragma omp barrier
#pragma omp single
      MPI_Xxx(...);
```

- But this implies that all other threads are sleeping!
- The additional barrier implies also the necessary cache flush! □

MPI + OpenMP on Hitachi SR8000

- MPI_Init_thread returns always provided=MPI_THREAD_SINGLE
- MPI library supports THREAD_SIMPLE and MPI_THREAD_FUNNELED
  - MPI can be called only by the master thread, or
  - outside of parallel regions
- Compilation + Linkage:
  - mpif90 -OSS **-parallel -omp** -o *my_appl* *my_appl*.f or *my_appl*.f90
  - mpicc -O4 -pvec +Op **-parallel -omp** -o *my_appl* *my_appl*.c
- NQS queue and interactive partition:
  - qsub -q multi -N #nodes jobscript
  - mpiexec -p multi -N #nodes -n #nodes my_appl my_options
- OpenMP has always 8 threads per MPI process
- OpenMP without nested parallelism
- It works!
- But, MPI+Compas may be faster than MPI+OpenMP

Batchjob-Script Recommendation on Hitachi SR8000

```
#!/bin/ksh
#@$-eo          # std-error into file std-out
#@$-N  2        # number of nodes
#@$-lE  3:20:00 # set a max. per-request etime
                # limit of 3h 20min
#@$-lT  6:40:00  # set a max. per-request cpu
                # limit of 6h 40min
#@$-lM  4000mb  # memory limit per request:
                # maximum: 6500MBytes in "multi"
                # in single recommended: < 1500 MB
#@$-ls   10mb   # per-process stack-segment size limits
#@$-q multi     # Queue request to pipe queue: multi
                # Available Pipe Queues:
                #   multi : for multi-nodes jobs
                #   single: for single-node parallel jobs
                #   scalar: for serial jobs (only one CPU)
#
#@$             # no more NQS parameters
. /etc/profile  # setup shell environment
. ~/.profile    # user specific
DEFPART="$QSUB_PARTNAME"  # set partition for ...
export DEFPART            # ... parallel jobs
```

continued

```
NODES=1                          # set number of Nodes
if [ "$QSUB_NODE" -gt 0 ]; then
  NODES="$QSUB_NODE"
fi
export NODES
(( PROCESSES=$NODES * 1 ))       # set number of processes
                                 # =nodes for MPI-COMPAS jobs
                                 #      and for MPI-OpenMP jobs
# or
#(( PROCESSES=$NODES * 8 ))      # set number of processes
                                 # =nodes*8cpu's  for MPI jobs
export PROCESSES
cd $SCRDIR/work_dir              # go into working directory
date
echo  start application on Nodes: $NODES in partition: $DEFPART
                                 # start application using all nodes
                                 # (mpirun doesn't search $PATH)
mpiexec -N $NODES -n $PROCESSES $SCRDIR/work_dir/my_prog \
                                                  my_options
# or old-fashion mpirun:
#mpirun -n $NODES -np $PROCESSES $SCRDIR/work_dir/my_prog \
                                                  my_options
date
```

Questions

Why should I use hybrid programming models?

How can I program hybrid programming models?

Which model is the best for my application?
(Advantages and Disadvantages)

Which model is the best?

MPI–MPP
... use the cluster only like an MPP?

MPI + OpenMP
... sounds optimal, but really?

MPI + automatic SMP parallelization (e.g., Compas)
... like heaven on earth?

Advantages (+) and Disadvantages (–), I.

Hybrid programming

- MPI+OpenMP or MPI + automatic SMP-parallelization (e.g., Compas)
  - + **no communication inside of a node**
  - – **Amdahl's law, e.g., on 8-way-SMP :**
    - **sequential part = 2 % ==> efficiency = 88 %only**
    - **= 1 % ==> = 93 %**

    ==> more than 98 % or 99 % should be suitable for SMP-parallelization!!!
  - – **the MPI routines are executed only on the master thread, therefore:**
  - – **CPU-intensive MPI routines (MPI_Reduce, MPI_Allreduce) and memory-intensive MPI routines (concatenation of strided derived datatypes) may be slow**
  - + **less number of nodes may change the convergence rate of the mathematical algorithm**

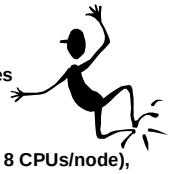
Advantages (+) and Disadvantages (–), II.

Hybrid programming (continued)

- MPI + OpenMP with THREAD_SIMPLE
  - + **easy to program, does not imply a load balance problem**
  - – **other threads are sleeping while master thread calls MPI routines**
  - – **node-to-node communication time therefore weighted by number of processors/node !!!**

    ==> node-to-node bandwidth = 1 GB/s reduces to 125 MB/s (on 8 CPUs/node),

    latency = 20 µs explodes to 160 µs



- MPI + OpenMP with THREAD_FUNNELED,

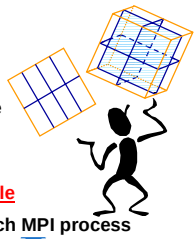
  i.e. MPI is called by the master thread and other threads are computing in parallel
  - + **node-to-node communication time counts single**
  - – **but load balance is necessary !!!**
  - + **useful for dynamic task distribution programming**
  - – **problematic for domain decomposition programming**

Advantages (+) and Disadvantages (–), III.

Hybrid programming (continued)

- MPI + automatic SMP-parallelization (e.g., Compas)
  - **+ Compas generates parallel regions automatically**
  - **+ Compiler has to guarantee correctness**
  - **– Compas directives (if necessary) are not standardized**
  - **– Same MPI performance problems as with THREAD_SINGLE: node-to-node communication time therefore weighted by number of processors/node !!!**

The MPP model: MPI processes on each CPU



- **+ no SMP parallelization necessary**
- **– additional communication inside of each node**
- **– not easy to align the process neighbors on the same SMP node**
- **+ with optimal process alignment, inner node communication comparable to node-to-node communication**

  ==> node-to-node communication time counts only about double
- **– bad convergence, if multigrid implementation only inside of each MPI process**

Advantages and Disadvantages, Summary

- Hybrid programming
  - may reduce the execution time in comparison with MPI-MPP programming
  - in most cases only**, if the MPI communication is done while the other threads are computing** (i.e. they are not sleeping)
- This overlap of computation and MPI communication:
  - requires sophisticated programming
  - risk of race conditions
    (i.e. the shared read+write access of two threads is not synchronized)
  - risk of bad load balance

Summary

- Why MPI + OpenMP/Compas?
  - **may be a good model for hybrid hardware — cluster of SMP**
- Programming Techniques
  - **each MPI process is now multi-threaded**
  - **the multi-threading inside of each MPI process is done with OpenMP automatic SMP parallelization, e.g., Compas on Hitachi**
- Examples
  - **there is no optimal programming model**
- Rules for MPI + OpenMP
  - **Is the MPI-library thread-safe?**
  - **MPI+OpenMP: – Where can I call MPI-routines?**
    – Where can I use parallel regions?
- Which programming model is the best?
  - **depends on your application needs for communication, and**
  - **capability for SMP parallelization, and**
  - **your available working hours for hybrid programming!**