Parallel Computing

Introduction to Parallel Computing

Distributed memory systems and programming

Jesper Larsson Träff

Technical University of Vienna

Parallel Computing

©Jesper Larsson Träff

TU WIEN

Tools for MPI

- Evaluating MPI+system performance
- Monitoring execution of MPI applications (post-mortem)
- Writing own tools

©Jesper Larsson Träff

Evaluating MPI+system performance

Benchmarking of individual MPI functions gives information on how a particular MPI library implementation performs on a particular system

Benchmarking is done by special benchmarking program, employing (hopefully) well-defined, explicit benchmarking and reporting principles. Exercise GREAT CARE!!

Some requirements/expectations:

- Accurate
- Reproducible
- Realistic
- Reasonably fast…

©Jesper Larsson Träff

MPI support for benchmarking

- MPI_Wtime to time indidual (or sequence) of MPI calls
- MPI_Barrier (careful!) or special algorithm to synchronize clocks

Repeat measurements, eliminate outliers, apply statistics, select measurement points carefully (e.g. not only powers of two message sizes)

Some well-known MPI benchmarks:

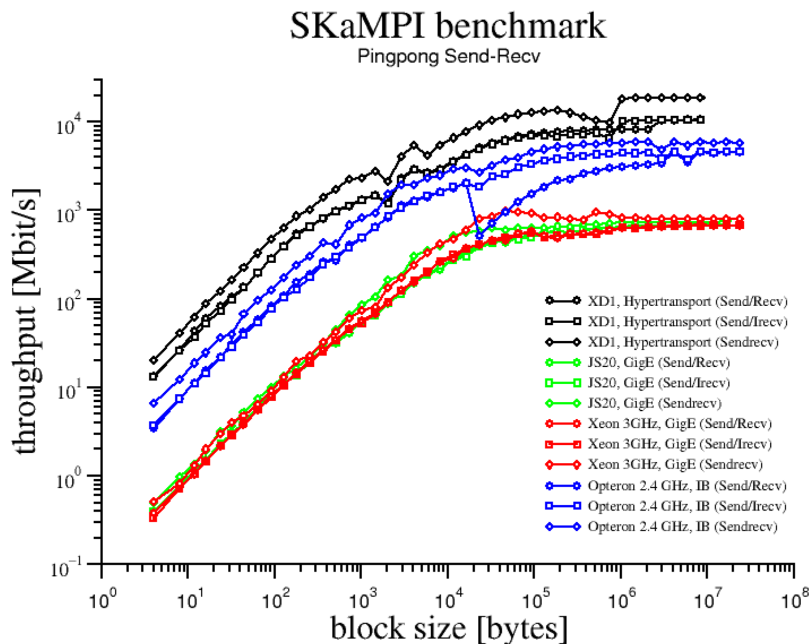
- mpptest – very basic benchmarking, delivered with MPICH
- PMB (Pallas MPI Benchmark), now IMB (Intel …) – was unsound
- MPIBench
- SKaMPI
- …

But:
No universally accepted benchmark with broad coverage of MPI functions

©Jesper Larsson Träff

Some examples with SKaMPI

Performance: communication time or bandwidth as function of message size for fixed number of processes



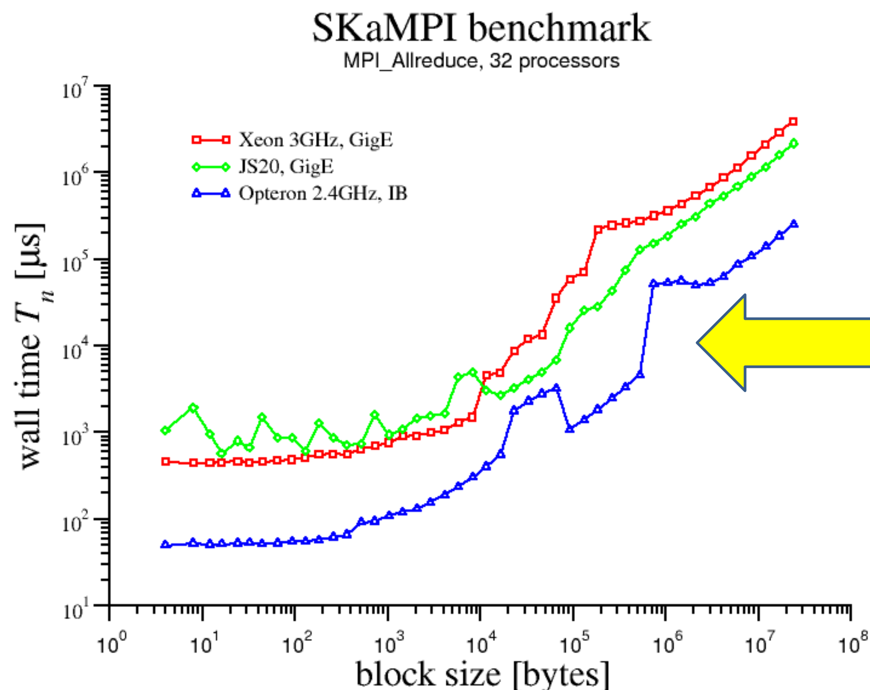
Rank 0	Rank 1
beg = MPI_Wtime()	
MPI_Send()	MPI_Recv()
MPI_Recv()	MPI_Send()
end = MPI_Wtime()	

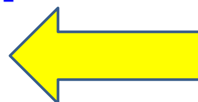
Ping-pong test

©Jesper Larsson Träff

Some examples with SKaMPI

Performance: communication time or bandwidth as function of message size for fixed number of processes





Algorithm change?

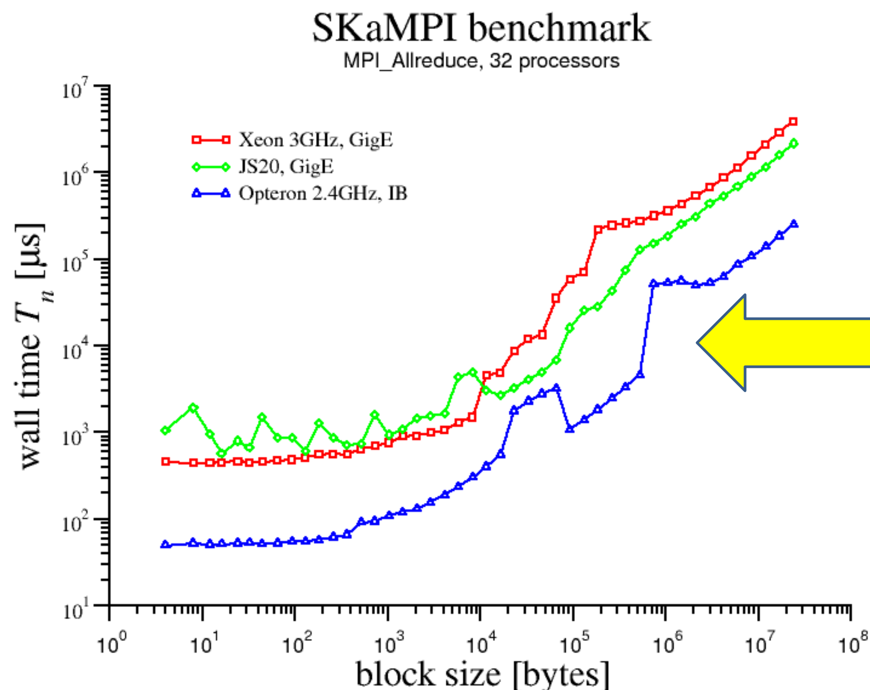
This behavior is not what would be expected; not good for performance portability

©Jesper Larsson Träff

Parallel Computing

Some examples with SKaMPI

Performance: communication time or bandwidth as function of message size for fixed number of processes

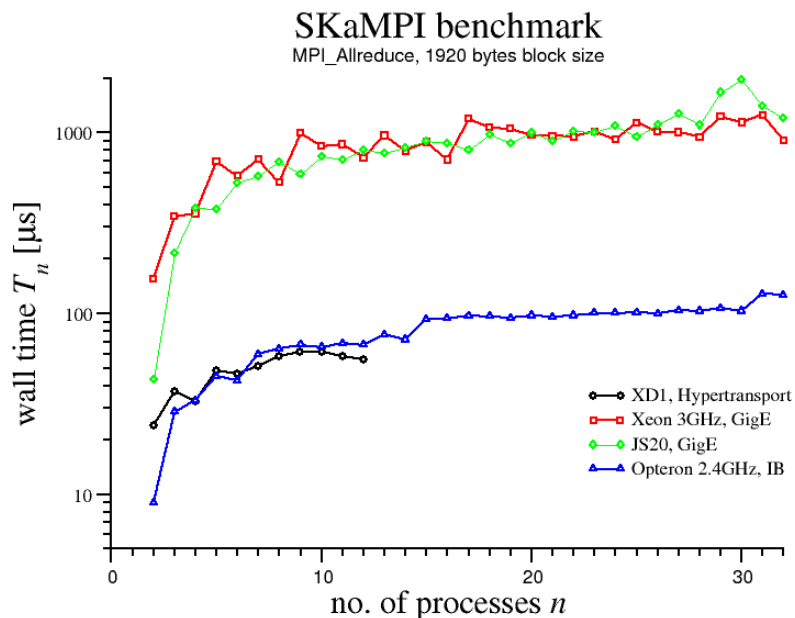


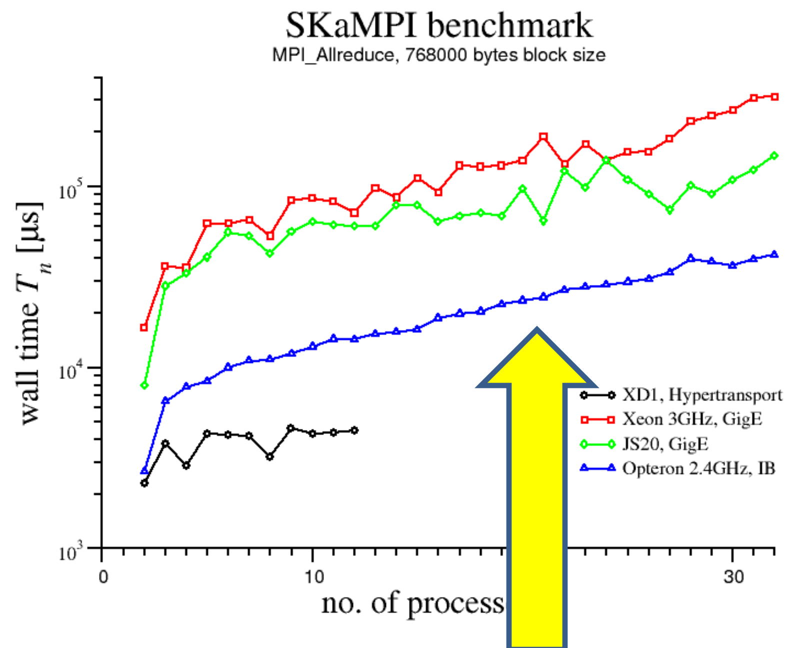
Algorithm change?

Application programmer may be tempted to program around performance anomaly

TU WIEN

Scalability: communication time/bandwidth for fixed message size for varying number of processes

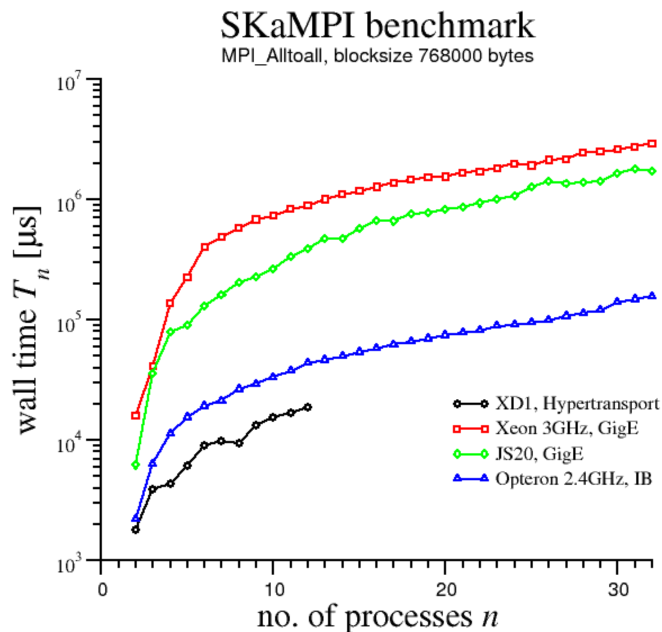


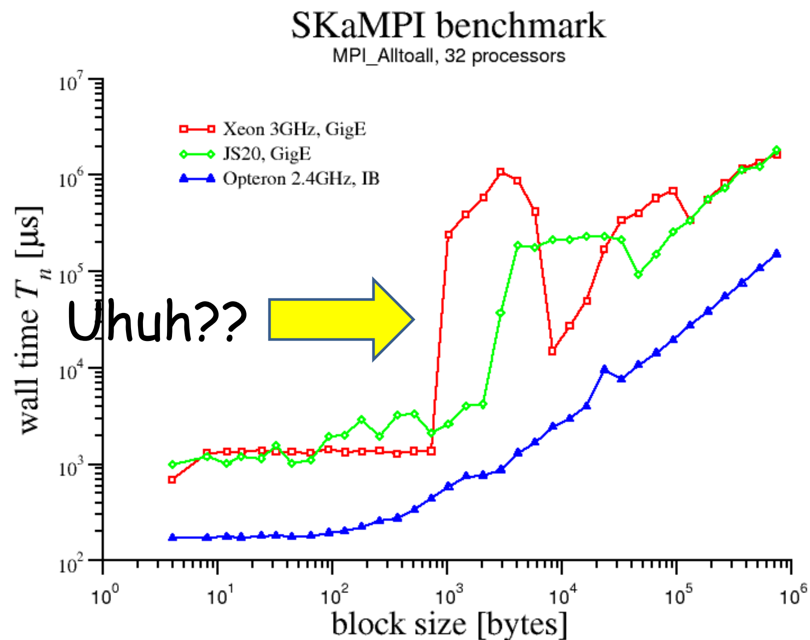


MPI_Allreduce should be O(m)+o(n) for large data m

©Jesper Larsson Träff

Performance and scalability, MPI_Alltoall





Uhuh??

©Jesper Larsson Träff

Monitoring execution of MPI applications

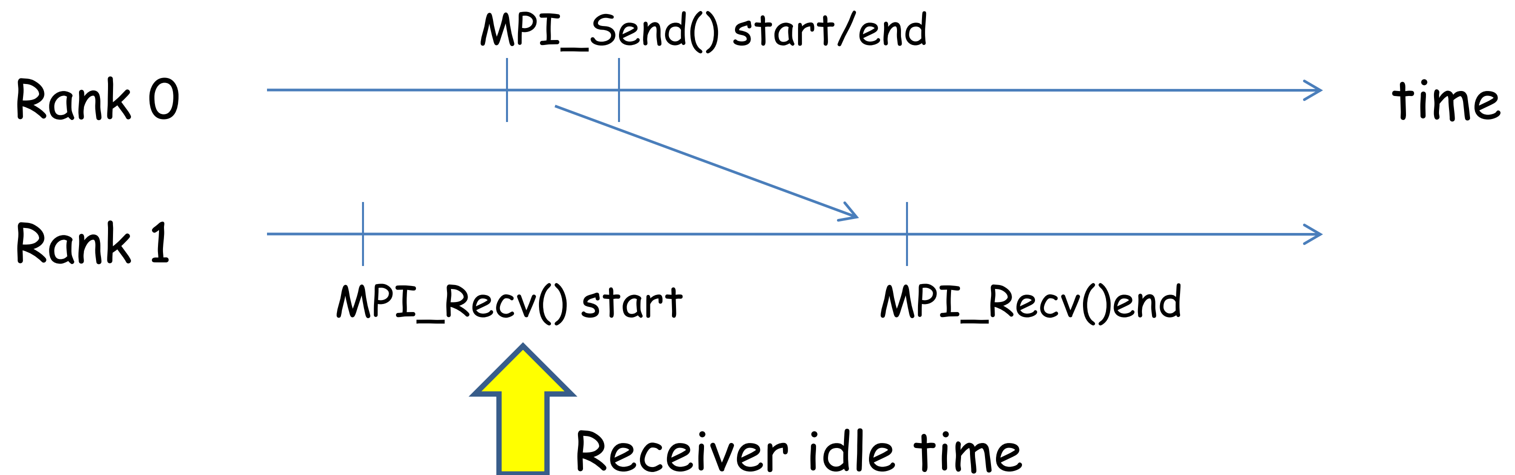
Basic:
how much time does the application spend in (different types of) MPI communication?

Purpose: to identify inefficiencies and load imbalances, and socalled „bottlenecks“: points where some processes wait unnecessarily for other processes due to unfortunate communication patterns

Advanced tools:
(Semi-)automatic identification of bottlenecks

©Jesper Larsson Träff

Common to such tools:
-Trace (parts) of execution
-Post-mortem visualization and analysis of trace



Obvious problem:
scalability for large numbers of processes (visualization and trace files generation/storage/analysis)

©Jesper Larsson Träff

Parallel Computing

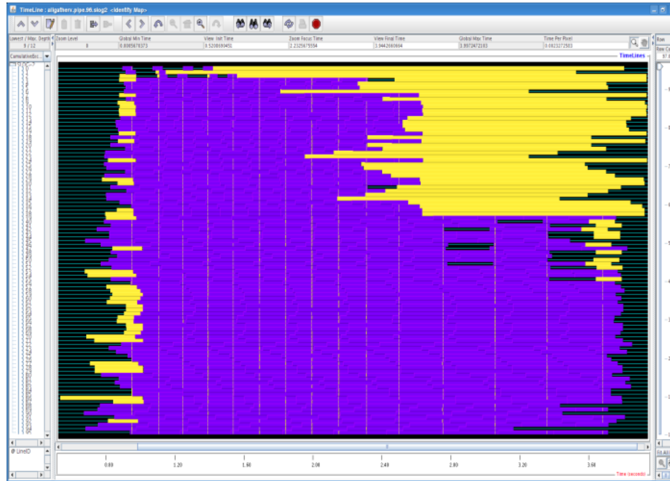
Portable tools typically developed independently of any particular MPI implementation

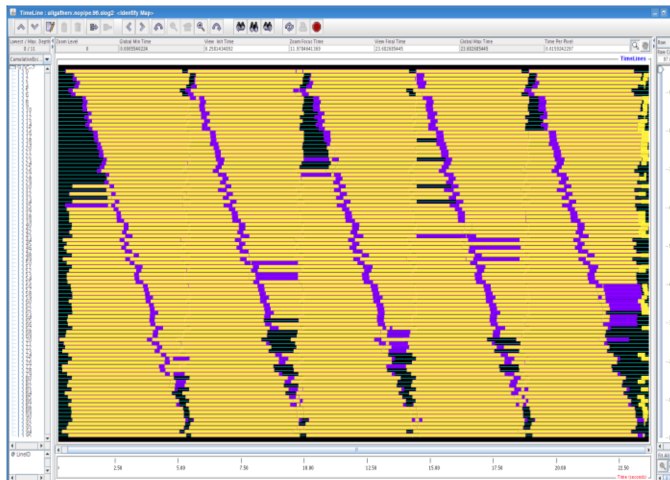
How is this possible?

Basic tools:
Nupshot/jumpshot developed with MPICH

©Jesper Larsson Träff

TU WIEN





Jumpshot example:
analysing two versions of an
MPI_Allgatherv algorithm

idle

comm

©Jesper Larsson Träff

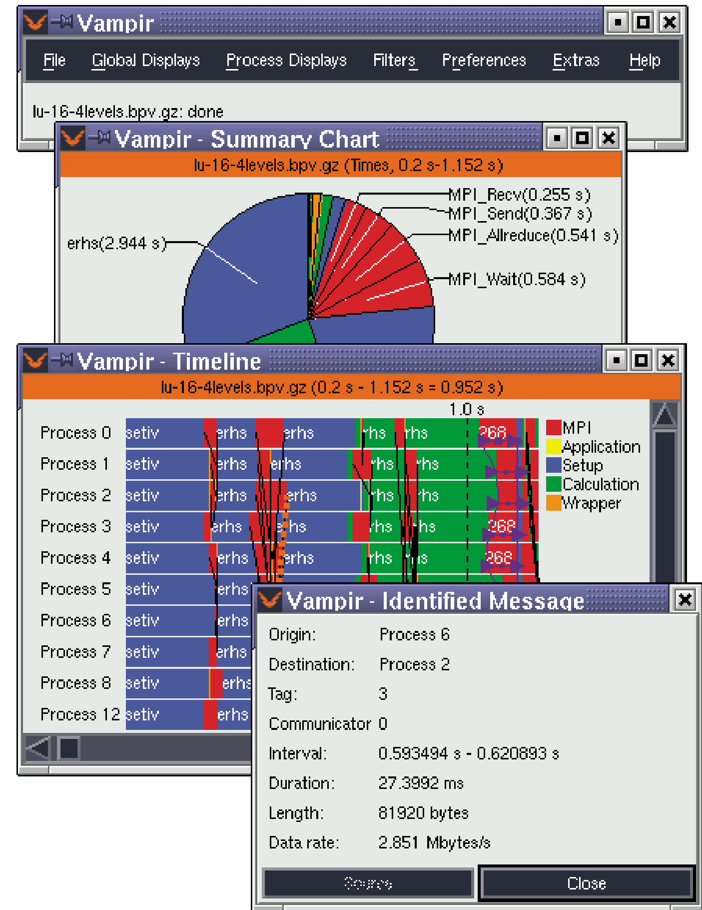
Parallel Computing

www.vampir.eu

VAMPIR (Visualization and Analysis of MPI Resources)

Advanced visualization, lots of summary information can be generated and from traces.

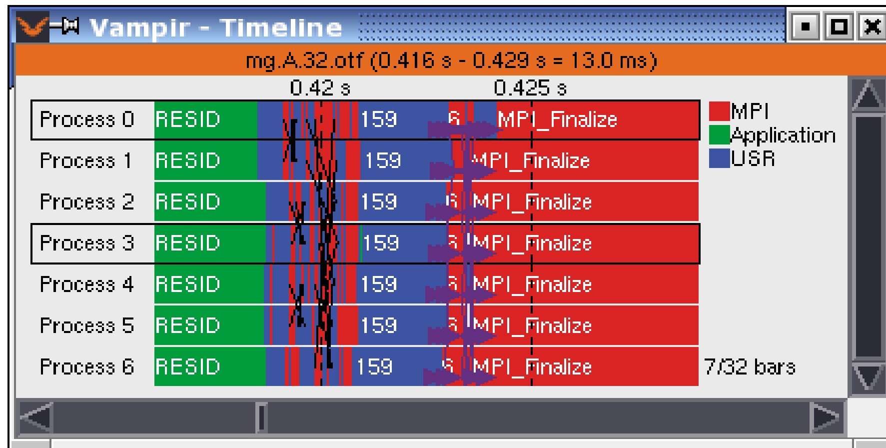
Now part of Intel tool suite



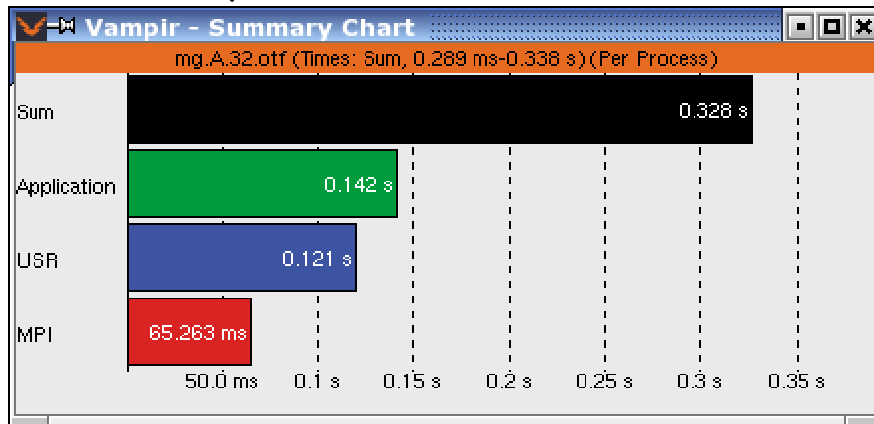
©Jesper Larsson Träff

TU WIEN

Timeline – communication behavior/process



Summary



©Jesper Larsson Träff

Parallel Computing

Advanced performance tools:
- Kojak, Scalasca, Periscope
- …

Even more advanced tools:
Suggestions for eliminating identified bottlenecks: research topics(!)

Performance tool development an active research area within MPI community, active groups e.g. in Germany
- Efficient, parallel analysis of very large trace files
- Handling of large trace files, compression,
- …

©Jesper Larsson Träff

MPI correctness tools

MPI implementations normally do only local, and very limited usage checking.

Wrong use of MPI functions can have disastrous effects:
- Application crashes
- Application deadlocks
- Application produces wrong results
- …

directly at wrong call, or – worse! – later: trace/debug gives no immediate insight where the problem was

©Jesper Larsson Träff

Example:

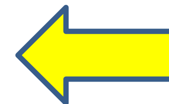
Root:

```
arguments[0] = SIZE1; arguments[1] = SIZE2;
MPI_Bcast(arguments,2,MPI_INT,root,comm);

MPI_Scatter(data,SIZE1,…);
MPI_Scatter(data,SIZE2,…);
```

Non-root:

```
MPI_Bcast(arguments,1,MPI_INT,root,comm);
MPI_Scatter(…,data,arguments[0],…);
MPI_Scatter(…,data,arguments[1],…);
```



Big problem at non-root

Parallel Computing

Run-time checking of correct use of MPI functions:
- Allocating and freeing MPI objects
- Pending operations correctly closed
- Correct semantics of e.g. collective operations
- …

Some correctness tools:
- Marmot
- IMC (Intel Message Checker)
- MPI-CHECK (rudimentary tool for FORTRAN)
- NEC collectives verification interface
- ANL collectives verification interface
- …

TU WIEN

Parallel Computing

MPI debugging

Hard-core: fprintf + pdbx …

MPI specific tools, giving access to relevant parts of MPI internals require non-standard „hooks", and are therefore not always portable

MPI 3.0 may address this point (other project: MPI Peruse)

Example: totalview – supported by many MPICH derived MPI library implementations

©Jesper Larsson Träff

Parallel Computing

Writing own tools: the MPI standard Profiling interface

Idea: for each MPI function, there is a PMPI function with same functionality. This makes it posible to any replace MPI function with own function that does additional „profiling“

Example: profiling for MPI_Send

```
MPI_Send(buffer,count,datatype,…)
{
  <tool specific action>
  PMPI_Send(buffer,count,datatype,…);
  <more tool specific action>
}
```


©Jesper Larsson Träff

TU WIEN

Parallel Computing

```
MPI_Send(buffer,count,datatype,…)
{
  <tool specific action>
  PMPI_Send(buffer,count,datatype,…);
  <more tool specific action>
}
```

Compile normally with mpicc into some libmyprof.c

Link application with

mpicc -lmyprof -lpmpi -lmpi …

©Jesper Larsson Träff

Special MPI function for controlling profiling interfaces

```
MPI_Pcontrol(int level, …);
```

Typically:
level = 0: no profiling
level = 1: default level
level = 2: verbose

MPI_Pcontrol() allowed to take additional parameters

©Jesper Larsson Träff

Parallel Computing

Example 1: collecting summary information

Amount of data communicated
-Per MPI function (or grouped: point-to-point, one-sided, collective)
-Per process/total

-Time spent in communication

©Jesper Larsson Träff

```
MPI_Init(int argc, …)
{
   prof_sent = 0; prof_sendtime = 0.0;
   prof_recv = 0; prof_recvtime = 0.0;
   <and other initialization…>
   PMPI_Init(argc,argv);
}
```

```
MPI_Send(buffer,count,datatype,…)
{
  MPI_Type_size(datatype,&typesize);
  prof_sent += count*typesize;
  start = PMPI_Wtime();
  PMPI_Send(buffer,count,datatype,…);
  prof_sendtime += (PMPI_Wtime()-start);
}
```

For each communication operation: Trace time spent and total data

©Jesper Larsson Träff

```
MPI_Finalize()
{
  if (rank==0) {
    <gather results from all processes>
    for (i=0; i<size; i++) {
      fprintf(stderr,"Proc %i:\n");
      fprintf(stderr,"Sent %d\n",proc_sent_all[i]);
      …
  } else {
    …
  }
  PMPI_Finalize();
}
```


©Jesper Larsson Träff

Parallel Computing

Example 2:simple trace

```
MPI_Bcast(buffer,…)
{
  TRACE_event(START_BCAST, …);
  PMPI_Bcast(buffer,…);
  TRACE_event(END_BCAST, …);
}
```

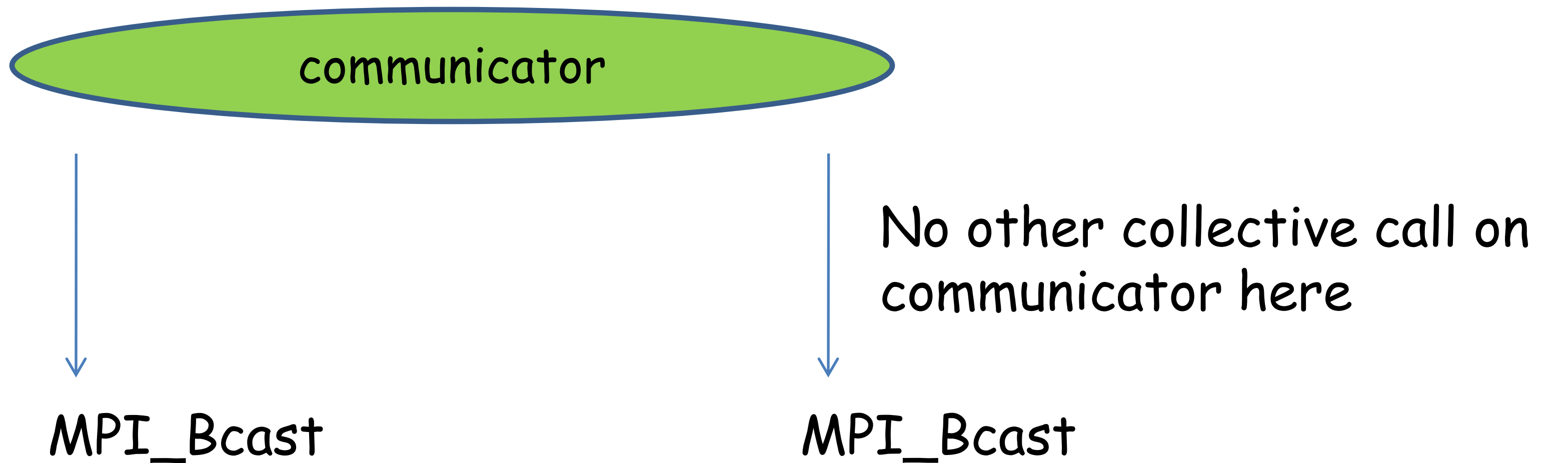

©Jesper Larsson Träff

Example 3: collective correctness tool

MPI collectives have strict, and sometimes quite complex requirements on argument consistency:

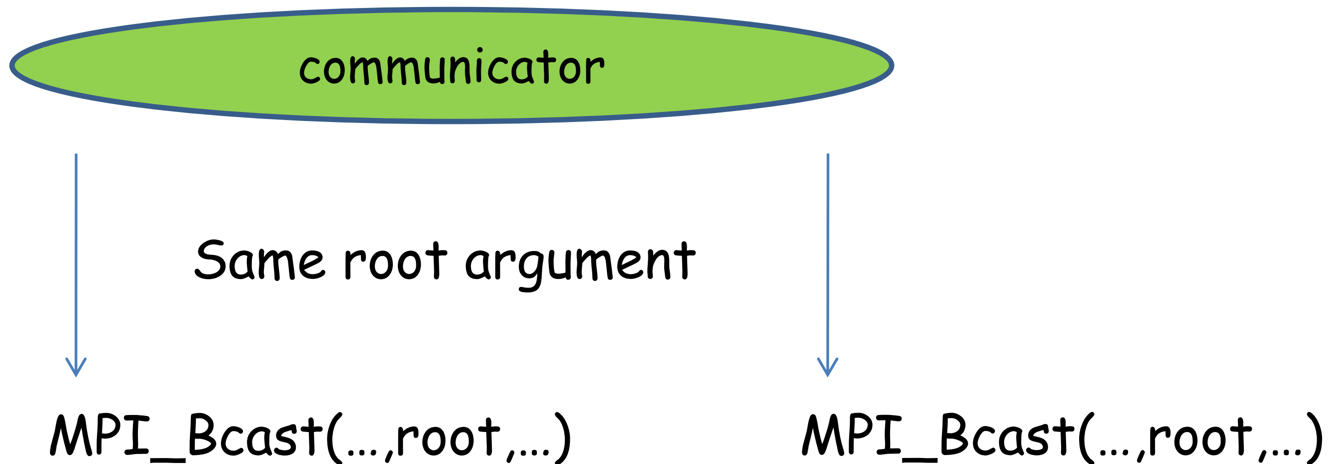
1. Collective calls must be done in same order among all processes
2. Root, operator, and similar arguments must be same for all processes
3. Data sizes must match exactly between receiving and sending processes
4. Datatype signatures must match
5. …

Idea: create „profiling“ tool to check correct use - these checks are too expensive for production code



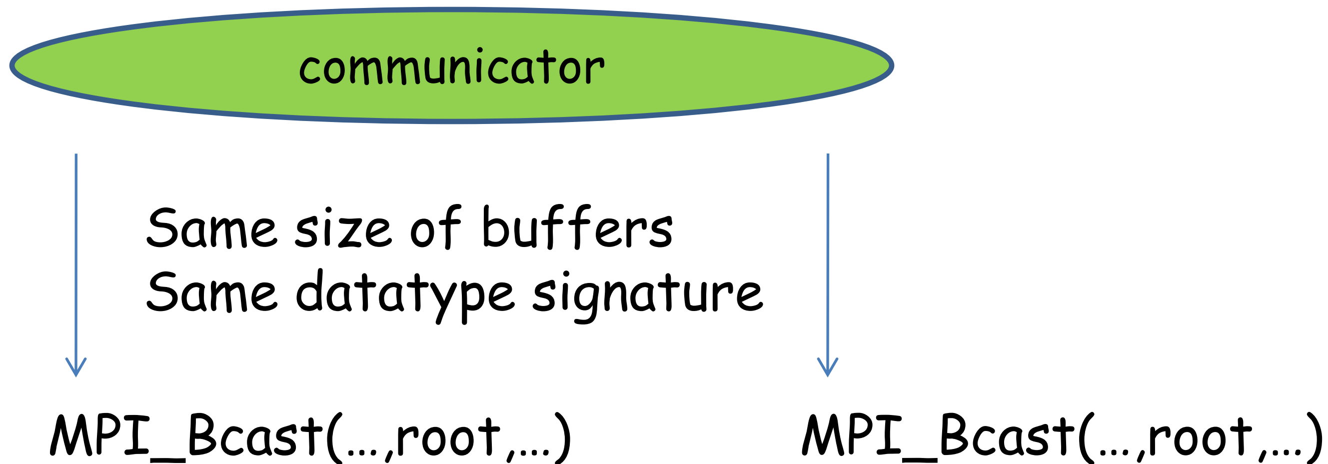

```
MPI_Bcast(buffer,count,datatype,root,comm);
{
  if (rank==0) {
    calltype = BCASTTYPE; // type of collective operation
    PMPI_Bcast(calltype,1,MPI_INT,0,comm);
  } else {
    PMPI_Bcast(calltype,1,MPI_INT,0,comm);
    if (calltype!=BCASTTYPE) {
      <ALERT: inconsistent use of collective>
    }
  }
}
```


©Jesper Larsson Träff




```
MPI_Bcast(buffer,count,datatype,root,comm);
{
  if (rank==0) {
    rootarg = root; // set root argument
    PMPI_Bcast(rootarg,1,MPI_INT,0,comm);
  } else {
    PMPI_Bcast(rootarg,1,MPI_INT,0,comm);
    if (rootarg!=root) {
      <ALERT: inconsistent root argument>
    }
  }
}
```


©Jesper Larsson Träff



Same idea, but MPI does not provide good functionality for accessing the type signature

```
MPI_Bcast(buffer,count,datatype,root,comm);
{
  <checking>
  PMPI_Bcast(buffer,count,datatype,root,comm);
  // all correct, now Bcast
}
```


©Jesper Larsson Träff

Summary

Profiling interface facility has proved useful, lots of MPI tools exist built on this functionality, easy to create own, small tools

Profiling interface drawback:
Only one PMPI function fallback, not possible to combine different profiling interfaces

Is being addressed in MPI Forum, new, better profiling functionality may come in MPI 3.0

©Jesper Larsson Träff

Benchmarking MPI functions

©Jesper Larsson Träff

Benchmarking of MPI functions

Aim:
a realistic picture of performance aspects of MPI library and system

Overall performance: use application (kernel), several application kernel. Know what they are doing

Specific aspects: performance of individual MPI functions

©Jesper Larsson Träff

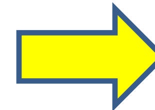
Benchmarking of MPI functions

- What is being measured?
- Reproducible (robust)
- Accurate
- Detailed, realistic picture

Basic communication properties of library and system

Mostly ignores interaction with other functions

```
MPI_Send(buf,count,…,dest,…,comm);
```



```
MPI_Recv(buf,count,…,source,…,comm,&status);
```


©Jesper Larsson Träff

•What is being measured?
•Make assumptions clear and control them:

1. Which functions?

2. Point-to-point: One pair or all pairs? Process mapping (MPI_COMM_WORLD, comm)? Which data sizes? Datatypes? Where are the data (cache effects)? Data dependent (non-oblivious)?

3. Collectives: effect of root placement, process mapping (MPI_COMM_WORLD vs. random comm). Data sizes? Datatypes? Where are the data?

4. Other MPI functionality: …

©Jesper Larsson Träff

Parallel Computing

Pitfall: benchmarking incorrect functions

Incorrectly implemented functionality can sometimes be faster, e.g. functionality that only works in special cases

Make sure the functionality is correct, separate testing!!

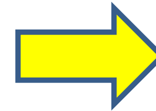
©Jesper Larsson Träff

- Reproducible (robust)

Individual measurements may vary: load on system (noise), communication system itself rarely deterministic

Repeat measurement, synchronize with barrier (MPI_barrier)

```
for (#repetitions): synchronize and time:
MPI_Send(buf,count,…,dest,…,comm);
                               MPI_Recv(buf,count,…,source,…,comm,&status);
```



©Jesper Larsson Träff

Pitfall: forgetting resolution of timer

```
stime = MPI_Wtime();
MPI_<function>
etime = MPI_Wtime();
```

Irreproducible and inaccurate results if MPI_<function> is fast compared to resolution of MPI_Wtime(). Normally MPI_Wtime() is good enough for communication functions

```
res = MPI_Wtick();
```

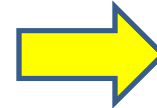

©Jesper Larsson Träff

•Reproducible (robust)

Averages are not reproducible (outliers, system disturbance, first call may be expensive)

Repeat measurement, synchronize with barrier, use mimimum

```
for (#repetitions): time:
MPI_Send(buf,count,…,dest,…,comm);
```



```
                MPI_Recv(buf,count,…,source,…,comm,&status);
```

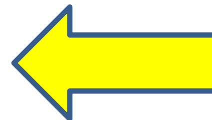
record best measured time so far

Pitfall: repeat correctly

```
for (#repetitions) {
  MPI_Barrier(comm);
  stime = MPI_Wtime();
  MPI_Bcast(...,comm);
  etime = MPI_Wtime();
}
```

can be quite different from

```
MPI_Barrier(comm);
stime = MPI_Wtime();
for (#repetitions) {
  MPI_Bcast(...,comm)
}
etime = MPI_Wtime();
```



Measures pipelining effects

©Jesper Larsson Träff

- Accurate

What is completion time?

- Point-to-point: ping-pong, roundtrip
- Collective: time for slowest process to complete

©Jesper Larsson Träff

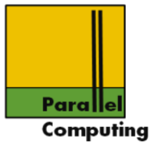
Parallel Computing

•Accurate

Performance as function of data size; measure points close enough; automatic refinement in critical regions

Pitfall: only powers of 2 (many MPI implementations have switch points and special cases at power-of-2 boundaries, performance might be exceptionally good or bad)

©Jesper Larsson Träff



0,0000000000
-1,0000000000
-2,0000000000
-3,0000000000
-4,0000000000
-5,0000000000
-6,0000000000

32 47 73 109 168 256 387 588 890 1350 2048 3102 4705 7129 10809 16384 24832 37640 57051 86475 131072 198667 301124 456416 691801 1048576 1589342

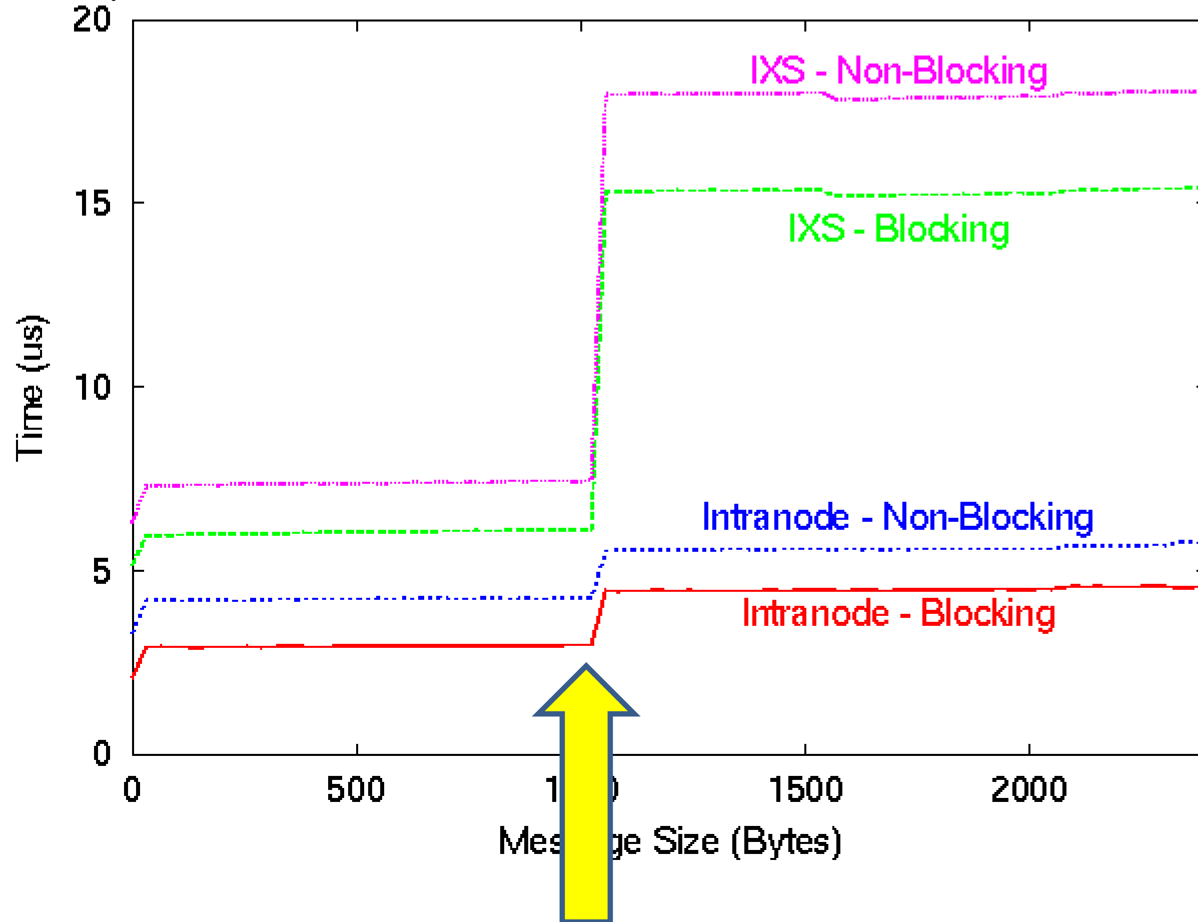
1
47
10
36

m = 2^x

Intel SCC, measurements by Markus Pichler, January 2011

©Jesper Larsson Träff

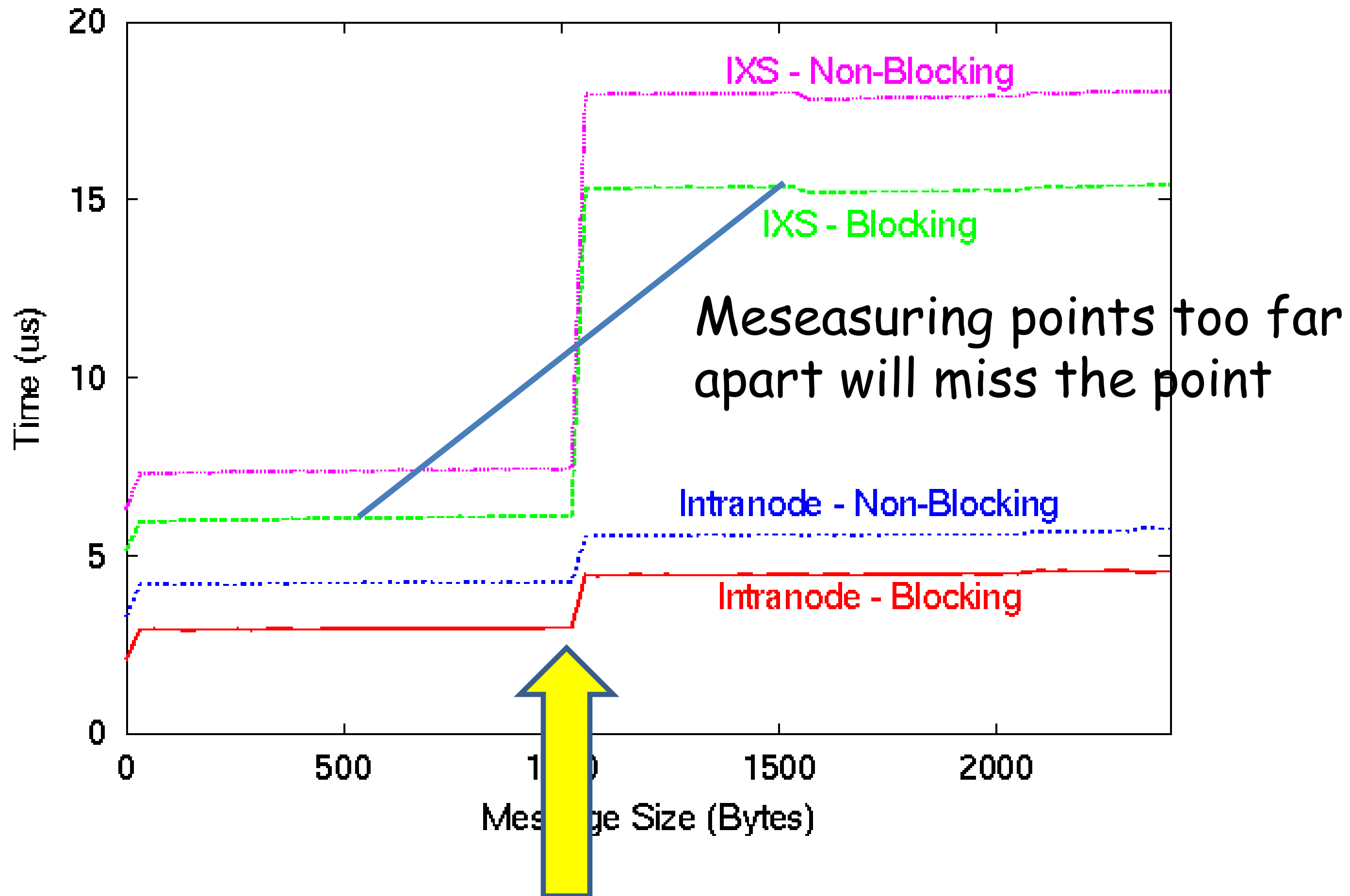
Typical point-to-point performance (mpptest of mpich2, NEC's MPI/SX)



What happens here

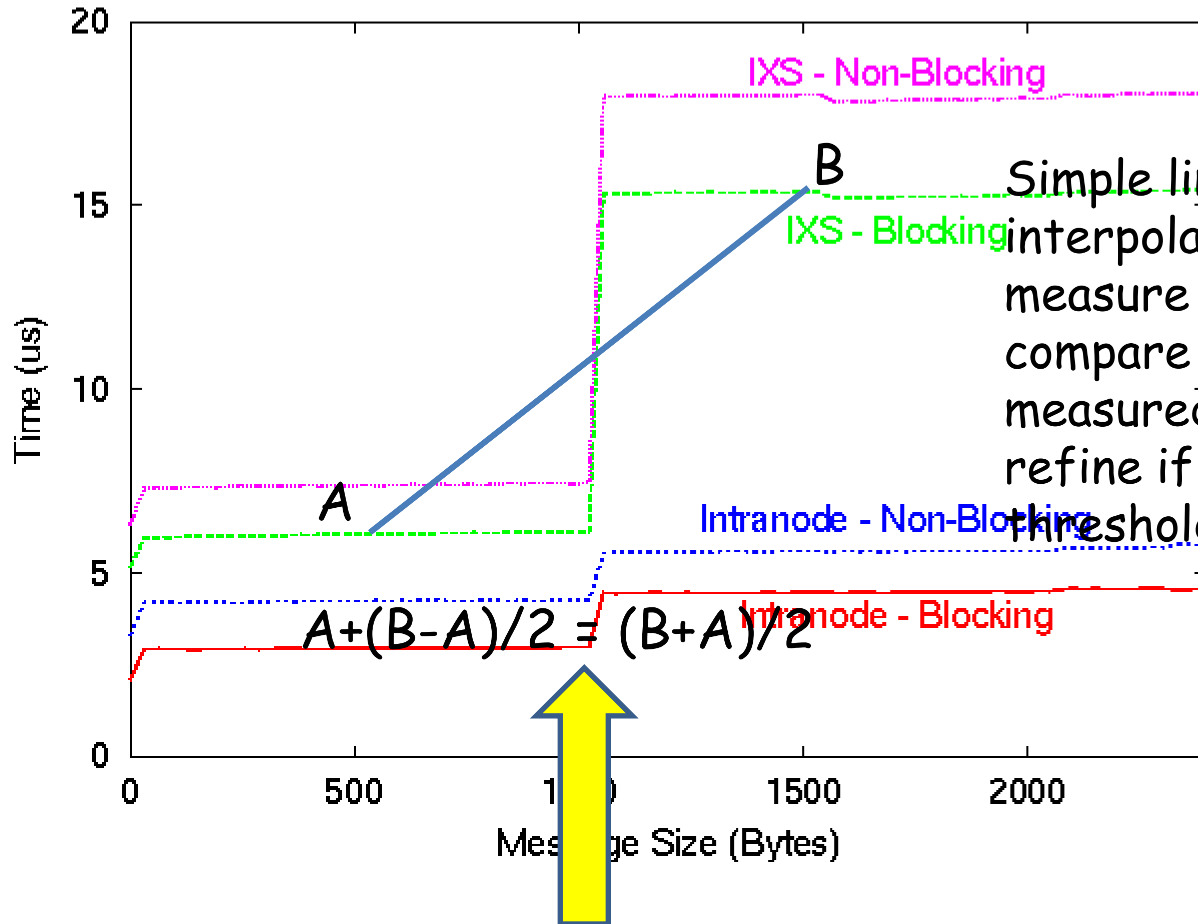
©Jesper Larsson Träff

Typical point-to-point performance (mpptest of mpich2)



©Jesper Larsson Träff

Typical point-to-point performance (mpptest of mpich2)



Simple linear interpolation refinement: measure f(A), f(B), compare (f(B)+f(A))/2 to measured f((B+A)/2), refine if beyond threshold accuracy

What happens here

©Jesper Larsson Träff

•Detailed, realistic picture

•Data in cache or not? Process mapping (communicator).
•Point-to-point bandwidth (one pair) or bisection (all pairs)

©Jesper Larsson Träff

Pitfall: using special 2-process communicator for point-to-point benchmarking

Unfair/unrealistic – has sometimes been used to demonstrate good performance…

Data structures and time for setting up and using such a communicator may be smaller than MPI_COMM_WORLD

©Jesper Larsson Träff

Cache? What is accurate, realistic?

View A:
Applications compute into buffer, then communicate, so communication might well be from cache?

View B:
Applications compute many things, communicate later, communication might as well be from memory (not cached)?

From-cache communication: touch buffer before communication

Out of memory communication: use large communication buffer, step through this cyclically (see benchmark structure)

©Jesper Larsson Träff

Parallel Computing

Good practice:

- Ensure correctness! (separate testing)
- Run through all counts once (without timing) to „prime“ system
- Use repetitions, measure minimum
- Many counts, not only powers of 2, powers of 10
- Automatic refinement (if possible)
- Provide for different communicators
- Point-to-point: provide for one pair/all pairs
- Provide for different datatypes
- Control cache-behavior if possible

Suggested benchmark structure, precomputed counts

```
for (#repetitions) {
  for (i=0; i<counts; i++) { // step through all counts
    MPI_Barrier()
    stime = MPI_Wtime()
    MPI_<function>(buffer); // in or not in cache?
    etime = MPI_Wtime();
    time[i] = etime-stime; // local time
    totaltime[i] += time[i]; // add to total
  }
  // find slowest process
  MPI_Reduce(time,counts,MPI_DOUBLE,MPI_MAX,0,comm);
  if (rank==root) {
    for (i=0; i<counts; i++) {
      if (time[i]<besttime[i]) besttime[i] = time[i];
      // best of slowest for all counts
  }
}
```

„Best possible time of last process to finish"

Parallel Computing

This gets more complicated if refinement is employed (the number of measurement points can very over the repetitions)

Statistical question:

How many repetitions needed to get a stable "best of slowest" value?

Experience: 20-100 iterations suffice

©Jesper Larsson Träff

MPI_<function> for unidirectional bisection BW measurement:

```
if (rank%2==0&&rank<size-1) {
  MPI_Send(sendbuf,c,type,rank+1,0,comm);
  MPI_Recv(recvbuf,c,type,rank+1,0,comm,
           MPI_STATUS_IGNORE);
} else if (rank%2==1) {
  MPI_Recv(recvbuf,c,type,rank-1,0,comm,
           MPI_STATUS_IGNORE);
  MPI_Send(sendbuf,c,type,rank-1,0,comm);
} else {
  MPI_Sendrecv(sendbuf,c,type,rank,0,
               recvbuf,c,type,rank,0,comm,
               MPI_STATUS_IGNORE);
}
```


©Jesper Larsson Träff

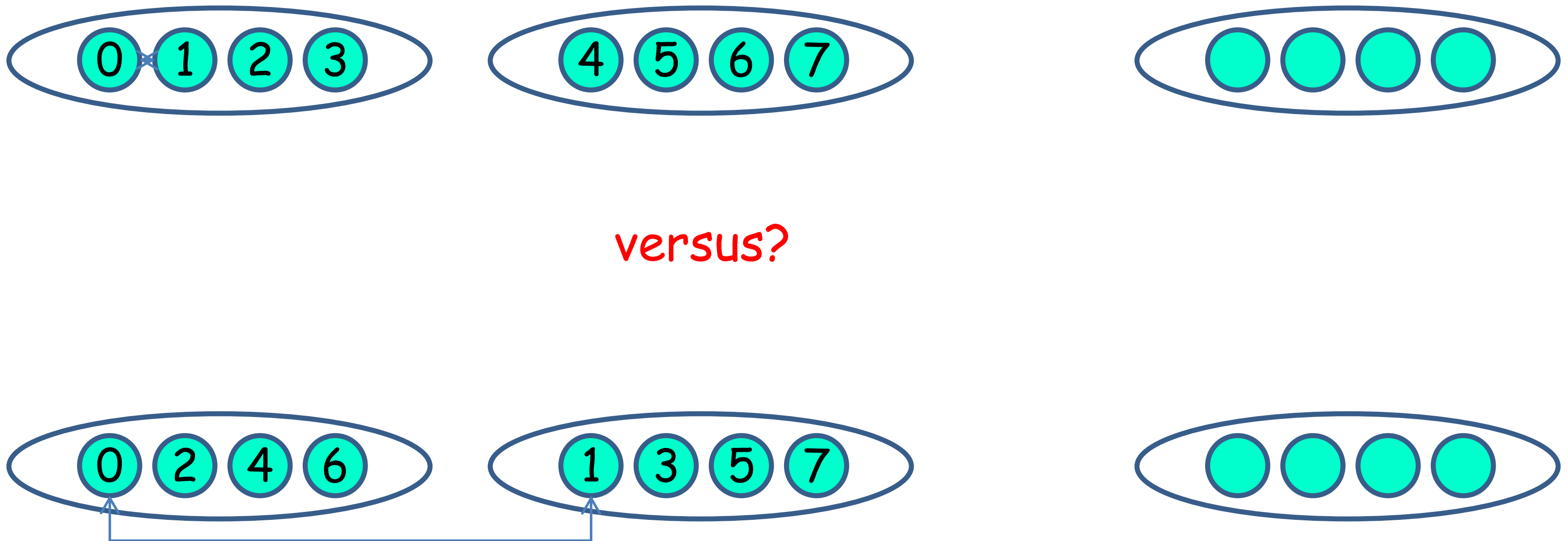
Parallel Computing

MPI_<function> for bidirectional bisection BW measurement:

```
if (rank%2==0&&rank<size-1) {
  MPI_Sendrecv(sendbuf,c,type,rank+1,0,
               recvbuf,c,type,rank+1,0,comm,
               MPI_STATUS_IGNORE);
} else if (rank%2==1) {
  MPI_Sendrecv(sendbuf,c,type,rank-1,0,
               recvbuf,c,type,rank-1,0,comm,
               MPI_STATUS_IGNORE);
} else {
  MPI_Sendrecv(sendbuf,c,type,rank,0,
               recvbuf,c,type,rank,0,comm,
               MPI_STATUS_IGNORE);
}
```

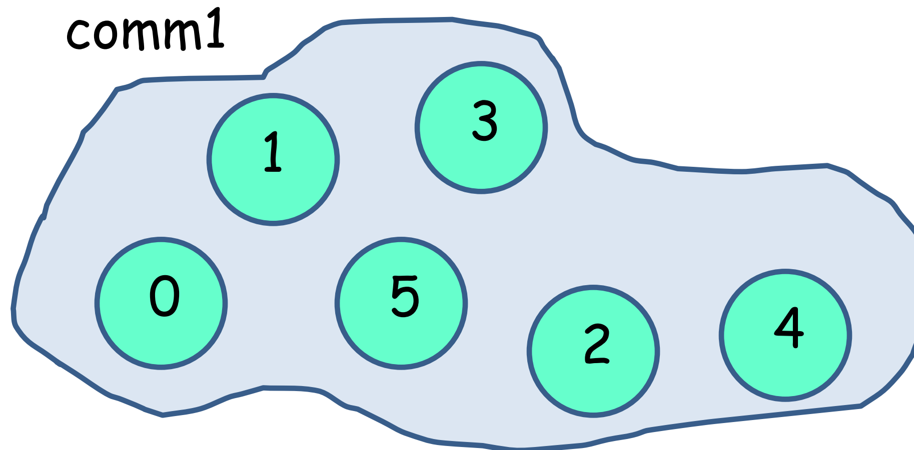

©Jesper Larsson Träff

SMP system communication performance

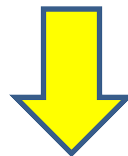


„nodes" with „fast" (shared memory) communication, interconnected with „slower" (1-ported) network

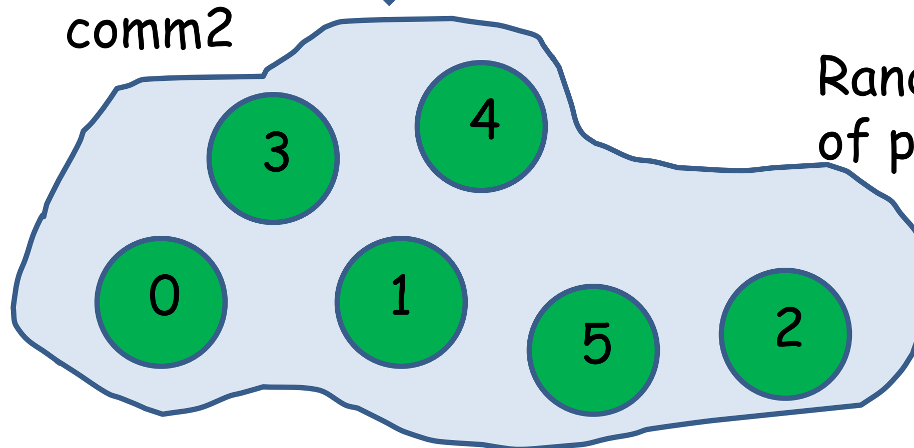
©Jesper Larsson Träff



„Random" communicator: Random association of processes to processors



MPI_Comm_split(comm1,0,rand(),&comm2);



Random key determines order of processes in comm2