AMPI: Adaptive MPI Tutorial

Celso Mendes & Chao Huang

Parallel Programming Laboratory

University of Illinois at Urbana-Champaign

Motivation

- Challenges
  - New generation parallel applications are:
    - Dynamically varying: load shifting, adaptive refinement
  - Typical MPI implementations are:
    - Not naturally suitable for dynamic applications
  - Set of available processors:
    - May not match the natural expression of the algorithm
- AMPI: Adaptive MPI
  - MPI with virtualization: VP (“Virtual Processors”)

Outline

- MPI basics
- Charm++/AMPI introduction
- How to write AMPI programs
  - Running with virtualization
- How to convert an MPI program
- Using AMPI extensions
  - Automatic load balancing
  - Non-blocking collectives
  - Checkpoint / restart mechanism
  - Interoperability with Charm++
  - ELF and global variables
- Recent work

MPI Basics

- Standardized message passing interface
  - Passing messages between processes
  - Standard contains the technical features proposed for the interface
  - Minimally, 6 basic routines:
    - int MPI_Init(int *argc, char ***argv)
      int MPI_Finalize(void)
    - int MPI_Comm_size(MPI_Comm comm, int *size)
      int MPI_Comm_rank(MPI_Comm comm, int *rank)
    - 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)

MPI Basics

- MPI-1.1 contains 128 functions in 6 categories:
  - Point-to-Point Communication
  - Collective Communication
  - Groups, Contexts and Communicators
  - Process Topologies
  - MPI Environmental Management
  - Profiling Interface
- Language bindings: for Fortran, C
- 20+ implementations reported

MPI Basics

- MPI-2 Standard contains:
  - Further corrections and clarifications for the MPI-1 document
  - Completely new types of functionality
    - Dynamic processes
    - One-sided communication
    - Parallel I/O
  - Added bindings for Fortran 90 and C++
  - Lots of new functions: 188 for C binding

AMPI Status

- Compliance to MPI-1.1 Standard
  - Missing: error handling, profiling interface
- Partial MPI-2 support
  - One-sided communication
  - ROMIO integrated for parallel I/O
  - Missing: dynamic process management, language bindings

MPI Code Example: Hello World!

```
#include <stdio.h>
#include <mpi.h>

int main( int argc, char *argv[] )
{
  int size,myrank;
  MPI_Init(&argc, &argv);

  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  printf( "[%d] Hello, parallel world!\n", myrank );

  MPI_Finalize();
  return 0;
}
```

[Example: hello, in MPI…]

Another Example: Send/Recv

```
...
double a[2], b[2];
MPI_Status sts;

if(myrank == 0){
  a[0] = 0.3;   a[1] = 0.5;
  MPI_Send(a,2,MPI_DOUBLE,1,17,MPI_COMM_WORLD);
}else if(myrank == 1){
  MPI_Recv(b,2,MPI_DOUBLE,0,17,MPI_COMM_WORLD,&sts);
  printf(“[%d] b=%f,%f\n”,myrank,b[0],b[1]);
}
...
```

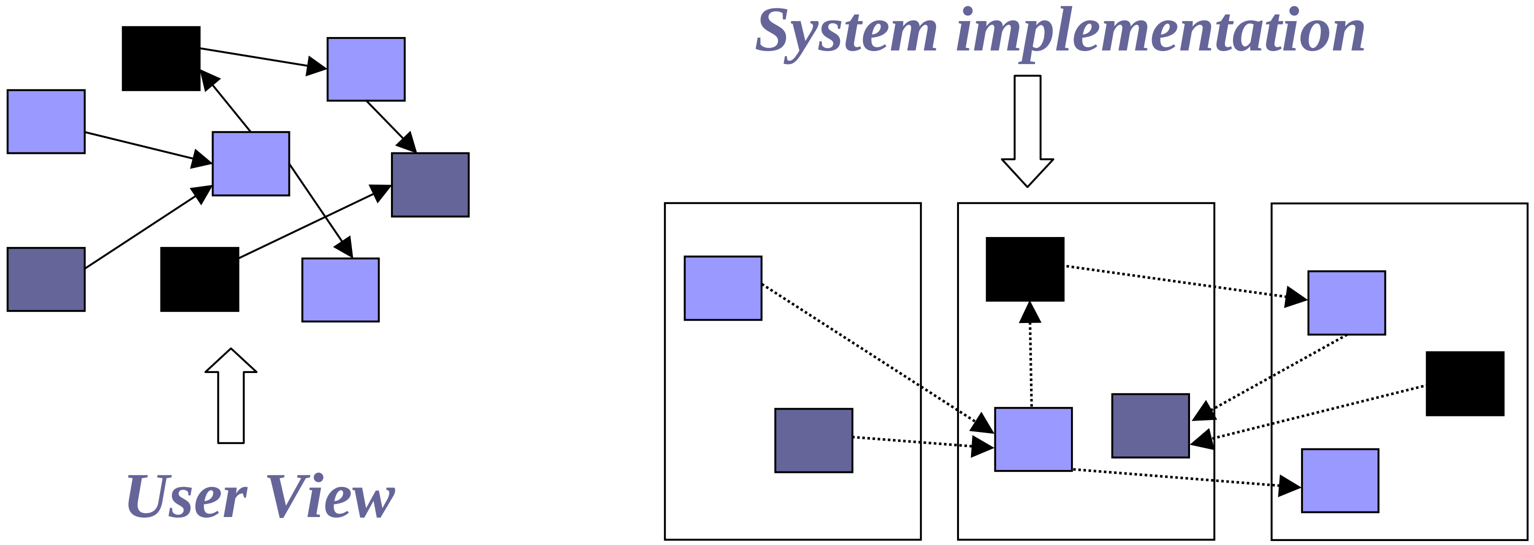
[Example: later…]

Outline

- MPI basics
- Charm++/AMPI introduction
- How to write AMPI programs
  - Running with virtualization
- How to convert an MPI program
- Using AMPI extensions
  - Automatic load balancing
  - Non-blocking collectives
  - Checkpoint/restart mechanism
  - Interoperability with Charm++
  - ELF and global variables
- Recent work

Charm++

- Basic idea of processor virtualization
  - User specifies interaction between objects (VPs)
  - Runtime-system maps VPs onto physical processors
  - Typically, # virtual processors > # processors

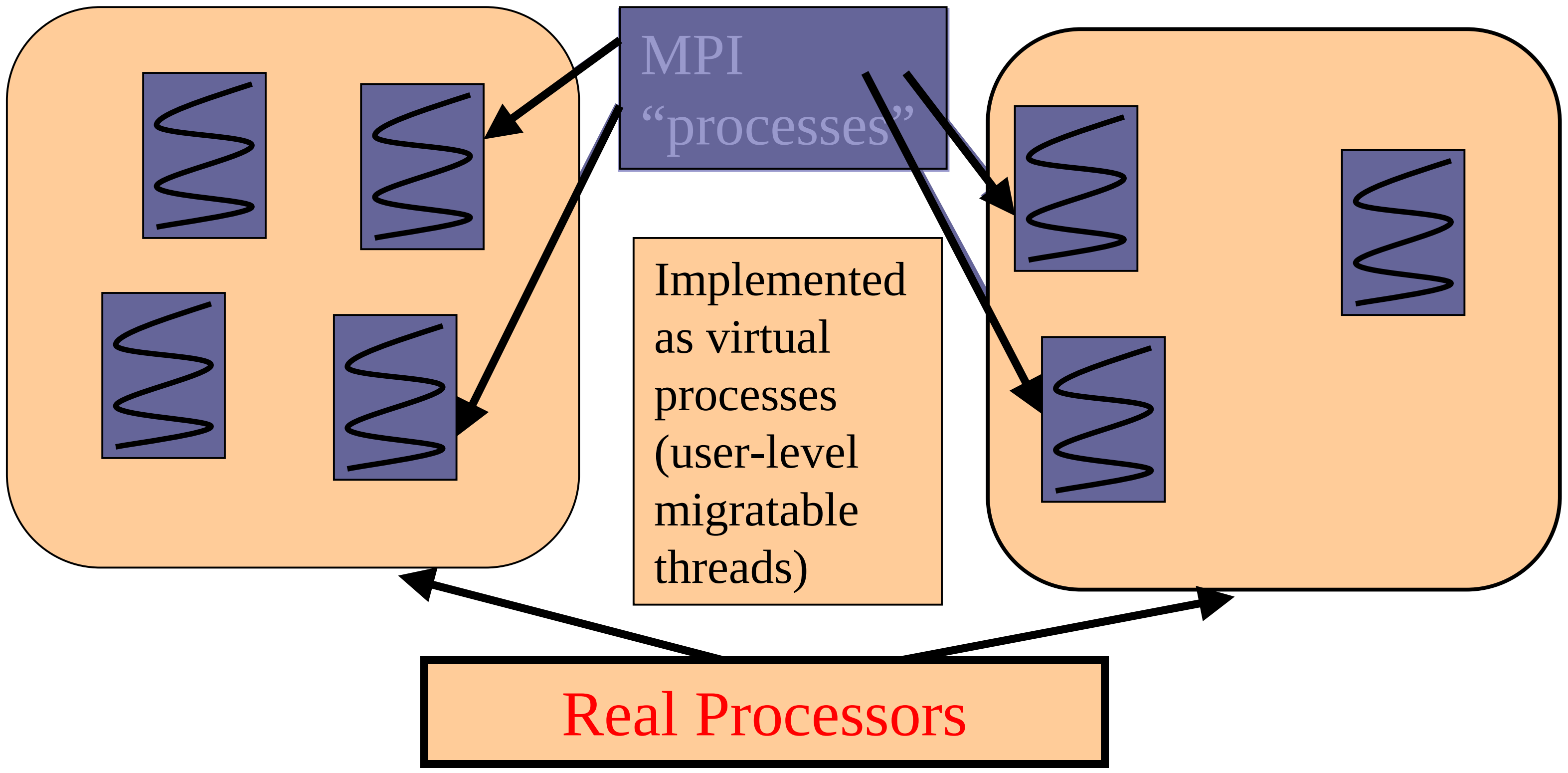


Charm++

- Charm++ characteristics
  - Data driven objects
  - Asynchronous method invocation
  - Multiple objects mapped per processor
  - Load balancing, static and dynamic
  - Portability
- Charm++ features explored by AMPI
  - User level threads, do not block CPU
  - Light-weight: small context-switch time (~ 1μs)
  - Migratable threads

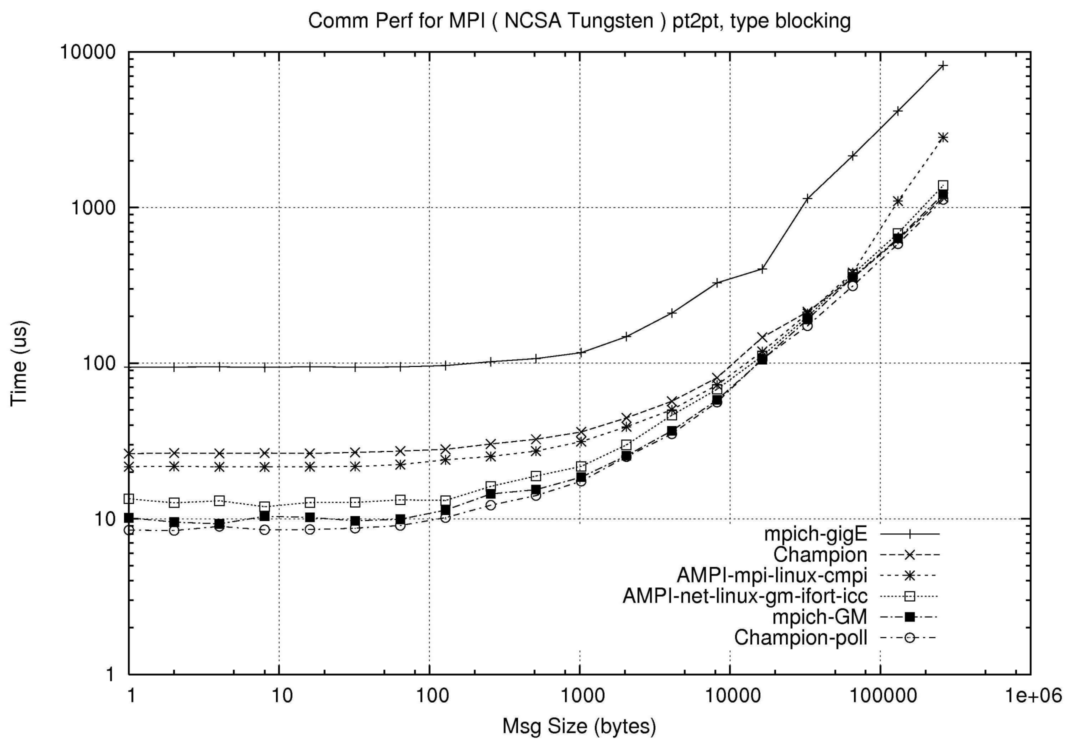
AMPI: MPI with Virtualization

- Each virtual process implemented as a user-level *thread* embedded in a Charm++ *object*


MPI "processes"
Implemented as virtual processes (user-level migratable threads)
Real Processors

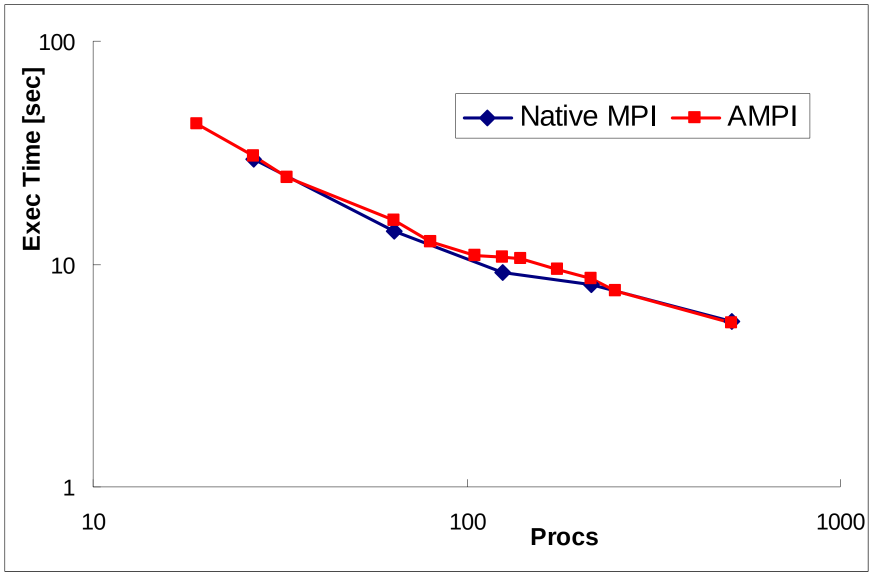
Comparison with Native MPI

- Performance
  - Slightly worse w/o optimization
  - Can be improved, via Charm++
  - Run time parameters for better performance



Comparison with Native MPI

- Flexibility
  - Big runs on any number of processors
  - Fits the nature of algorithms



Problem setup: 3D stencil calculation of size 240^3 run on Lemieux.

AMPI runs on any # of PE's (eg 19, 33, 105). Native MPI needs $P=K^3$

Building Charm++ / AMPI

- Download website:
  - http://charm.cs.uiuc.edu/download/
  - Please register for better support
- Build Charm++/AMPI
  - *> ./build <target> <version> <options> [charmc-options]*
  - See *README* file for details
  - To build AMPI:

    > ./build AMPI net-linux -g (-O3)

Outline

- MPI basics
- Charm++/AMPI introduction
- How to write AMPI programs
  - Running with virtualization
- How to convert an MPI program
- Using AMPI extensions
  - Automatic load balancing
  - Non-blocking collectives
  - Checkpoint/restart mechanism
  - Interoperability with Charm++
  - ELF and global variables
- Recent work

How to write AMPI programs (1)

- Write your normal MPI program, and then…
- Link and run with Charm++
  - Build your charm with target *AMPI*
  - Compile and link with *charmc*
    - include *charm/bin/* in your path
    - > ***charmc*** *-o hello hello.c* ***-language ampi***
  - Run with *charmrun*
    - > *charmrun hello*

How to write AMPI programs (2)

- One can run *most* MPI programs with Charm++
- *mpirun –npK* ≡ *charmrun prog +pK*
  - MPI's machinefile: Charm's *nodelist* file
- Example - Hello World! (via *charmrun*)

How to write AMPI programs (3)

- Avoid using global variables
- Global variables are dangerous in multithreaded programs
  - Global variables are shared by all the threads on a processor and can be changed by any of the threads
  - Example:

Thread 1	Thread 2
count=1	
block in MPI_Recv	
	count=2
	block in MPI_Recv
b=count	

time

incorrect value is read!

How to run AMPI programs (1)

- We can run multithreaded on one processor
- Running with many virtual processors:
  - +*p* command line option: # of physical processors
  - +*vp* command line option: # of virtual processors
  - > *charmrun hello +p3 +vp8*
- Example - Hello Parallel World!
- Example - 2D Jacobi Relaxation

How to run AMPI programs (2)

- Multiple initial processor mappings are possible

  > charmrun hello +p3 +vp6 +mapping <map>

  - Available mappings at program initialization:
    - RR_MAP: Round-Robin (cyclic) {(0,3)(1,4)(2,5)}
    - BLOCK_MAP: Block (default) {(0,1)(2,3)(4,5)}
    - PROP_MAP: Proportional to processors' speeds {(0,1,2,3)(4)(5)}

➢ Example – Mapping…

How to run AMPI programs (3)

- Specify stack size for each thread
  - Set smaller/larger stack sizes
  - Notice that thread's stack space is unique acros processors
  - Specify stack size for each thread with +*tcharm_stacksize* command line option:

    charmrun hello +p2 +vp8 +tcharm_stacksize 8000000
  - Default stack size is 1 MByte for each thread
  - Example – bigstack
    - Small array, many VP's x Large array, any VP's

Outline

- MPI basics
- Charm++/AMPI introduction
- How to write AMPI programs
  - Running with virtualization
- How to convert an MPI program
- Using AMPI extensions
  - Automatic load balancing
  - Non-blocking collectives
  - Checkpoint/restart mechanism
  - Interoperability with Charm++
  - ELF and global variables
- Recent work

How to convert an MPI program

- Remove global variables if possible
- If not possible, *privatize* global variables
  - Pack them into struct/TYPE or class
    - Allocate struct / type in heap or stack

Original Code

```
MODULE shareddata
  INTEGER :: myrank
  DOUBLE PRECISION :: xyz(100)
END MODULE
```

AMPI Code

```
MODULE shareddata
  TYPE chunk
    INTEGER :: myrank
    DOUBLE PRECISION :: xyz(100)
  END TYPE
END MODULE
```

How to convert an MPI program

Original Code

```
PROGRAM MAIN
  USE shareddata
  include 'mpif.h'
  INTEGER :: i, ierr
  CALL MPI_Init(ierr)
  CALL MPI_Comm_rank(
       MPI_COMM_WORLD,
       myrank, ierr)
  DO i = 1, 100
    xyz(i) = i + myrank
  END DO
  CALL subA
  CALL MPI_Finalize(ierr)
END PROGRAM
```

AMPI Code

```
SUBROUTINE MPI_Main
  USE shareddata
  USE AMPI
  INTEGER :: i, ierr
  TYPE(chunk), pointer :: c
  CALL MPI_Init(ierr)
  ALLOCATE(c)
  CALL MPI_Comm_rank(
       MPI_COMM_WORLD,
       c%myrank, ierr)
  DO i = 1, 100
    c%xyz(i) = i + c%myrank
  END DO
  CALL subA(c)
  CALL MPI_Finalize(ierr)
END SUBROUTINE
```

How to convert an MPI program

Original Code

```
SUBROUTINE subA
  USE shareddata
  INTEGER :: i
  DO i = 1, 100
    xyz(i) = xyz(i) + 1.0
  END DO
END SUBROUTINE
```

AMPI Code

```
SUBROUTINE subA(c)
  USE shareddata
  TYPE(chunk) :: c
  INTEGER :: i
  DO i = 1, 100
    c%xyz(i) = c%xyz(i) + 1.0
  END DO
END SUBROUTINE
```

- C examples can be found in the AMPI manual

How to convert an MPI program

- Fortran program entry point: MPI_Main

```
program pgm    →   subroutine MPI_Main
...                 ...
end program         end subroutine
```

- C program entry point is handled automatically, via *mpi.h*

Outline

- MPI basics
- Charm++/AMPI introduction
- How to write AMPI programs
  - Running with virtualization
- How to convert an MPI program
- Using AMPI extensions
  - Automatic load balancing
  - Non-blocking collectives
  - Checkpoint/restart mechanism
  - Interoperability with Charm++
  - ELF and global variables
- Recent work

AMPI Extensions

- Automatic load balancing
- Non-blocking collectives
- Checkpoint/restart mechanism
- Multi-module programming
- ELF and global variables

Automatic Load Balancing

- Load imbalance in dynamic applications hurts the performance
- Automatic load balancing: *MPI_Migrate*()
  - Collective call informing the load balancer that the thread is ready to be migrated, if needed.
  - If there is a load balancer present:
    - First sizing, then packing on source processor
    - Sending stack and packed data to the destination
    - Unpacking data on destination processor

Automatic Load Balancing

- To use automatic load balancing module:
  - Link with Charm’s LB modules

    > charmc –o pgm hello.o -language ampi -module EveryLB

  - Run with +balancer option

    > charmrun pgm +p4 +vp16 +balancer GreedyCommLB

Automatic Load Balancing

- Link-time flag *-memory isomalloc* makes heap-data migration transparent
  - Special memory allocation mode, giving allocated memory the same virtual address on all processors
  - Ideal on 64-bit machines
  - Should fit in most cases and highly recommended

Automatic Load Balancing

- Limitation with isomalloc:
  - Memory waste
    - 4KB minimum granularity
    - Avoid small allocations
  - Limited space on 32-bit machine

- Alternative: PUPer
  - Manually Pack/UnPack migrating data
    (see the AMPI manual for PUPer examples)

Automatic Load Balancing

- Group your global variables into a data structure
- Pack/UnPack routine (a.k.a. PUPer)
  - heap data –(Pack)–>

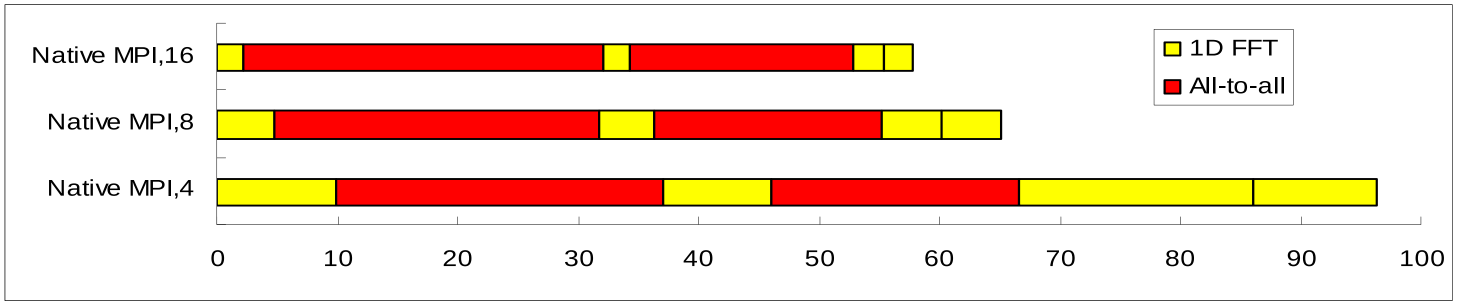
    network message

    –(Unpack)–> heap data

- Example – Load balancing

Collective Operations

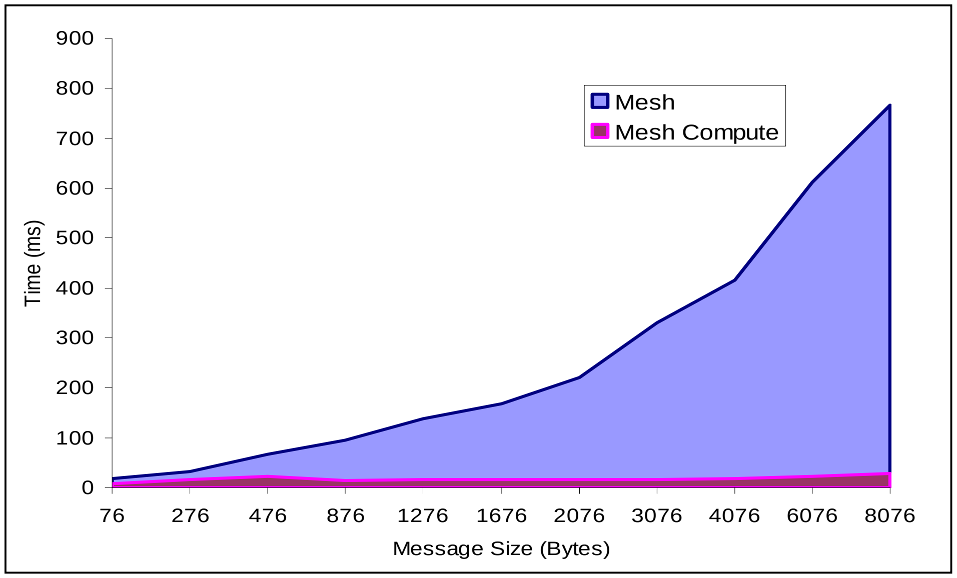
- Problem with collective operations
  - Complex: involving many processors
  - Time consuming: designed as blocking calls in MPI



Time breakdown of 2D FFT benchmark [ms]

(Computation is a small proportion of elapsed time)

Motivation for Collective Communication Optimization



Time breakdown of an all-to-all operation using Mesh library

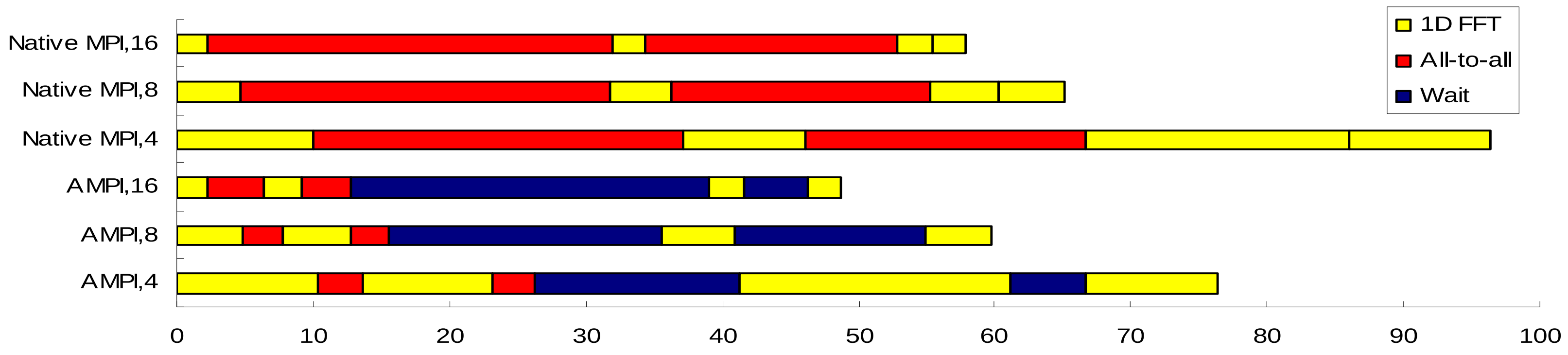
- Computation is only a small proportion of the elapsed time
- A number of optimization techniques are developed to improve collective communication performance

Asynchronous Collectives

- Our implementation is asynchronous
  - Collective operation posted
  - test/wait for its completion
  - Meanwhile useful computation can utilize CPU

```
MPI_Ialltoall( … , &req);
/* other computation */
MPI_Wait(req);
```

Asynchronous Collectives



Time breakdown of 2D FFT benchmark [ms]

- VPs implemented as threads
- Overlapping computation with waiting time of collective operations
- Total completion time reduced

Checkpoint/Restart Mechanism

- Large scale machines may suffer from fails
- Checkpoint/restart mechanism
  - State of applications checkpointed to disk files
  - Capable of restarting on different # of PE's
  - Facilitates future efforts on fault tolerance

Checkpoint/Restart Mechanism

- Checkpoint with collective call
  - In-disk: `MPI_Checkpoint(DIRNAME)`
  - In-memory: `MPI_MemCheckpoint(void)`
  - Synchronous checkpoint
- Restart with run-time option
  - In-disk: > *./charmrun pgm +p4 +restart DIRNAME*
  - In-memory: automatic failure detection and resurrection

➢ Example: checkpoint/restart an AMPI program

Interoperability with Charm++

- Charm++ has a collection of support libraries
- We can make use of them by running Charm++ code in AMPI programs
- Also we can run MPI code in Charm++ programs

➢ Example: interoperability with Charm++

ELF and global variables

- Global variables are not thread-safe
  - Can we switch global variables when we switch threads?
- The Executable and Linking Format (ELF)
  - Executable has a Global Offset Table containing global data
  - GOT pointer stored at *%ebx* register
  - Switch this pointer when switching between threads
  - Support on Linux, Solaris 2.x, and more
- Integrated in Charm++/AMPI
  - Invoked by compile time option *-swapglobals*

➢ Example: thread-safe global variables

Performance Visualization

- Projections for AMPI
  - Register your function calls:

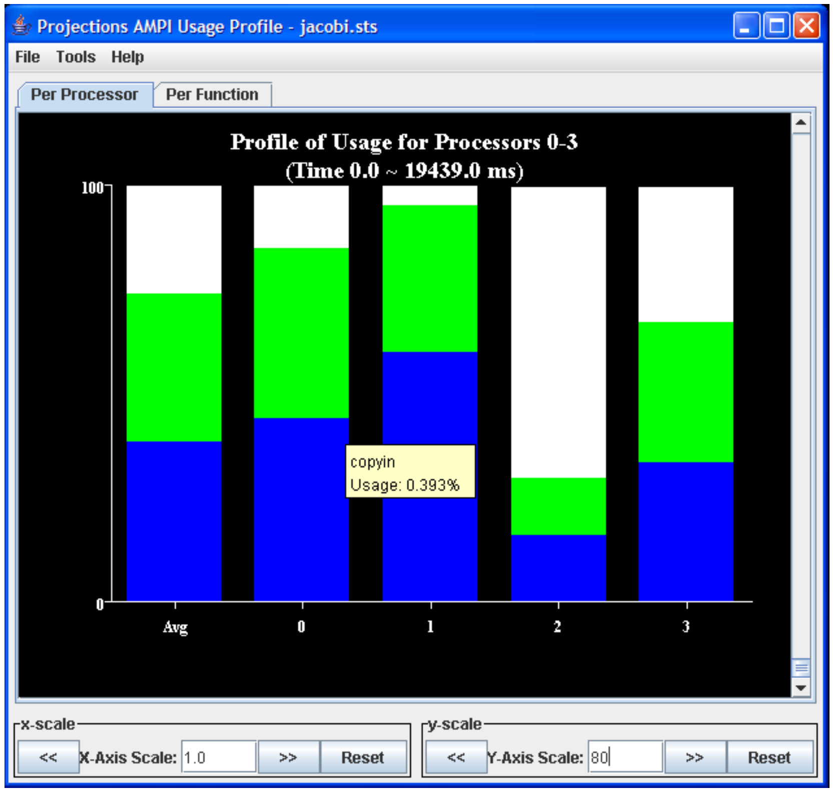
```
extern int traceRegisterFunction(const char *name, int idx);
```

  - Bracket the code fragment you choose to trace:

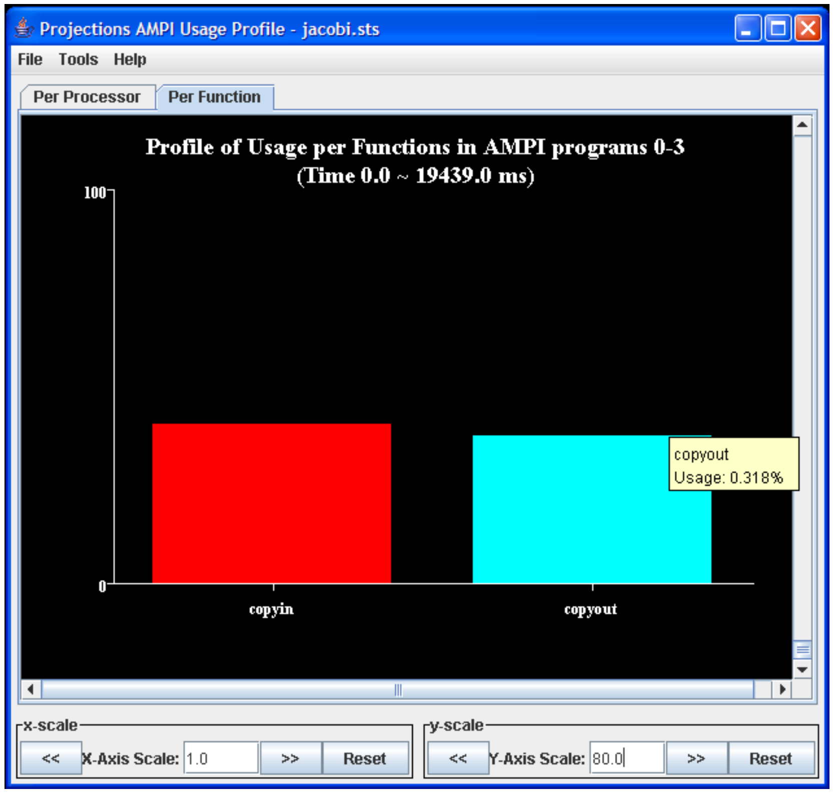
```
extern void traceBeginFuncProj(char *name,char *filename, int line);
. . . <fragment > . . .
extern void traceEndFuncProj(char *name);
```

  - Your code will be instrumented as a Projections event

Performance Visualization



Per Processor View



Per Function View

Outline

- MPI basics
- Charm++/AMPI introduction
- How to write AMPI programs
  - Running with virtualization
- How to convert an MPI program
- Using AMPI extensions
  - Automatic load balancing
  - Non-blocking collectives
  - Checkpoint/restart mechanism
  - Interoperability with Charm++
  - ELF and global variables
- Recent work

Recent Work

- Message logging
  - Problem: sequential running environment (architecture simulator) for a node of a parallel system
  - Solution: log output values of all MPI calls, duplicating MPI environment
  - Writing run in parallel, reading run as sequential program

```
#define AMPIMSGLOG 1
Writing log: > ./charmrun ./pgm +p4 +msgLogWrite
  +msgLogRank 2
Reading log: > ./pgm +msgLogRead +msgLogRank 2
```

  - Issue: log file might get too large
    - Using zlib to compress (1:6~7 compression)

Thank You!

Free download and manual available at:
http://charm.cs.uiuc.edu/

Parallel Programming Lab
at University of Illinois