Adaptive MPI Tutorial

Chao Huang

chuang10@uiuc.edu

Parallel Programming Laboratory

University of Illinois

Motivation

- Highly dynamic parallel applications
  - Adaptive mesh refinement
  - Crack propagation
- Usually limited supercomputing platforms availability
  - Cannot always get 2^n PEs required by parallel model
- Cause load imbalance and programming complexity

Motivation

- Little change to normal MPI program
- Load balancing
  - System can automatically migrate virtual MPI processors to achieve load balance
- Virtual processors
  - +*vp* option allows execution on desired number of virtual processors
- MPI extensions:
  - More asynchronous calls

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 and C++
- 20+ different implementations reported.

Example: Hello World!

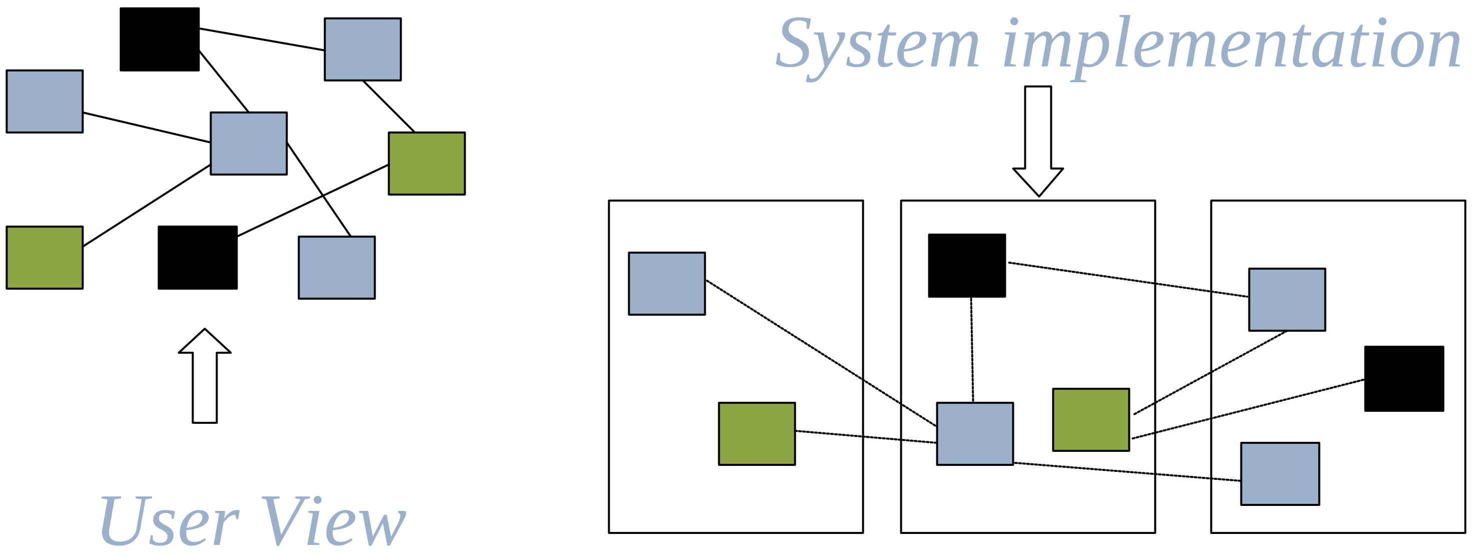
```
#include <stdio.h>
#include <mpi.h>

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: Send/Recv

```
...

double a[2] = {0.3, 0.5};
double b[2] = {0.7, 0.9};
MPI_Status sts;

if(myrank == 0){
  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);
}

...
```

Charm++

- Object-based virtualization
  - Divide the computation into a large number of pieces: *Chares*
  - Let the *system* map objects to processors
  - User is concerned with interaction between objects



Charm++

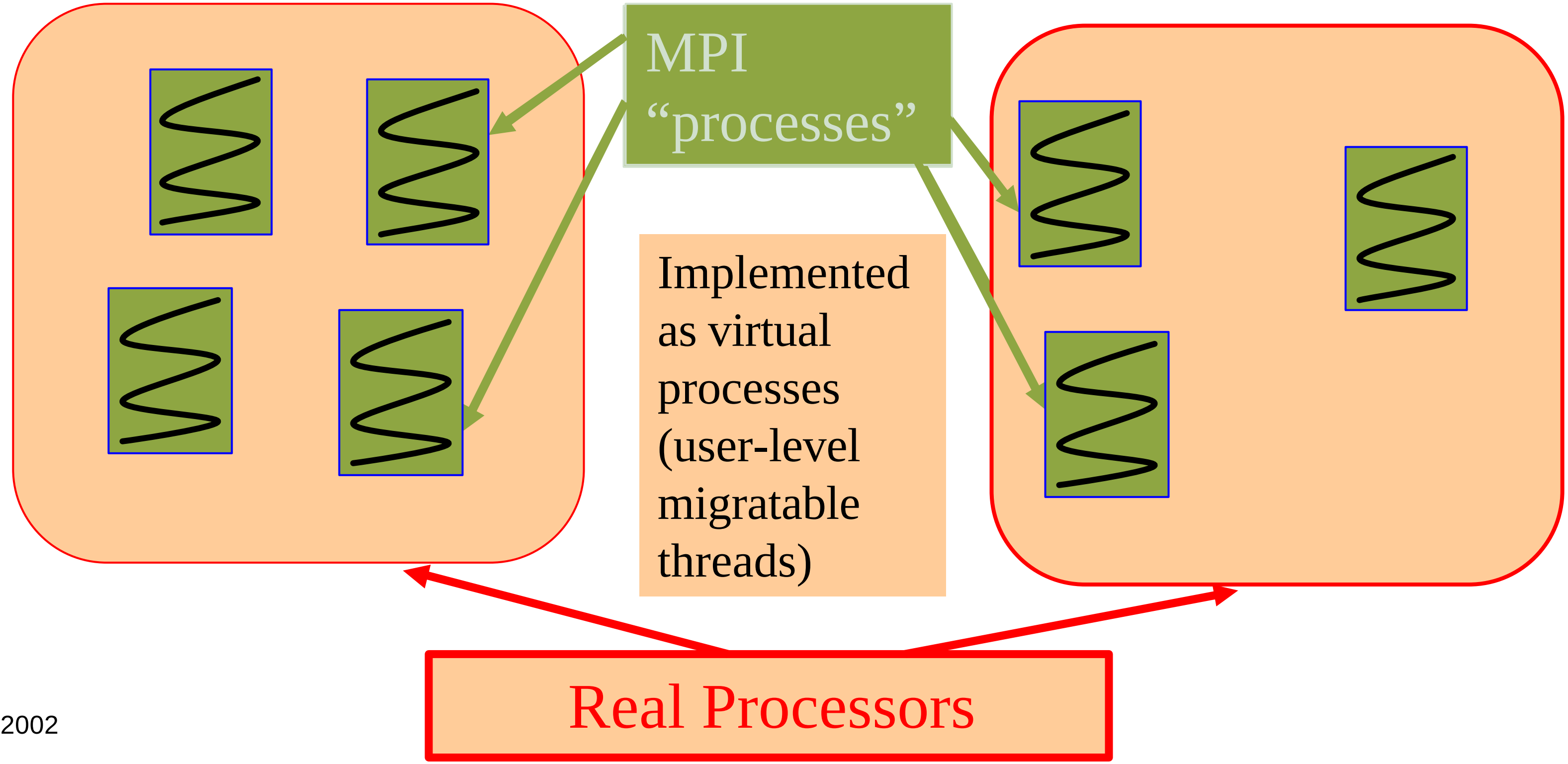
- Features
  - Data driven objects
  - Asynchronous method invocation
  - Mapping multiple objects per processor
  - Load balancing, static and run time
  - Portability
- TCharm
  - User level threads, do not block CPU
  - Language-neutral interface for run-time load balancing via migration

Charm++

- Download and install
  - http://charm.cs.uiuc.edu/download.html
    - Please register
  - Build Charm++/AMPI
    - *> ./build <target> <version> <options> [charmc-options]*
    - To build AMPI:
      - *> ./build AMPI <version> [-g]*

AMPI: MPI with Virtualization

- Each virtual process implemented as a user-level *thread* associated with a message-driven *object*



MPI “processes”

Implemented as virtual processes (user-level migratable threads)

Real Processors

How to write AMPI program (1)

- Write your normal MPI program, and then…
- Link and run with Charm++
  - Build your charm with target *AMPI*
  - Compile and link with *charmc*
    - *charm/bin/mpicc|mpiCC|mpif77|mpif90*
    - *> **charmc** -o hello hello.c **-language ampi***
  - Run with *charmrun*
    - *> charmrun +p3 hello*

How to write AMPI program (1)

- Now we can run MPI program with Charm++
- Demo - Hello World!

How to write AMPI program (2)

- Do not use global variables
- Global variables are dangerous in multithread programs
  - Global variables are shared by all the threads on a processor and can be changed by other thread

Thread 1	Thread2
count=1	
block in MPI_Recv	
	count=2
	block in MPI_Recv
b=count	

How to write AMPI program (2)

- Now we can run multithread on one processor
- Running with many virtual processors
  - +*vp* command line option
  - > *charmrun +p3 hello +vp8*
- Demo - Hello World!
- Demo - 2D Jacobi Relaxation

How to write AMPI program (3)

- Load balancing with migration
- 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 pupped data to the destination
    - Unpacking data on destination processor

How to write AMPI program (3)

- Link-time flag *-memory isomalloc* makes migration transparent
  - Special memory allocation mode, giving allocated memory the same virtual address on all processors
  - Ideal on 64-bit machines
  - No need for PUPer routines: trouble-free
  - Should fit in most cases and we highly recommend it

How to write AMPI program (3)

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

How to write AMPI program (3)

- Pack/UnPack routine (aka PUPer)
  - Heap data –(Pack)–> network message –(Unpack)–> heap data
  - A typical PUPer looks like this:

```
SUBROUTINE chunkpup(p, c)
  USE pupmod
  USE chunkmod
  IMPLICIT NONE
  INTEGER :: p
  TYPE(chunk) :: c

  call pup(p, c%t)
  call pup(p, c%xidx)
  call pup(p, c%yidx)
  call pup(p, c%bxm)
  call pup(p, c%bxp)
  call pup(p, c%bym)
  call pup(p, c%byp)
end subroutine
```

How to write AMPI program (3)

- Demo – Migrating Jacobi Relaxation

How to convert an MPI program

- Remove global variables
  - Pack them into struct/TYPE or class
    - Allocated 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
```

How to run an AMPI program

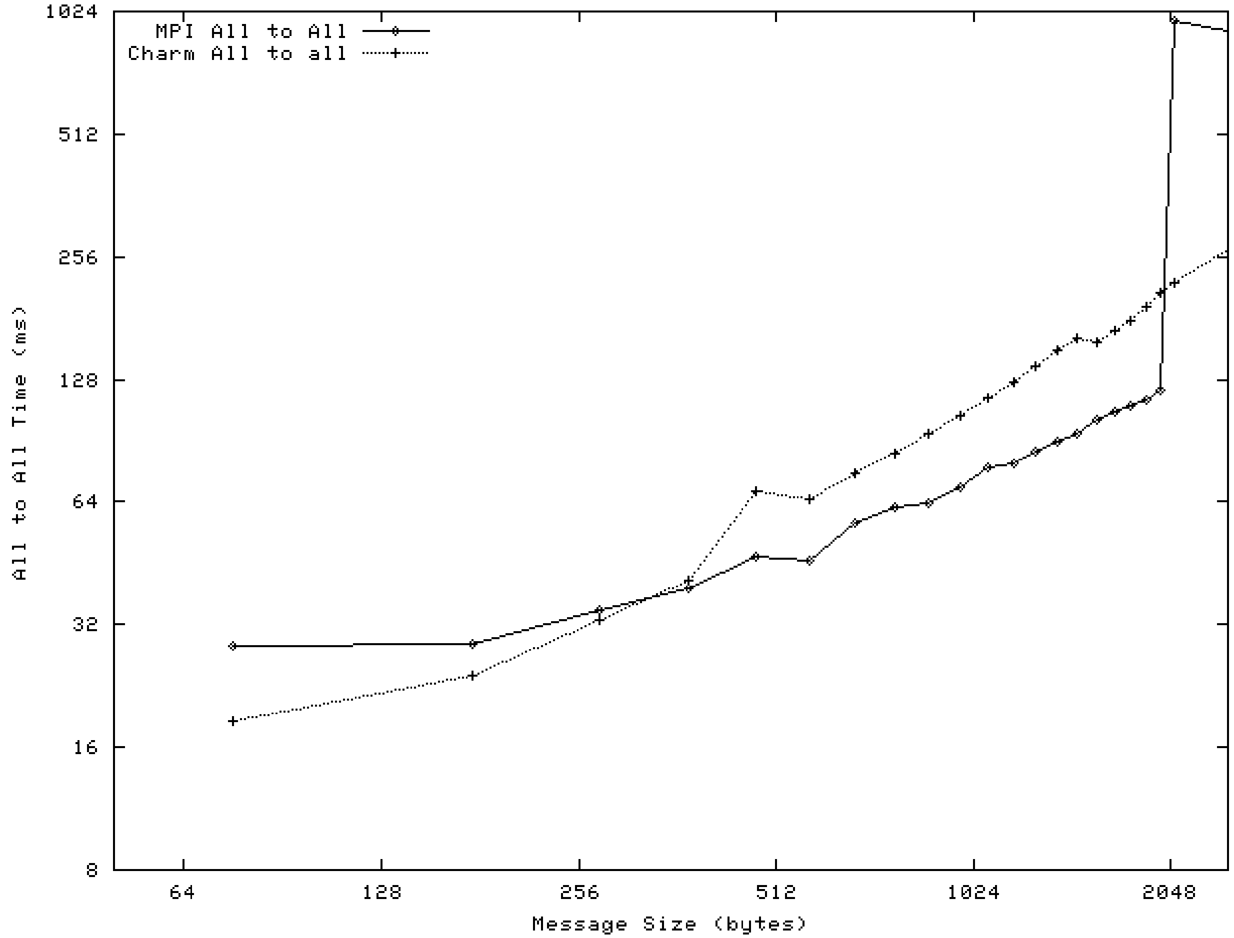
- Use virtual processors
  - Run with +*vp* option
  - Specify stacksize with +*tcharm_stacksize* option
  - Demo – large stack

Communication Optimization

- Collective communications in MPI are complex and time consuming!
- May involve a lot of data movement
- Implemented as blocking calls in MPI
  - MPI_Alltoall
  - MPI_Reduce

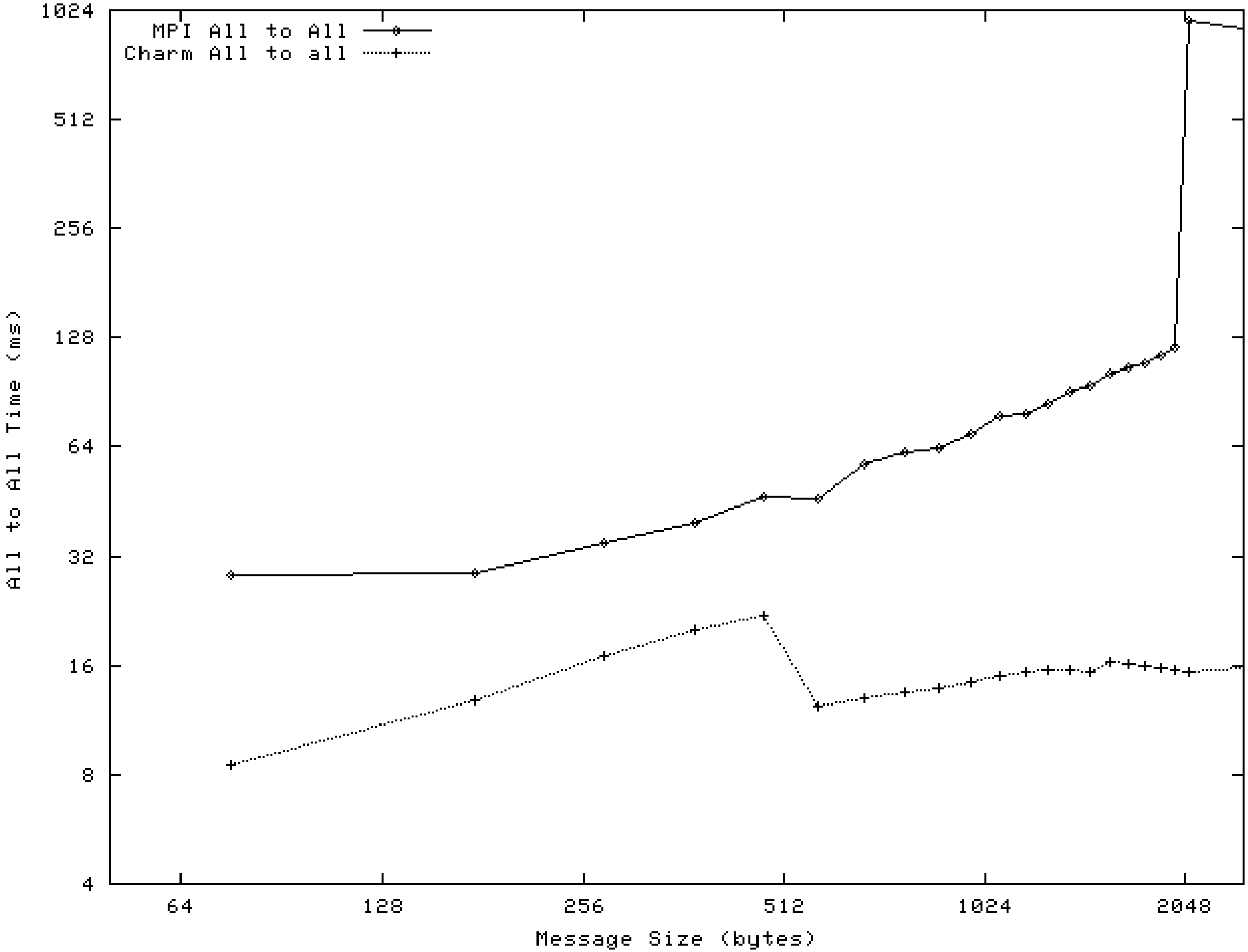
Communication Optimization

Alltoall time on 1K processors



Communication Optimization

Alltoall Software Overhead on 1K processors



Communication Optimization

- Our implementation is asynchronous
- Collective operation is first scheduled
- Each process the polls for its completion
- Implemented through the Charm++ message scheduler

  AMPI_Alltoall_Start(……………………..);

  AMPI_Alltoall_Poll();
- Each processor in the mean time can do useful computation

Future work

- Projector/Projections support
- Read-only data

Future work

- Projections: parallel visualization tool for Charm++
- Projector: enables programs written in language other than Charm++ to output visualization data for Projection