Adaptive MPI

Santos Dumont Supercomputing Summer School 2021

Esteban Meneses, PhD

Advanced Computing Laboratory
Costa Rica High Technology Center

School of Computing
Costa Rica Institute of Technology

emeneses@cenat.ac.cr

2021

Centro Nacional de Alta Tecnología

Outline

Presentation

Parallel Programming Model
- Ideal Features
- Parallel Objects
- Charm++

Adaptive Message Passing Interface
- Introduction
- Communication Optimizations
- Migration

Global Variable Privatization

Where do I come from?

Central America

Costa Rica

is not Puerto Rico
has no standing army since 1949
hosts 6% of world's biodiversity
produces 98% of its electricity from green sources

Costa Rica High Technology Center

CeNAT

UCR TEC UNA UNED uTn Universidad Técnica Nacional CONARE

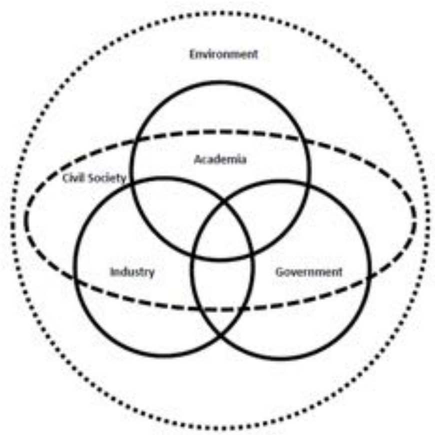
CeNAT Centro Nacional de Alta Tecnología

PRIAS

LANOTEC Laboratorio Nacional de Nanotecnología

CENIBiot

CNCA Colaboratorio Nacional de Computación Avanzada



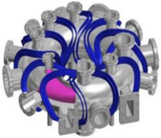
Development through Knowledge

CeNAT-CONARE Campus, Pavas, San José

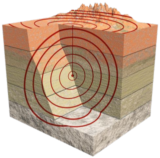
Collaborative Research Projects

Accelerating scientific discovery

Physics

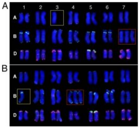


Seismology



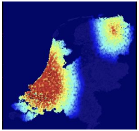
Biodiversity

Bioinformatics



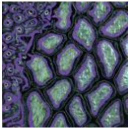
Mobility

Epidemics



HPC

Image Analysis



Trainings and Seminars

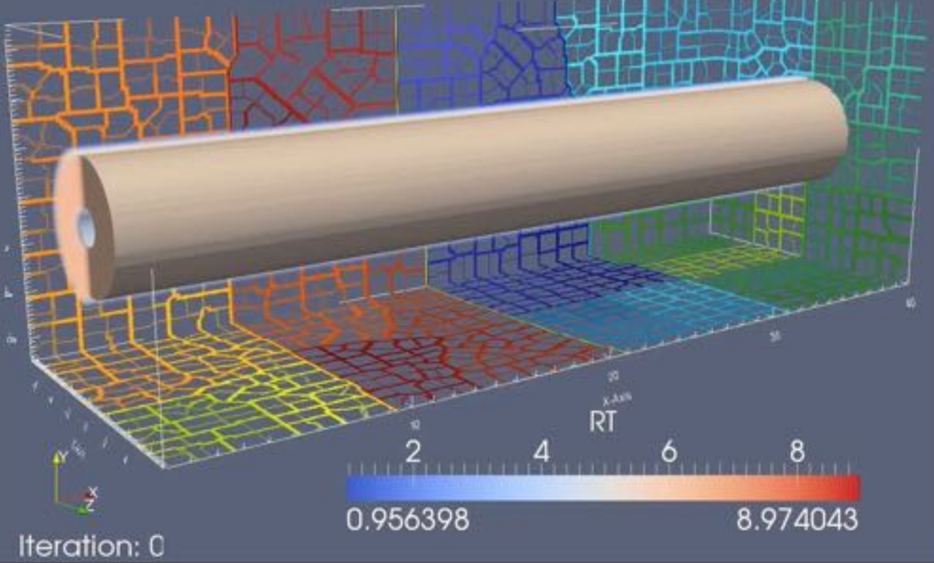
Advanced Computing Laboratory

Costa Rica HPC School

Costa Rica Big Data School

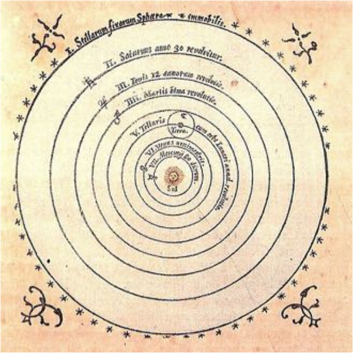
Parallel Programming Model

Flame Simulation on IPLMCFD Application



Parallel Programming Model

An abstract machine on which parallel programs will execute



Copernicus's heliocentric model

- Components:
  - Execution model: how code gets executed
  - Memory model: how data moves between memory hierarchy
- Most parallel systems expose multiple parallel programming models

Desirable Features
The HPC Holy Grail

- **Performant**: extracts as much performance as possible from the underlying hardware
- **Productive**: expresses abstract algorithms easily
- **Portable**: can be used on any computer
- **Expressive**: allows a broad range of algorithms
- **Scalable**: the general structure of the code persists as more hardware is used

Presentation
Parallel Programming Model
Ideal Features
Parallel Objects
Charm++
Adaptive Message Passing Interface
Introduction
Communication Optimizations
Migration
Global Variable Privatization
Conclusion

Implementation

Alternatives for using the model

- **Library**:
  - An API of function calls
  - Library gets linked with the executable; multiple languages
- **Programming Language Extension**:
  - Additional constructs for parallelism
  - Compiler support for translation
- **New Programming Language**:
  - Design of new language grammar
  - Flexibility to include features

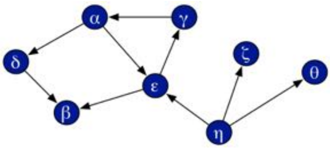
There are only two kinds of languages:
the ones people complain about and the ones nobody uses
Bjarne Stroustrup

Parallel Objects Model

Object-oriented parallel programming

- An application is decomposed into wudus (work and data units)
- Objects are *reactive* entities: interface of remote methods
- All message-passing operations are nonblocking: *asynchronous method invocation*
- A message-driven execution similar to Active Messages

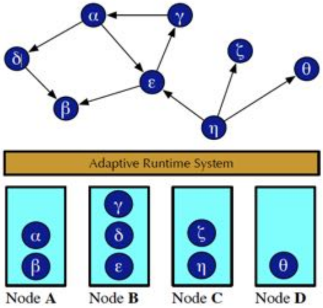
- Objects know how to serialize/deserialize, also called the pack-unpack (*PUP*) framework
- Goals:
  - Latency hiding
  - Load balancing
  - Adaptivity



Introspective Runtime System

Smart and automatic decision making

- A thin layer between the application and the machine
- Based on object-based *overdecomposition*: many more objects than processing entities
- Components:
  - Message scheduler
  - Routing tables
  - Load and communication monitoring



Presentation

Parallel Programming Model
Ideal Features
Parallel Objects
Charm++

Adaptive Message Passing Interface
Introduction
Communication Optimizations
Migration

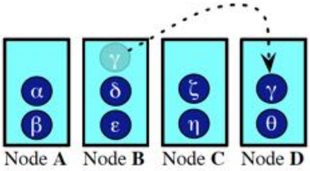
Global Variable Privatization

Conclusion

Migration

Objects can be relocated

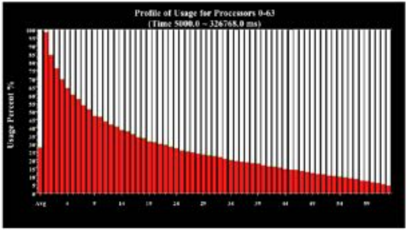
- The underlying system consists of a collection of processing entities (cores, processors, or nodes)
- Objects are distributed among the processing entities
- That assignment may change dynamically if load imbalance arises
- An introspective runtime system detects performance bottlenecks and balances load by moving objects around.

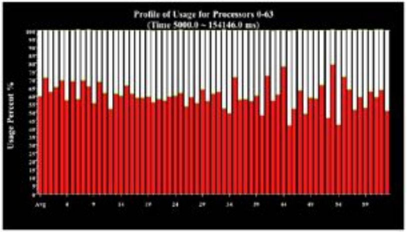


Load Balance

A complex optimization problem

- NP-complete problem: suboptimal, but fast heuristic algorithms
- Goal: avoid overloaded nodes
- Runtime collects load and communication data
- Greedy strategies, graph partitioning
- Runtime system shuffles objects around to avoid overloading
- Dynamic load balance
- Principle of persistence
- Based on PUP framework





Presentation

Parallel Programming Model
Ideal Features
Parallel Objects
Charm++

Adaptive Message Passing Interface
Introduction
Communication Optimizations
Migration

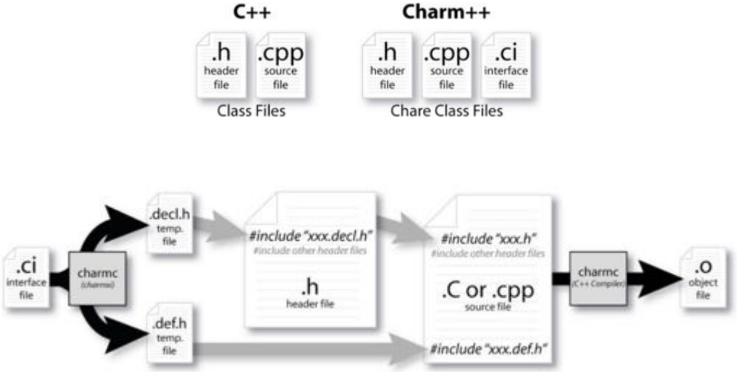
Global Variable Privatization

Conclusion

CeNAT

Charm++

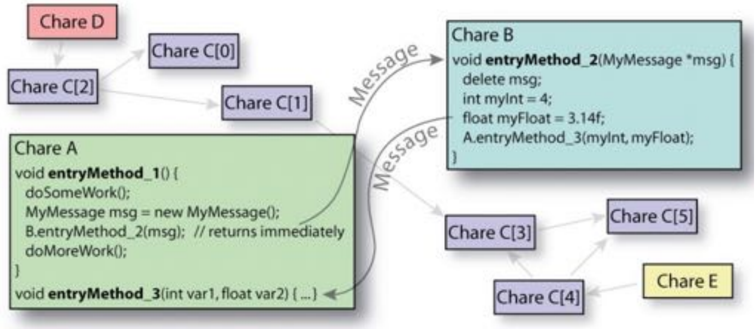
Actively developed since mid 90s



- Objects are called *chares*
- Chare arrays are the main object collection
- Chares export remote *entry methods*

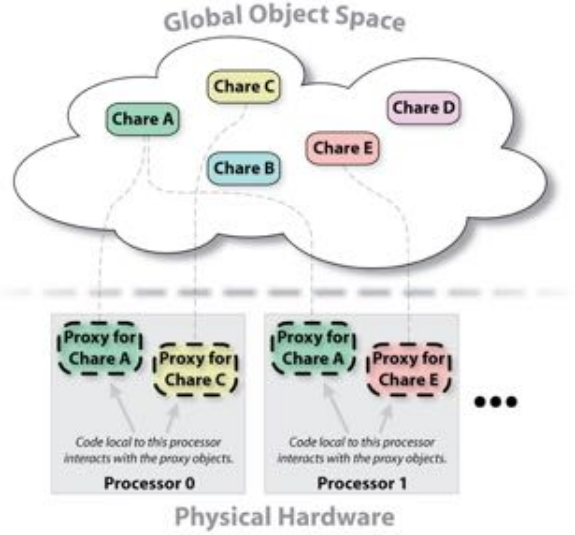
Global Object Space

Entry methods can be called from anywhere



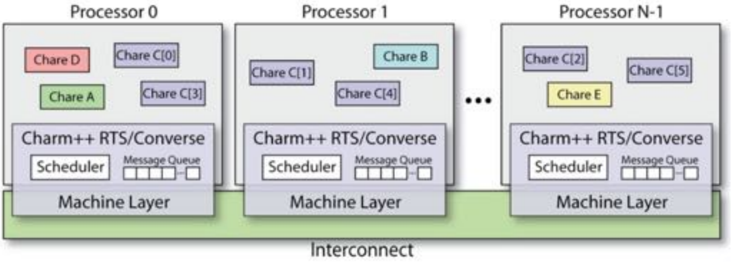
Global Object Space

Proxy chares channel remote method calls



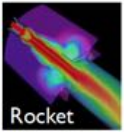
Charm++ Runtime System

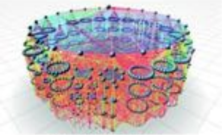
Multiple layers with different abstraction



Charm++ Applications

Span multiple scientific domains





Contagion

PSTIP

Engineering

Presentation

Parallel Programming Model
Ideal Features
Parallel Objects
Charm++

Adaptive Message Passing Interface
Introduction
Communication Optimizations
Migration

Global Variable Privatization

Conclusion

Exercise

Got Charm++?

- Started at the Parallel Programming Laboratory of the University of Illinois at Urbana-Champaign in the mid 90's by Prof. Laxmikant V. Kalé
- Maintained by Charmworks Inc
- Charm++ official website: **http://charmplusplus.org/**
- Get latest release version
- Build Charm++ and AMPI on your computer
  - Linux:
    ```
    ./build AMPI netlrts-linux-x86_64 --with-production
    --enable-error-checking -j4 -k
    ```
  - Mac:
    ```
    ./build AMPI netlrts-darwin-x86_64 --with-production
    --enable-error-checking -j4 -k
    ```


Presentation

Parallel Programming Model
- Ideal Features
- Parallel Objects
- Charm++

Adaptive Message Passing Interface
- Introduction
- Communication Optimizations
- Migration

Global Variable Privatization

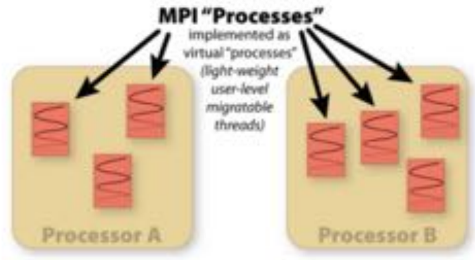
Conclusion

Adaptive Message Passing Interface

Adaptive Message Passing Interface

An MPI implementation on top of Charm++ runtime system

- Enables Charm++ dynamic features for pre-existing MPI codes
- Each MPI rank is wrapped as a Charm++ chare
- The collection of MPI ranks becomes a chare array
- MPI codes run on Charm++ runtime system



Presentation

Parallel Programming Model
Ideal Features
Parallel Objects
Charm++

Adaptive Message Passing Interface
Introduction
Communication Optimizations
Migration

Global Variable Privatization

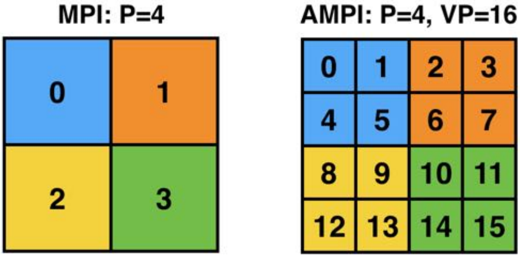
Conclusion

CeNAT

Process Virtualization

AMPI virtualizes MPI ranks

- MPI ranks are implementing as migratable user-level threads rather than OS processes
- Virtualization ratio akin to object overdecomposition



- If one MPI rank is blocked on communication, the scheduler picks other rank to run

Presentation

Parallel Programming Model

Ideal Features
Parallel Objects
Charm++

Adaptive Message Passing Interface

Introduction
Communication Optimizations
Migration

Global Variable Privatization

Conclusion

CeNAT

AMPI Library

AMPI virtualizes MPI ranks

- Another MPI implementation, similar to MPICH, OpenMPI, MVAPICH
- Currently compliant with MPI 2.2 standard
- Benefits:
  - Communication/computation overlap
  - Cache benefits to smaller working sets
  - Dynamic load balancing
  - Fault tolerance
  - Lower latency messaging within a process
  - Reuse existing MPI codes and developer skills, but scale them further
- Disadvantages:
  - Some code modifications are required, v.g., global/static variables shared must be privatized
  - Latest MPI functions might not be supported by AMPI

Communication Optimizations

Speeding up algorithms

- AMPI overlaps communication of one rank with computation of others scheduled on the same core
- Even blocking calls are executed asynchronously
- Supports non-blocking collectives since before MPI-3.0
- AMPI optimizes for communication locality (i.e. neighbor exchanges)
- Can even load balance based on the application communication graph, to improve communication locality dynamically

Presentation

Parallel Programming Model
- Ideal Features
- Parallel Objects
- Charm++

Adaptive Message Passing Interface
- Introduction
- Communication Optimizations
- Migration

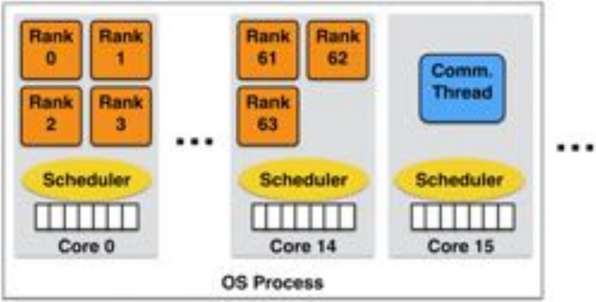
Global Variable Privatization

Conclusion

Communication Optimizations

Internal communications

AMPI offers lower latency and higher bandwidth than process-based MPIs for messages within a core or node



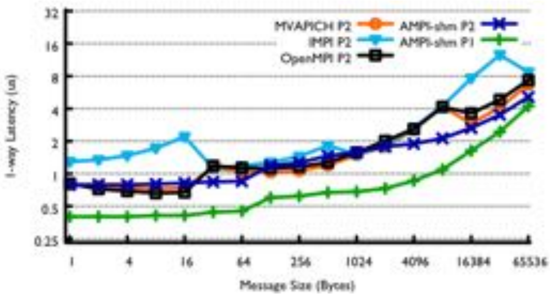
- P1: two ranks on the same core
- P2: two ranks on different cores in the same process

CeNAT

Communication Optimizations
Example, OSU MPI latency benchmark

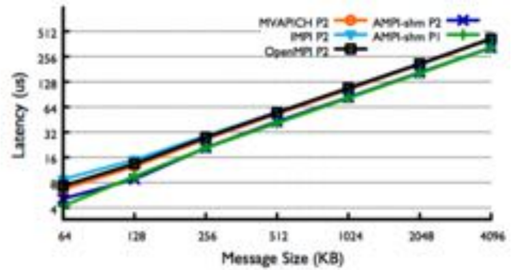
Running on Quartz (Intel Xeon/Omni-Path cluster at LLNL)

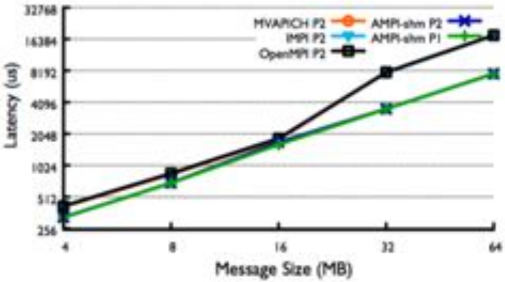
- P1: two ranks on the same core
- P2: two ranks on different cores in the same process



Communication Optimizations

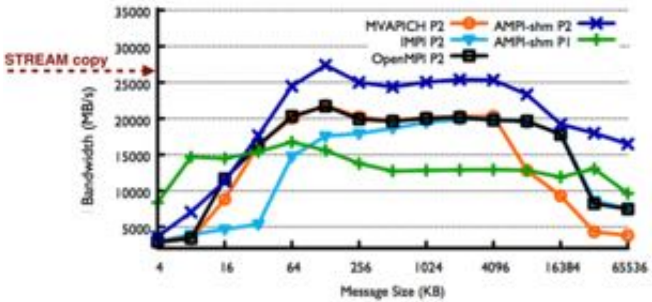
Example, OSU MPI latency benchmark





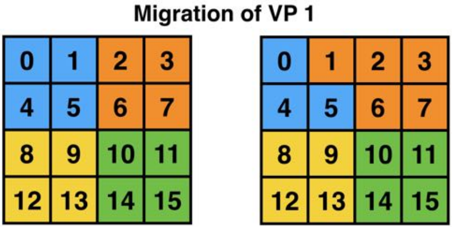
Communication Optimizations

Example, OSU MPI bi-directional bandwidth benchmark



Dynamic Load Balancing
AMPI instructions

- AMPI ranks are migratable across address spaces at runtime



- Add a call to `AMPI_Migrate(MPI_Info)`
- Where `info` is the following:
  `MPI_Info_set(info, ampi_load_balance sync)`

Dynamic Load Balancing

BRAMS weather simulation code

Rodrigues, Eduardo R. et al. *Optimizing an MPI Weather Forecasting Model via Processor Virtualization*, HiPC 2010.

Presentation

Parallel Programming Model

Ideal Features
Parallel Objects
Charm++

Adaptive Message Passing Interface

Introduction
Communication Optimizations
Migration

Global Variable Privatization

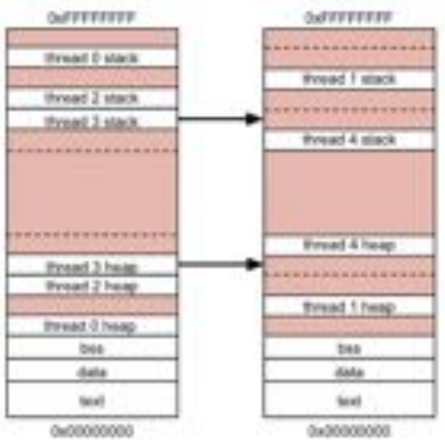
Conclusion

CeNAT

Isomalloc

Memory allocator

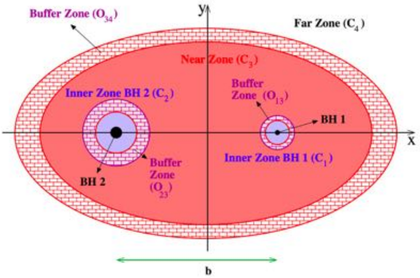
- User-level thread stack + heap
- Reserves globally unique slices of virtual memory on each process for all ranks
- No need for Pack/UnPack routines
- Works on all 64-bit platforms except BGQ and Windows



Dynamic Load Balancing

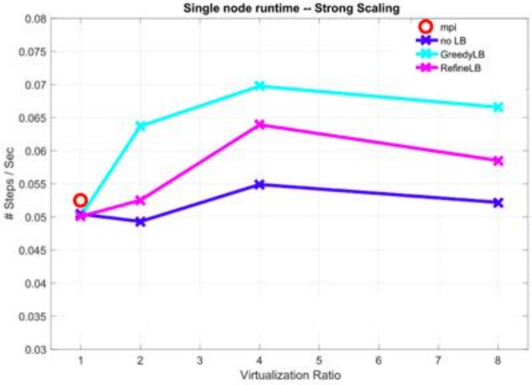
Example, Harm3D application

- Existing MPI astrophysics code developed by Scott Noble at Tulsa (in collaboration with NCSA)
- Imbalanced case: two black holes (zones) move through the grid with 3x more computational work in buffer zone than in near zone



Dynamic Load Balancing

Example, Harm3D application



Fault Tolerance

Checkpointing ranks

- AMPI ranks can be migrated to persistent storage or in remote memories for fault tolerance
- Storage can be disk, SSD, NVRAM
- Online fault detection and recovery
- Just pass a different `MPI_Info` to `AMPI_Migrate()`

```
MPI_Info_set(info1, ampi_checkpoint,
in_memory)
MPI_Info_set(info2, ampi_checkpoint,
to_file=dir_name)
```

Fault Tolerance

Example

```
PlasComCM: iteration =       96, dt =  0.870094D-02, time =  0.835290D+00, cfl =  0.500000D+00, maxT =  0.298000D+03
PlasComCM: iteration =       97, dt =  0.870094D-02, time =  0.843991D+00, cfl =  0.500000D+00, maxT =  0.298000D+03
PlasComCM: iteration =       98, dt =  0.870094D-02, time =  0.852692D+00, cfl =  0.500000D+00, maxT =  0.298000D+03
PlasComCM: iteration =       99, dt =  0.870094D-02, time =  0.861393D+00, cfl =  0.500000D+00, maxT =  0.298000D+03
PlasComCM: iteration =      100, dt =  0.870094D-02, time =  0.870094D+00, cfl =  0.500000D+00, maxT =  0.298000D+03
[0] Checkpoint started
[0] Checkpoint finished in 0.455819 seconds
PlasComCM: iteration =      101, dt =  0.870094D-02, time =  0.878795D+00, cfl =  0.500000D+00, maxT =  0.298000D+03
```

1. Checkpoint

Fault Tolerance

Example

```
PlasComCM: iteration =      96, dt =  0.870094D-02, time =  0.835290D+00, cfl =  0.500000D+00, maxT =  0.298000D+03
PlasComCM: iteration =      97, dt =  0.870094D-02, time =  0.843991D+00, cfl =  0.500000D+00, maxT =  0.298000D+03
PlasComCM: iteration =      98, dt =  0.870094D-02, time =  0.852692D+00, cfl =  0.500000D+00, maxT =  0.298000D+03
PlasComCM: iteration =      99, dt =  0.870094D-02, time =  0.861393D+00, cfl =  0.500000D+00, maxT =  0.298000D+03
PlasComCM: iteration =     100, dt =  0.870094D-02, time =  0.870094D+00, cfl =  0.500000D+00, maxT =  0.298000D+03
[0] Checkpoint started
[0] Checkpoint finished in 0.455819 seconds
PlasComCM: iteration =     101, dt =  0.870094D-02, time =  0.878795D+00, cfl =  0.500000D+00, maxT =  0.298000D+03
PlasComCM: iteration =     102, dt =  0.870094D-02, time =  0.887496D+00, cfl =  0.500000D+00, maxT =  0.298000D+03
PlasComCM: iteration =     103, dt =  0.870094D-02, time =  0.896197D+00, cfl =  0.500000D+00, maxT =  0.298000D+03

Socket closed before recv.
Socket 4 failed
```

1. Checkpoint

2. Failure

Fault Tolerance

Example

```
PlasComCM: iteration =      96, dt =  0.870094D-02, time =  0.835290D+00, cfl =  0.500000D+00, maxT =  0.298000D+03
PlasComCM: iteration =      97, dt =  0.870094D-02, time =  0.843991D+00, cfl =  0.500000D+00, maxT =  0.298000D+03
PlasComCM: iteration =      98, dt =  0.870094D-02, time =  0.852692D+00, cfl =  0.500000D+00, maxT =  0.298000D+03
PlasComCM: iteration =      99, dt =  0.870094D-02, time =  0.861393D+00, cfl =  0.500000D+00, maxT =  0.298000D+03
PlasComCM: iteration =     100, dt =  0.870094D-02, time =  0.870094D+00, cfl =  0.500000D+00, maxT =  0.298000D+03
[0] Checkpoint started
[0] Checkpoint finished in 0.455819 seconds
PlasComCM: iteration =     101, dt =  0.870094D-02, time =  0.878795D+00, cfl =  0.500000D+00, maxT =  0.298000D+03
PlasComCM: iteration =     102, dt =  0.870094D-02, time =  0.887496D+00, cfl =  0.500000D+00, maxT =  0.298000D+03
PlasComCM: iteration =     103, dt =  0.870094D-02, time =  0.896197D+00, cfl =  0.500000D+00, maxT =  0.298000D+03

Socket closed before recv.
Socket 4 failed

Charmrun finished launching new process in 1.153346 seconds
Charmrun says Processor 1 failed on Node 1
[1] Restarting after crash
[1] Restart finished in 0.458689 seconds at 0.463579.
PlasComCM: iteration =     101, dt =  0.870094D-02, time =  0.878795D+00, cfl =  0.500000D+00, maxT =  0.298000D+03
```

1. Checkpoint
2. Failure
3. Recover
4. Resume execution

Fault Tolerance

Example

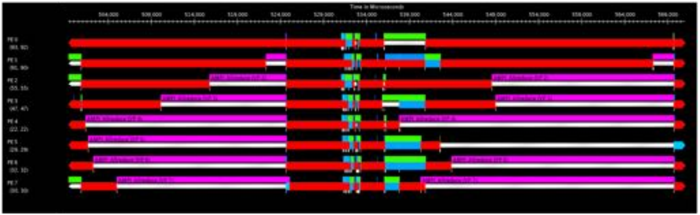
```
PlasComCM: iteration =      96, dt =  0.870094D-02, time =  0.835290D+00, cfl =  0.500000D+00, maxT =  0.298000D+03
PlasComCM: iteration =      97, dt =  0.870094D-02, time =  0.843991D+00, cfl =  0.500000D+00, maxT =  0.298000D+03
PlasComCM: iteration =      98, dt =  0.870094D-02, time =  0.852692D+00, cfl =  0.500000D+00, maxT =  0.298000D+03
PlasComCM: iteration =      99, dt =  0.870094D-02, time =  0.861393D+00, cfl =  0.500000D+00, maxT =  0.298000D+03
PlasComCM: iteration =     100, dt =  0.870094D-02, time =  0.870094D+00, cfl =  0.500000D+00, maxT =  0.298000D+03
[0] Checkpoint started
[0] Checkpoint finished in 0.455819 seconds
PlasComCM: iteration =     101, dt =  0.870094D-02, time =  0.878795D+00, cfl =  0.500000D+00, maxT =  0.298000D+03
PlasComCM: iteration =     102, dt =  0.870094D-02, time =  0.887496D+00, cfl =  0.500000D+00, maxT =  0.298000D+03
PlasComCM: iteration =     103, dt =  0.870094D-02, time =  0.896197D+00, cfl =  0.500000D+00, maxT =  0.298000D+03

Socket closed before recv.
Socket 4 failed

Charmrun finished launching new process in 1.153346 seconds
Charmrun says Processor 1 failed on Node 1
[1] Restarting after crash
[1] Restart finished in 0.458689 seconds at 0.463579.
PlasComCM: iteration =     101, dt =  0.870094D-02, time =  0.878795D+00, cfl =  0.500000D+00, maxT =  0.298000D+03

CharmLB> RefineLB: PE [0] starting at 69.353145
CharmLB> RefineLB: PE [0] #Objects migrating: 7
CharmLB> RefineLB: PE [0] finished at 69.355673 duration 0.002528 s

PlasComCM: iteration =     102, dt =  0.870094D-02, time =  0.887496D+00, cfl =  0.500000D+00, maxT =  0.298000D+03
PlasComCM: iteration =     103, dt =  0.870094D-02, time =  0.896197D+00, cfl =  0.500000D+00, maxT =  0.298000D+03
PlasComCM: iteration =     104, dt =  0.870094D-02, time =  0.904898D+00, cfl =  0.500000D+00, maxT =  0.298000D+03
PlasComCM: iteration =     105, dt =  0.870094D-02, time =  0.913599D+00, cfl =  0.500000D+00, maxT =  0.298000D+03
PlasComCM: iteration =     106, dt =  0.870094D-02, time =  0.922300D+00, cfl =  0.500000D+00, maxT =  0.298000D+03
PlasComCM: iteration =     107, dt =  0.870094D-02, time =  0.931001D+00, cfl =  0.500000D+00, maxT =  0.298000D+03
```

1. Checkpoint
2. Failure
3. Recover
4. Resume execution
5. Load balance

No Virtualization

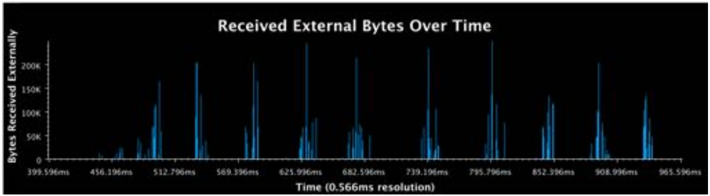
Example

Load imbalance appears during point-to-point messaging and in `MPI_Allreduce` each timestep



No Virtualization
Example

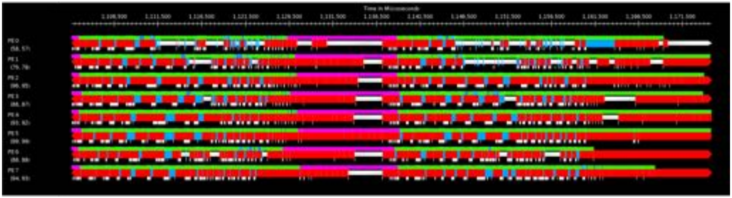
Communication/computation cycles mean the network is underutilized most of the time



Virtualization 8x

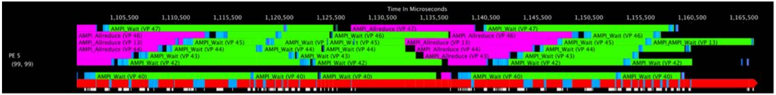
Example

Most of the idle time due to point-to-point messaging and `MPI_Allreduce` is now hidden by computation



Virtualization and Load Balancing
Example

The communication of each virtual rank is overlapped with the computation of others scheduled on the same core



Presentation

Parallel Programming Model
Ideal Features
Parallel Objects
Charm++

Adaptive Message Passing Interface
Introduction
Communication Optimizations
Migration

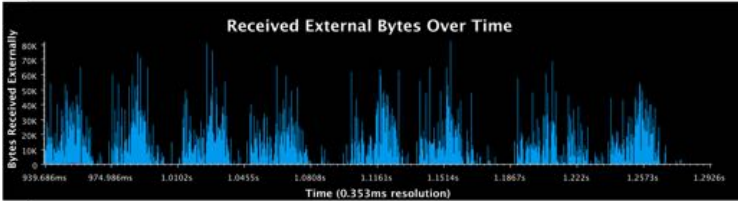
Global Variable Privatization

Conclusion

Virtualization 8x

Example

- Communication is spread over the whole timestep
- Peak network bandwidth used is reduced by 3x



AMPI Code

Compiling and running

- To compile an AMPI program:
  `charm/bin/ampicc pgm pgm.o`
- For migratability, link with: `-memory isomalloc`
- For LB strategies, link with: `-module CommonLBs`
- To run an AMPI job, specify the number of virtual processes (+vp)

```
./charmrun +p 1024 ./pgm
./charmrun +p 1024 ./pgm +vp 16384
./charmrun +p 1024 ./pgm +vp 16384 +balancer
RefineLB
```

Exercise

Compiling and running

Steps:

1. Build and run the LULESH mini-app on AMPI
2. Experiment with varying degrees of virtualization (ranks/core)
3. Add calls to create `MPI_Info` for LB and to `AMPI_Migrate()`
   ```
   MPI_Info_create(&info);
   MPI_Info_set(info, "ampi_load_balance",
   "sync");
   ```
4. Experiment with dynamic load balancing (frequency, strategy)

Get started:
AMPI is distributed with Charm++, and is already built in the pre-installed directory

Global Variable Privatization

Global Variable Privatization

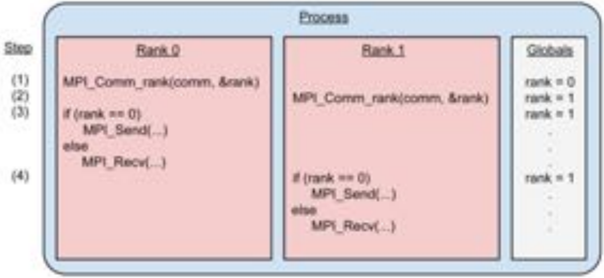
AMPI virtualizes the ranks of MPI_COMM_WORLD

- Ranks are implemented as user-level threads rather than OS processes
- Is this safe?

```
int rank, size;
int main (int argc, char *argv[]){

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    MPI_Barrier(MPI_COMM_WORLD);

    if(rank == 0) MPI_Send(..);
    else if (rank == 1) MPI_Recv(...);

    MPI_Finalize();
}
```

Global Variable Privatization

Unsafe code without modification



Global Variable Privatization

Recap

- AMPI virtualizes the ranks of `MPI_COMM_WORLD`
- It is unsafe to use a mutable global state:
  - Global state: global and static variables that can be modified
  - Mutable: written multiple times
- Rule:
  If global/static variables are written-once (or read-only) to same value across all ranks, they are safe. Otherwise, they are unsafe.

CeNAT

Global Variable Privatization

How to make an existing code safe for AMPI virtualization?

- For new codes, this is easy: avoid mutable global state
- For existing codes, how should we safely virtualize them?
  - Avoid using mutable global/static variables, or refactor to avoid them
  - Tag declarations of unsafe variable as `thread_local`
  - AMPI supports privatizing these to each rank at runtime:
    `ampicc -tlsglobals`
  - Other approaches possible, but either less portable or still under development

Global Variable Privatization

Manual encapsulation

- Method of refactoring an application to not use mutable global/static state
  - One-time refactoring with minor but pervasive changes, can be done by novice programmers
  - Can keep non-mutable variables at global scope
  - Results in a portable program that can be run with both MPI and AMPI
- Kinds of unsafe global/static variables:
  - C/C++: non-const globally scoped variables, static variables
  - Fortran: non-PARAMETER variables that are COMMON, SAVE, or MODULE

Presentation

Parallel Programming Model

Ideal Features
Parallel Objects
Charm++

Adaptive Message Passing Interface

Introduction
Communication Optimizations
Migration

Global Variable Privatization

Conclusion

Global Variable Privatization

Example

```
int rank, size;

int main (int argc, char *argv[]){
    initMPI(argc, argv);
    doWork();
}

int initMPI(int argc, char *argv[]){
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
}
```


Presentation

Parallel Programming Model

Ideal Features
Parallel Objects
Charm++

Adaptive Message Passing Interface

Introduction
Communication Optimizations
Migration

Global Variable Privatization

Conclusion

Global Variable Privatization

Solution

```
int size;

int main (int argc, char *argv[]){
    int rank;
    initMPI(argc, argv, &rank);
    doWork(rank);
}

int initMPI(int argc, char *argv[], int *rank){
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
}
```


Presentation

Parallel Programming Model

Ideal Features

Parallel Objects

Charm++

Adaptive Message Passing Interface

Introduction

Communication Optimizations

Migration

Global Variable Privatization

Conclusion

CeNAT

AMPI Privatization

Manual encapsulation

In applications with many global variables, it is often easier to define a new structure or derived type that contains all the mutable global variables

- Can do this hierarchically within each module first, then define one top-level structure that contains a type for each module
- Can safely ignore PARAMETERs or const global data
- Benefit: only pass one extra argument to each function that uses one or many global variables

AMPI Privatization

TLS globals

- Thread-Local Storage (TLS) provides per-thread copies of memory
  - C/C++11 provide standard support for `thread_local` attribute
  - Fortran has no standard support for TLS, though OpenMP has `threadprivate`
- AMPI provides support for privatizing TLS variables to its user-level threads
  - Only change necessary is tagging global variable declarations with TLS attribute
  - Runtime overhead is switching the TLS pointer at each ULT context switch
  - Currently requires gcc/gfortran and Linux

Other approaches for AMPI Privatization

Automatic ELF Global Offset Table swapping

Benefits:

- Full automation, no developer effort
- Already implemented in AMPI

Limitations:

- Requires ELF binary format
- Requires disabling linker optimizations in ld v2.23+
- It does not handle static variables
- Runtime overhead proportional to the number of global variables

Presentation

Parallel Programming Model

Ideal Features
Parallel Objects
Charm++

Adaptive Message Passing Interface

Introduction
Communication Optimizations
Migration

Global Variable Privatization

Conclusion

Other approaches for AMPI Privatization

icc mpc-privatize

Benefits:

- Full automation, no developer effort

Limitations:

- Requires icc, or patched version of gcc
- Not yet supported by AMPI

Other approaches for AMPI Privatization

Process-in-Process library

Benefits:

- Full automation, no developer effort, no compiler support needed

Limitations:

- Requires patched version of glibc
- Requires dynamic linking of application and libraries with globals
- Not yet implemented in AMPI

AMPI Privatization

Fortran support

Additional concerns for AMPI-izing Fortran codes:

- Fortran program main must be renamed `subroutine MPI_Main`
- Fortran command line argument parsing must be done with AMPI extension routines similar to Fortran2003 standard routines
- Implicit SAVE variables are static and can be hard to identify
- Use of AUTOMATIC arrays can bloat the ULT stack size
- Must use OpenMP `threadprivate` attribute for TLS declarations

CeNAT

Exercise

Manual privatization with single data structure

```
int myrank;
double xyz[100];

void subA();
int main(int argc, char** argv){
  int i;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  for(i=0;i<100;i++)
    xyz[i] = i + myrank;
  subA();
  MPI_Finalize();
}

void subA(){
  int i;
  for(i=0;i<100;i++)
    xyz[i] = xyz[i] + 1.0;
}
```

Exercise

Solution

```
int main(int argc, char** argv){
  int i,ierr;
  struct shareddata *c;
  MPI_Init(&argc, &argv);
  c = (struct shareddata*)malloc(sizeof(struct
      shareddata));
  MPI_Comm_rank(MPI_COMM_WORLD, &(c->myrank));
  for(i=0;i<100;i++)
    c->xyz[i] = i + c->myrank;
  subA(c);
  MPI_Finalize();
}

void subA(struct shareddata *c){
  int i;
  for(i=0;i<100;i++)
    c->xyz[i] = c->xyz[i] + 1.0;
}
```

Exercise

FORTRAN code

Goal: Learn how to privatize an existing (Fortran) MPI code using two privatization techniques:

- Manual encapsulation
- TLS globals

MiniGhost: a mini-application from the Mantevo suite:

- Fortran90 MPI stencil code
- Contains multiple global and static variables, across multiple modules
- Mix of read-only, and written-once, and mutable variables

Exercise
FORTRAN code

1. Identify the global variables in MiniGhost
2. Only declared in 2 files: `MG_CONSTANTS.F` and `MG_OPTIONS.F`
3. Classify them as mutable, written-once, or read-only
4. Privatize mutable global variables using OpenMP `threadprivate`:
5. Compile with `ampif90 -tlsglobals` option
6. Run with different degrees of virtualization

```
INTEGER :: VARIABLE
!$omp threadprivate(VARIABLE)
```

AMPI

More resources

1. AMPI Tutorial: https://charm.readthedocs.io/en/latest/ampi/manual.html
2. AMPI Research Papers: https://charm.cs.illinois.edu/papers
3. AMPI applications: `git clone https://charm.cs.illinois.edu/gerrit/benchmarks/ampi-benchmarks`

Acknowledgements

- Dr. Carla Osthoff for invitation
- Sam White at University of Illinois for helping with AMPI material and questions
- AMPI Tutorial by Parallel Programming Lab of University of Illinois at Urbana-Champaign

Presentation

Parallel Programming Model

Ideal Features
Parallel Objects
Charm++

Adaptive Message Passing Interface

Introduction
Communication Optimizations
Migration

Global Variable Privatization

Conclusion

Concluding Remarks

AMPI provides the dynamic RTS support of Charm++ with the familiar API of MPI

- Overdecomposition
- Communication optimizations
- Dynamic load balancing
- Automatic fault tolerance
- Checkpoint/restart
- OpenMP runtime integration

CeNAT
Centro Nacional de Alta Tecnología

Presentation

Parallel Programming Model
Ideal Features
Parallel Objects
Charm++

Adaptive Message Passing Interface
Introduction
Communication Optimizations
Migration

Global Variable Privatization

Conclusion

CeNAT