Charm++: A Portable Concurrent Object Oriented System Based on C++

Laxmikant V. Kale **Sanjeev Krishnan**

University of Illinois

1993

Speaker: Ariadni-Karolina Alexiou

Motivation

- Yet another concurrent programming system
- Why not just use **threads**?

Motivation

- Yet another concurrent programming system

- Why not just use **threads**?
  - OS-dependent

Motivation

- Yet another concurrent programming system
- Why not just use **threads**?
  - OS-dependent
  - Low level

Motivation

- Yet another concurrent programming system
- Why not just use **threads**?
  - OS-dependent
  - Low level
  - Difficult communication

Motivation

- Yet another concurrent programming system

- Why not just use **threads**?
  - OS-dependent
  - Low level
  - Difficult communication
  - Not taking advantage of special architectures

Motivation

- So we'd want a system that:
  - Is portable
  - Provides high level abstractions
  - Provides flexibility in communication
  - Can take advantage of the architecture of special parallel machines
  - Performs well

What is Charm++?

- C++ extension & runtime system
  - specifically aimed for highly scalable parallel applications
  - portable to many types of parallel machines (late 80s → burst of parallel machine technology)

Philosophy : '*Aid the programmer in the design of parallel algorithms (language), leave the resource management to the system (runtime)***'**

Features

- How is **Charm++** different? (from similar work from the 90s)
  - Supports **both** message passing AND shared memory
  - Optimizations for **performance** (load balancing, message scheduling)
  - Object oriented paradigm → **modularity, reusability**
  - **Data abstractions** specifically aimed at concurrency → programmer productivity

How does Charm++ work?

- **C++ Extensions**
  - New type of parallel object → **chare**
  - Message objects
  - Shared objects → basically abstractions of commonly used patterns in parallelism (shared counters etc)

- **Restrictions**
  - All of C++ functionality as we know it
  - Some restrictions on **global variables** → replaced by shared objects

Chares

- **Chare : The parallel building block**
  - A Class that is defined as 'chare'
  - Chare object created→ **process** spawned by the runtime
  - 'mailboxes' to receive messages (**Entry Points**)
    - Special functions with the expected message type as the argument

- **Capabilities:**
  - sends **messages** to another chare's Entry Points
  - receives messages in the EntryPoints
  - **asynchronous** creation/message passing → performance

Message Objects

- **Message:** basically a C-struct which is labeled as 'message'
- Sent **asynchronously**

Shared Objects

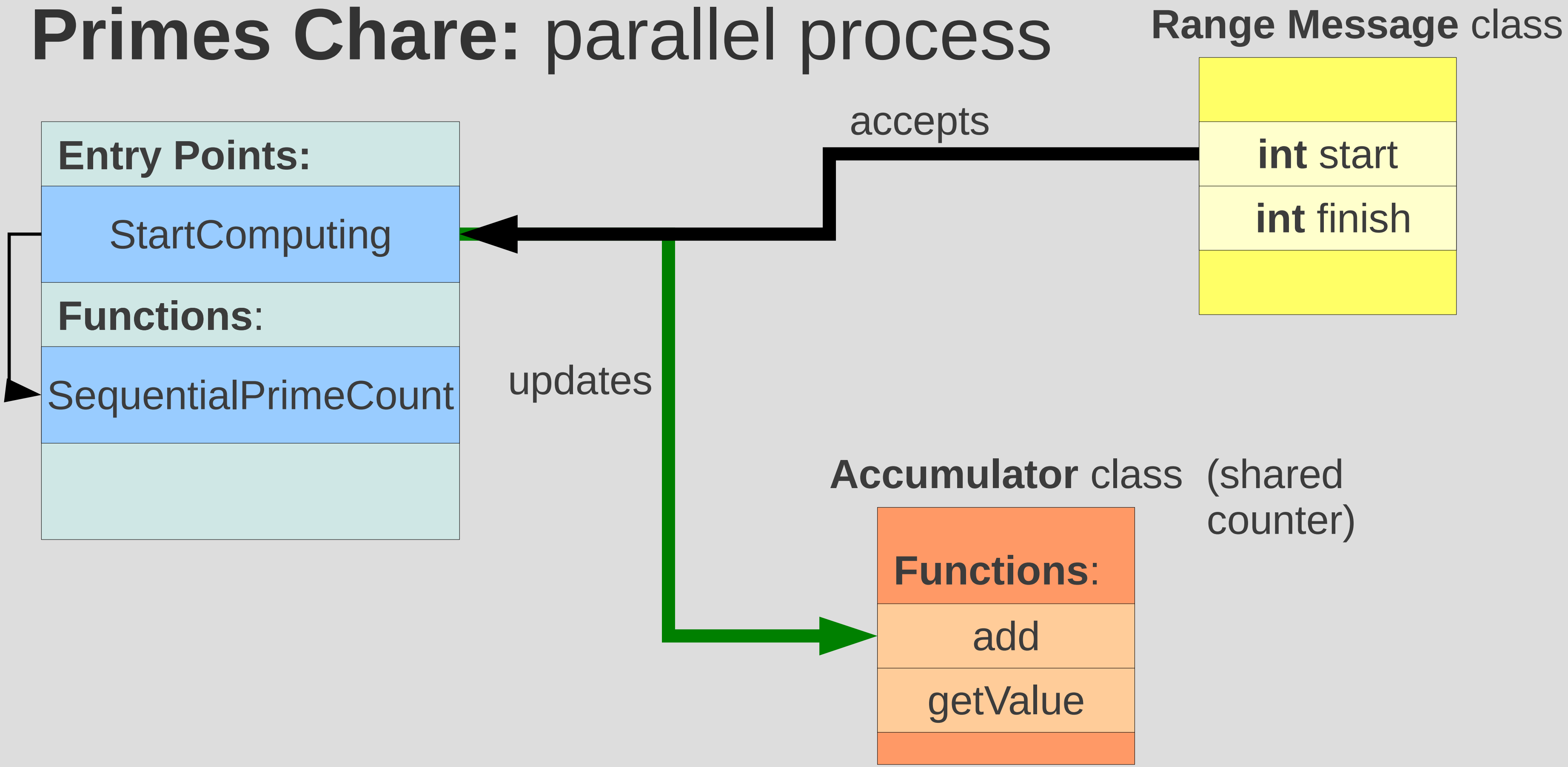
- Basically **abstractions** of patterns commonly used in parallel applications
- Read-only objects
- Write-once objects
- **Accumulators (shared counters)**
- Monotonic objects (for branch-and-bound)

Example Program: Primes

- We want to count **prime** numbers from 0 to N
- We will recursively divide the range in half until **range < 100**
- When the range is small → sequential computation
- Code made more abstract for readability

Example Program: Our Objects

Primes Chare: parallel process



Range Message class

int start

int finish

accepts

Entry Points:

StartComputing

Functions:

SequentialPrimeCount

updates

Accumulator class (shared counter)

Functions:

add

getValue

Example Program: Primes (main)

```
Accumulator * total;
   //special shared object,
   visible to all chares

main(){


total = new
   Accumulator(0);

newChare(PrimesChare,
   StartComputing, new
   RangeMessage(0,N));
}
```

PrimesChare → class of created chare

StartComputing → entry point function called upon creation of chare

RangeMessage → the message sent to the StartComputing EP

Example Program: The PrimesChare class

```
chare class PrimesChare
  entryPoint:
  StartComputing(RangeMessage m)

  if(m.finish – m.start > 100)  //if range is > 100, split
       int middle = m.finish – m.start /2; //the work to
                                           //two processes
       newChare(PrimesChare, StartComputing,
                new RangeMessage(m.start,middle));

       newChare(PrimesChare, StartComputing,
             new RangeMessage(middle+1,m.finish)); }

  else //else, do the work
       int count = sequentialPrimeCount
                    (m.start,m.finish);
       total->add(count);
```

Example Program: The PrimesChare class

```
chare class PrimesChare
  entryPoint:
  StartComputing(RangeMessage m)

  if(m.finish – m.start > 100)  //if range is > 100, split
        int middle = m.finish – m.start /2; //the work to
                                             //two processes
        newChare(PrimesChare, StartComputing,
                new RangeMessage(m.start,middle));

        newChare(PrimesChare, StartComputing,
             new RangeMessage(middle+1,m.finish)); }

  else //else, do the work
        int count = sequentialPrimeCount
                         (m.start,m.finish);
        total->add(count);
```

asynchronous

Example Program: The PrimesChare class

```
chare class PrimesChare
  entryPoint:
  StartComputing(RangeMessage m)

  if(m.finish – m.start > 100)  //if range is > 100, split
        int middle = m.finish – m.start /2; //the work to
                                            //two processes
        newChare(PrimesChare, StartComputing,
                new RangeMessage(m.start,middle));

        newChare(PrimesChare, StartComputing,
             new RangeMessage(middle+1,m.finish)); }

  else //else, do the work
        int count = sequentialPrimeCount
                        (m.start,m.finish);
        total->add(count);
```

asynchronous

shared (global)

Load Balancing Strategies

- Random
- Central Manager
- Adaptive
- Token-based
- Greatly enhanced over time

Performance Results

- **nCUBE/2** (Intel, late 80s)

Speed Up

Processors	TSP	Primes	Jacobi
1	1	1	1
16	12	8	9
64	21.7	31	35
256	21.8	146	130

Performance Results

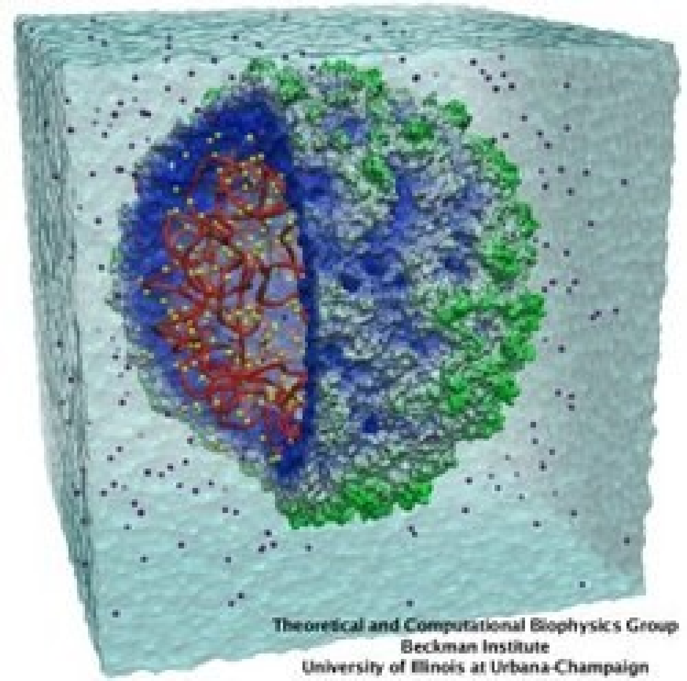
- **Sequent Symmetry** (Intel, 1987)
  - shared memory
  - up to 30 processors (66 MHz)

Speed Up

Processors	**TSP**	**Primes**	**Jacobi**
1	1	1	1
4	4	4	3.7
9	8.7	8.9	7.5
16	15.1	15.8	12

Applications

- OpenAtom (quantum chemistry modeling)
- NAMD (molecular dynamics simulation)



Applications

- ChaNGa
  - collisionless N-body simulation
  - hydrodynamics
  - Charm++ chosen for
    - support for massive parallelism
    - dynamic load balancing schemes

  - Scales to up to **20,000** processors on an IBM Bluegene/L

Evolution

- Charm ++ v 6.4.0 released this March
    - Syntax has been refined
    - Multiple value parameters as entry point arguments
    - Vastly enhanced load balancing
    - More platforms supported
    - Talks, tutorials, active research

Conclusions

- **Charm++** is a system suited for massively parallel applications
  - Very active for almost two decades
  - Has scientific applications
  - Portable, highly optimized and modular

Conclusions

- Would I use it?
  - Overhead/Learning Curve (-)
  - A language I already know (+)
  - Depends on the task

- Questions/Criticism
  - Results are compared to the sequential version
  - How exactly are the shared objects managed?
  - Few implementation details

Extra Example Program: Primes (main)

```
Accumulator * total; //special shared object, visible to
   all chares

main(){

int start = 0;
int finish = N;
total = new Accumulator(0);

newChare(PrimesChare, StartComputing,
   Message(start,finish));
}

Quiescence(){ //executed when all chares have finished
  int result = total->getValue();
  print(result);
}
```

Extra Example Program: Primes (main)

```
Accumulator * total;
   //special shared object,
   visible to all chares

main(){

int start = 0;
int finish = N;
total = new
   Accumulator(0);

newChare(PrimesChare,
   StartComputing,
   Message(start,finish));
}
```

PrimesChare → class of created chare

StartComputing → entry point function called upon creation of chare

Message → the message sent to the StartComputing EP

Extra: Parallel Machines from the Past

- **Ncube/2**
  - Non shared memory machine
  - Processors → vertices of hypercube
  - Connections between processors → edges of hypercube
- **Sequent Symmetry** (Intel, 1987)
  - shared memory
  - up to 30 processors (66 MHz)

Extra: Adaptive MPI

- Adaptive MPI (2001)
  - Implementation of the MPI standard on top of Charm++
  - MPI takes advantage of the Charm++ runtime