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

L. V. Kale
Sanjeev Krishnan

Department of Computer Science
University of Illinois, Urbana-Champaign
Parallel Computing

1. Computationally demanding applications exist:
   - grand challenge problems
   - commercial applications

2. Parallel computing can make these problems tractable

3. Large scale commercial parallel computers are available – CM-5, Paragon, nCUBE/2, Cray T3D, KSR-1, SP-1.
A Hurdle

1. Programming parallel machines is difficult
2. Scheduling, load balancing, synchronization, communication latency
3. Portability
4. A new dimension to the complexity of programs
Object Orientation

A way of organizing and thinking about programming

1. Abstraction and encapsulation
2. Modularity and clean interfaces
3. Inheritance hierarchies
4. Software Reuse and libraries
5. Polymorphism
Can Object Orientation help Parallel Programming?

1. Fundamental concepts overlap:
   Processes and Objects
   - State and persistence
   - Interaction by messages

2. Abstraction controls complexity

3. Modularity helps reuse for different data distributions

*Combine the benefits of two powerful technologies*
The CHARM Parallel Programming

philosophy

1. Portability

2. Latency tolerance
   - Simple message passing wastes resources
   - Message driven execution overlaps computation and communication

3. Support dynamic creation of work:
   dynamic load balancing

4. Provide specific abstractions for sharing information

5. Support irregular as well as regular, data-parallel computations

CHARM: A C based parallel programming language
CHARM++: A high level view

Objects

- Sequential objects
- Chares (concurrent objects)
- Branched chares (a form of replicated objects)
- Shared objects
- Communication objects
Sequential objects are different from parallel objects

1. Programmers need to know how much an action costs (simple local call v/s expensive remote call)

2. Asynchronous, split-phase remote calls: different from function calls

3. Better algorithm design: parallel objects coordinate sequential objects

4. Reuse code for existing sequential classes

5. Better performance by explicit grain size control
message MessageName {
    .... data members ....
}

char class ShareName {

    .... data and function members ....

entry:

    void EntryPoint1(MessageType1 *Pointer) {
        .... C++ code block ....
    }

    void EntryPoint2(MessageType2 *Pointer) {
        .... C++ code block ....
    }

} ;
branched_chare class CharName {
    .... data and function members ....

    entry:
    voidEntryPointName(MessageType *Pointer)
    {
        .... C++ code block ....
    }
}

1. One branch on every processor

2. Public members can be accessed on the local processor by
   LocalBranch(CharHandle)->Function()
CHARM++ Language: System calls

1. Creating objects:
   - `new_char(ChareName, EntryPoint, Message)`
   - `new_branched_char(ChareName, EP, Message)`
   - `new_message(MessageType);`

2. Sending messages:
   - `to char_chares: ChareHandle=>EntryPoint(Message)`
   - `to branched char_chares:`
     - `ChareHandle[PE]=>EntryPoint(Message)`
     - `ChareHandle[ALL]=>EntryPoint(Message)`

3. Other calls for termination, I/O, timing.
Shared Objects: Data Sharing

1. Messages are too low level and generic

2. Communication overheads can be optimized if the pattern of data sharing is known

3. Need abstract template types for sharing information in specific modes
Shared Objects: Abstract Types

1. Read Only: initialize at beginning, read efficiently

2. Write Once: initialize anytime, read efficiently

3. Accumulator: efficient update, read once (e.g. global sum)

4. Monotonic: many reads and updates, need monotonicity

5. Distributed Tables:
   - each entry has a key and data field
   - asynchronous Insert, Delete and Find operations
Modularity

1. Separate compilation, libraries

2. Function pointers cannot be passed across address spaces
   - function reference indices

3. Modules must exchange data in a fully distributed manner

4. Modules must not assume data distribution
   - branched shares, distributed tables
Load balancing

1. Necessary to support irregular, dynamic creation of work

2. User selectable at compile time from many strategies
   - Random
   - Adaptive Contracting Within Neighborhood
   - Central Manager
   - Token based
Other Features

1. Many user selectable scheduling strategies

2. Prioritized Execution
   - Integer priority
   - Bit vector (unbounded) priority

3. Conditional Message Packing
   - Complex data structures having pointers must be packed before sending them across processors
   - System does packing *only* if message crosses address space
An Example: Primes
extern int seqPrimes(int low, int high);
const int LENGTH = 10000;
message MsgAccCount { int data; };
message RangeMsg {
    int Low, High;
};
class AccCount : public Accumulator {
    MsgAccCount *msg;
public:
    AccCount(MsgAccCount *initmsg) {
        msg = (MsgAccCount *)new message(MsgAccCount)
        msg->data = initmsg->data;
    }
    void Accumulate (int x) {
        msg->data += x;
    }
    void Combine (MsgAccCount *y) {
        msg->data += y->data;
    }
};
AccCount *total;
char class main {
  entry:
    main()
    {
      int Limit;
      CPrintf("Enter upper limit of range: ");
      CScanf("%d", &Limit);
      AccInitMsg *acc_msg = new_message(AccInitMsg);
      acc_msg->data = 0;
      total = new AccCount(acc_msg);
      RangeMsg *msg = new_message(RangeMsg);
      msg->Low = 1; msg->High = Limit;
      new_char(char(PrimesChar, Goal, msg));
    }
    Quiescence()
    {
      main handle *myid = MyChareHandle();
      total->CollectAccValue(PrintResult, myid);
    }
    PrintResult(MsgAccCount * result)
    {
      CPrintf("The total is:%d.", result->data);
      CharmExit();
    }
};
class PrimesChar {  
  entry:  
    Goal(RangeMsg * msg1)  
    { int L = msg1->Low;  
      int H = msg1->High;  
      if ((H-L+1) > LENGTH)  
      { int Mid = L + (H-L+1)/2;  
        RangeMsg *msg2 = new_message( RangeMsg);  
        msg2->Low = Mid; msg2->High = H;  
        msg1->High = Mid-1;  
        new_char(PrimesChar, Goal, msg1);  
        new_char(PrimesChar, Goal, msg2);  
      }  
      else {  
        int count = seqPrimes(L,H);  
        delete_message(msg1);  
        total->Accumulate(count);  
      }  
    CharExit();  
  }  
};
Implementation

1. Translator + Charm runtime system

2. Charm runtime ported to CM5, Paragon, nCUBE/2, networks of workstations, iPSC/860, Sequent, Multimax, uniprocessor
   - Others in future
   - See references for details

3. Translator produces C++ code and runtime interface code

4. Remote function call requires encoding function names into ids which can be sent across processors

5. Complicated by separate compilation requirement

6. Solved using mapping generated at run time
Current Status

1. First version completed in February

2. Second version now complete
   - full C++ parser
   - error recovery
   - some syntax changes

3. Currently running on CM5, nCUBE/2, networks of workstations
Performance : Sequent Symmetry
Performance: nCUBE/2
Processors | Speedup
---|---
1 64 128 256
128
64
256

Jacobi

Processors | Speedup
---|---
1 64 128 256
128
64
256

TSP

Processors | Speedup
---|---
1 64 128 256
128
64
256

Primes
Related work

- Actors (Agha)
- CST (Dally)
- Concurrent Aggregates, Concert (Chien)
- ABCL (Yonezawa)
- pC++ (Gannon)
- CC++ (Chandy and Kesselman)
- Mentat (Grimshaw)
- ESP-C++ (from MCC)
- Amber (Chase et al)
- Many others
Distinguishing features of Charm++

1. Message driven execution
2. Information sharing abstractions
3. Dynamic load balancing
4. Support for irregular AND data-parallel applications
5. Clean separation: sequential and parallel objects
6. Runs on many commercial parallel machines
7. Does not require threads package
Future work

1. Further optimize runtime system

2. Integrate Charm and Charm++ programs

3. Combine with Dagger (a visual language for specifying dependences between messages and computations)

4. Libraries

5. Applications