Table of Contents

- Introduction to Parallelism
- Introduction to Programming Models
- Shared Memory Programming
- Message Passing Programming
  - MPI
  - Charm++
- Shared Memory Models
- PGAS Languages
- Other Programming Models

Charm++

- C++ library for dynamic multithreaded applications
- Developed since 1993
  - Prequel Chare Kernel developed since 1988
- Parallel Programming Laboratory at University of Illinois at Urbana-Champaign
- Prof. Laxmikant V. Kale

Material from: http://charm.cs.uiuc.edu/

Charm++ – Programming Model

Execution Model

- Implicit parallelism
  - Parallelism expressed at the task level (Chare)
  - User unaware of concurrency
- Explicit communication
  - Exclusively through asynchronous RMI (on Chare entry methods)
  - User responsible for implementing packing/unpacking methods
- MPMD
- Message-driven execution
- Dynamic parallelism
  - Every task is a thread
  - Load-balancing with task migration
- Object Consistency model

Charm++ – Programming Model

Productivity

- Charmdebug graphical parallel debugger
- Graphical load balance monitor
- Relatively high level of abstraction

Performance

- Split-phase communication tolerates latency
- Static and dynamic load-balancing
- Processor virtualization

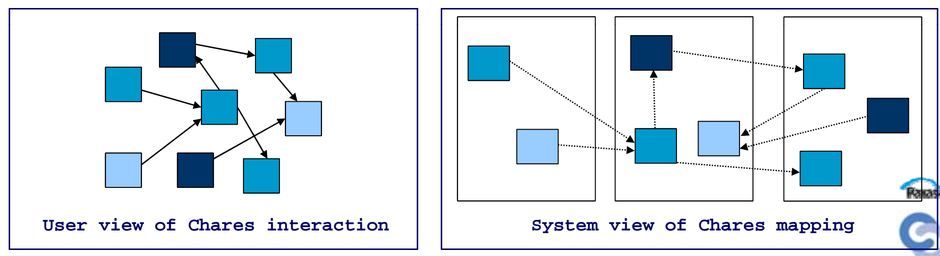
Portability

- Library implemented on top of MPI

Charm++ – Virtualization

Object-based decomposition

- Divide the computation into a large number of pieces
  - Independent of the number of processors
  - Preferably significantly larger than the number of processors
- Let the system map objects to processors



Charm++ – Chares

- Dynamically created on any available processor
- Can be accessed from other processors
  - Chare_ID instead of Thread_ID (virtualization)
- Send messages to each other asynchronously
- Contain entry methods that can be invoked from other Chares

Charm++ – Chares

- User only required to think of the interaction between chares
- Message-driven execution
  - New Chares are only created as "Seed messages"
  - Construction happens when a first message reaches the new Chare

Charm++ – "Hello World"

HelloWorld.ci

```
mainmodule hello {
  mainchare mymain {
    entry mymain (CkArgMsg *m);
  };
};
```

Charmc

Generates:

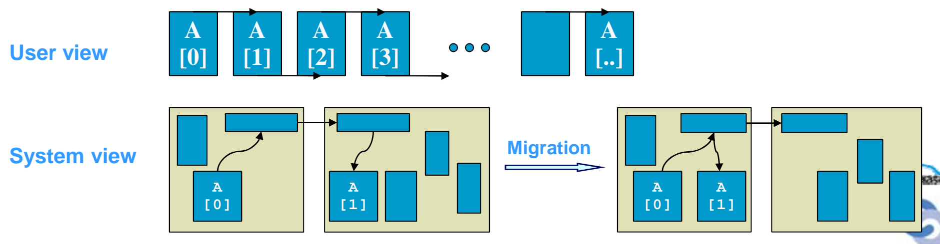
- HelloWorld.decl.h
- HelloWorld.def.h

HelloWorld.C

```
#include "HelloWorld.decl.h"

class mymain : public Chare {
public:

  mymain(CkArgMsg *m) {
    ckout << "Hello world !" << endl;
    CkExit();
  }

};
```

Charm++ – Chare Arrays

- Array of Chare objects
  - Each Chare communicates with the next one
  - More structured view than individual chares
- Single global name for the collection
- Members addressed by index
- Mapping to processors handled by the system



Charm++ – Dynamic Load-Balancing

- Object (Chare) migration
  - Array Chares can migrate from one processor to another
  - Migration creates a new object on the destination processor and destroys the original
  - Objects must define pack/unpack (PUP) methods
- Initial load-balancing
  - New Chares created on least loaded processors

Charm++ – Dynamic Load-Balancing

- Centralized load-balancing
  - High-quality balancing with global information
  - High communication cost and latency
- Distributed load-balancing
  - Same principle in small neighborhoods
  - Lower communication cost
  - Global load-imbalance may not be addressed

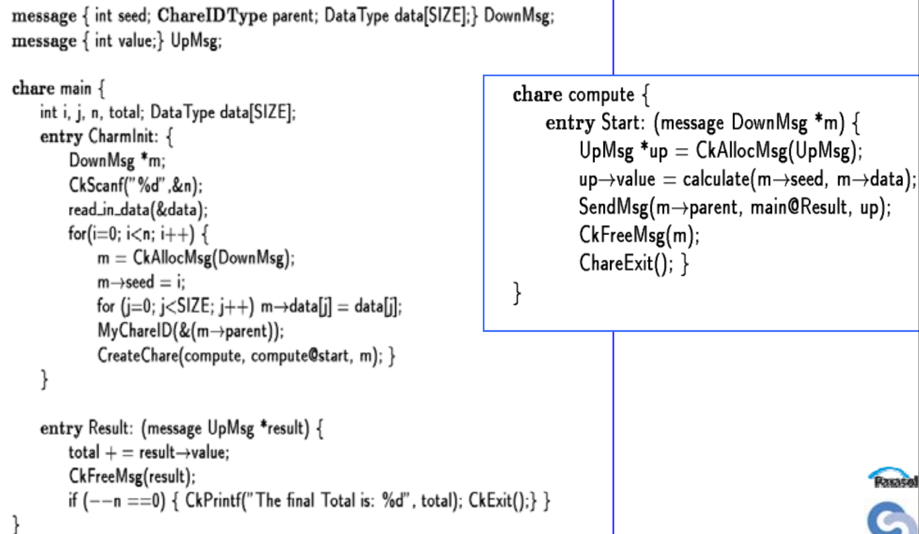
Charm++ – Split-phase Communication

- Asynchronous communication
  - Sender does not block or wait for a return
  - Sender provides callback handler that will process any return value
- Efficient for tolerating latency
  - No explicit waiting for data
  - No stalls with sufficient parallelism

```
chare Client {
    entry MakeRequest: (message MSG1 *m) {
        MyChareID(&(m->reply_id));
        m->ep = ProcessReply;
        SendMsg(Request, m, &chareB);
    }

    entry ProcessReply: (message MSG2 *m) {
        CkPrintf("%s\n", m->data);
    }
}
```

```
chare Server {
    entry Request: (message MSG1 *m) {
        MSG2 *m2 = (MSG2 *) CkAllocMsg(MSG2);
        m2->data = data;
        SendMsg(m->ep, m2, &(m->reply_id));
    }
}
```

Charm++



References

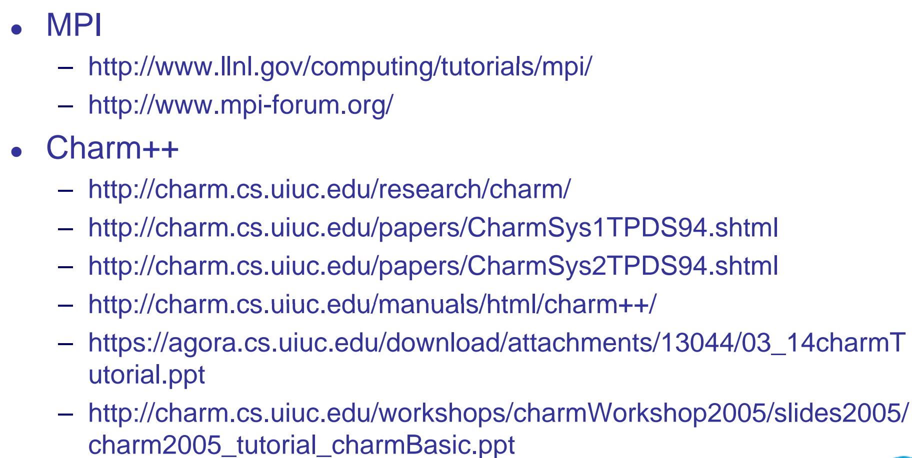


Table of Contents

- Introduction to Parallelism
- Introduction to Programming Models
- Shared Memory Programming
- Message Passing Programming
- Shared Memory Models
  - Cilk
  - TBB
  - HPF
  - Chapel
  - Fortress
  - Stapl
- PGAS Languages
- Other Programming Models

Cilk

- Language for dynamic multithreaded applications
- Superset of C
- Developed since 1994
- Supercomputing Technologies Group at MIT Laboratory for Computer Science
- Prof. Charles E. Leiserson

Materials from Charles Leiserson, “Multithreaded Programming in Cilk”, http://supertech.csail.mit.edu/cilk/ . Used with permission.

Cilk extends C

- C elision
  - Removal of Cilk keywords produces valid sequential C program
  - A valid implementation of the semantics of a Cilk program

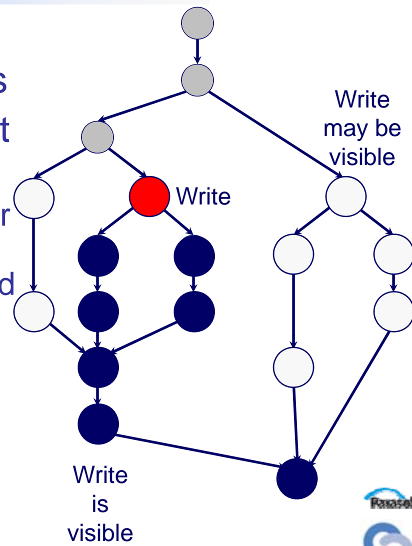
```
cilk int fib (int n) {
  if (n < 2)
    return n;
  else {
    int x, y;
    x = spawn fib (n-1);
    y = spawn fib (n-2);
    sync;
    return (x+y);
  }
}
```

The Cilk Model

- Execution Model
  - DAG consistency model
  - Explicit Parallelism
  - Explicit Synchronization
- Productivity
  - Simple extension of an existing language
  - No details of machine available to application
  - Low level of abstraction
  - No component reuse or language expansion possible
  - Debug and tune using standard tools

DAG consistency

- Vertices are tasks
- Edges are data dependencies
- Read operation can see result of write operation if:
  - There is a serial execution order of the tasks consistent with the DAG where the read is executed after the write
- Successors of a task guaranteed to see write
- Other tasks may or may not see the write



The Cilk Model

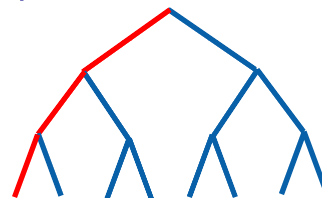
- Performance
  - Developer easily generates high degree of parallelism
  - Work stealing runtime scheduler provides load balance
- Portability
  - Source-to-source compiler provided
  - Runtime system must be ported to new platforms
  - Applications completely unaware of underlying system

Cilk Thread Management

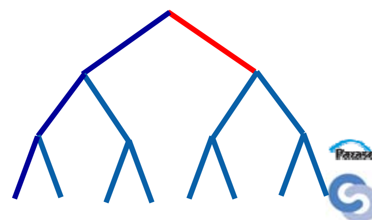
- Application completely unaware of threads
  - Work split into Cilk threads
    - Cilk thread is a task assigned to a processor
    - Tasks scheduled to run on processors by runtime system
    - “Spawn” of Cilk thread is 3-4 times more expensive than C function call
  - Runtime system employs work stealing scheduler

Work Stealing Task Scheduler

- Each processor maintains a deque of tasks
  - Used as a stack
  - Small space usage
  - Excellent cache reuse



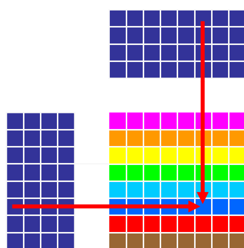
- Processor steals when nothing remains in deque
  - Chooses random victim
  - Treats victim deque as queue
  - Task stolen is usually large



Cilk Synchronization

- Cilk_fence()
  - All memory operations of a processor are committed before next instruction is executed.
- Cilk_lockvar variables provide mutual exclusion
  - Cilk_lock attempts to lock and blocks if unsuccessful
  - Cilk_unlock releases lock
  - Locks must be initialized by calling Cilk_lock_init()

Cilk Matrix Multiply



```
cilk void work(*A, *B, *C, i, L, N) {
  for(int j=0; j<N; ++j) {
    for(int k=0; k<L; ++k) {
      C[i][j] +=
          A[i][k]*B[k][j];
    }
  }
}

void matmul(*A, *B, *C, M, L, N) {
  for(int i=0; i<M; ++i) {
    spawn work(A, B, C, i, L, N);
  }
  sync;
}
```

Cilk Recursive Matrix Multiply

Divide and conquer —

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$= \begin{bmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{bmatrix} + \begin{bmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{bmatrix}$$

8 multiplications of $(n/2) \times (n/2)$ matrices.
1 addition of $n \times n$ matrices.

Matrix Multiply in Pseudo-Cilk

```
cilk void Mult(*C, *A, *B, n) {
  float *T = Cilk_alloca(n*n*sizeof(float));
  ⟨base case & partition matrices⟩
  spawn Mult(C11,A11,B11,n/2);
  spawn Mult(C12,A11,B12,n/2);
  spawn Mult(C22,A21,B12,n/2);
  spawn Mult(C21,A21,B11,n/2);
  spawn Mult(T11,A12,B21,n/2);
  spawn Mult(T12,A12,B22,n/2);
  spawn Mult(T22,A22,B22,n/2);
  spawn Mult(T21,A22,B21,n/2);
  sync;
  spawn Add(C,T,n);
  sync;
  return;
}
```

$C = A \times B$

Absence of type declarations.

Matrix Multiply in Pseudo-Cilk

```
cilk void Mult(*C, *A, *B, n) {
  float *T = Cilk_alloca(n*n*sizeof(float));
  h base ca  & partition matrices i
  spawn Mu t(C11,A11,B11,n/2);
  spawn Mult C12,A11,B12,n/2);
  spawn Mult(  2,A21,B12,n/2);
  spawn Mult(C2  A21,B11,n/2);
  spawn Mult(T11,  2,B21,n/2);
  spawn Mult(T12,A  B22,n/2);
  spawn Mult(T22,A2   2,n/2);
  spawn Mult(T21,A22,   n/2);
  sync;
  spawn Add(C,T,n);
  sync;
  return;
}
```

$C = A \cdot B$

Coarsen base cases for efficiency.

Matrix Multiply in Pseudo-Cilk

```
cilk void Mult(*C, *A, *B, n) {
  float *T = Cilk_alloca(n*n*  zeof(float));
  h base case & partition  atrices i
  spawn Mult(C11,A11 B11,n/2
  spawn Mult(C12,A11,  2,n/2
  spawn Mult(C22,A21,B  n/2
  spawn Mult(C21,A21,B1  /2
  spawn Mult(T11,A12,B21,
  spawn Mult(T12,A12,B22,
  spawn Mult(T22,A22,B22,n/
  spawn Mult(T21,A22,B21,n/2
  sync;
  spawn Add(C,T,n);
  sync;
  return;
}
```

$C = A \cdot B$

Also need a row-size argument for array indexing.

Submatrices are produced by pointer calculation, not copying of elements.

Matrix Multiply in Pseudo-Cilk

```
cilk void Mult(*C, *A, *B, n) {
  float *T = Cilk_alloca(n*n*sizeof(float));
  h base case & partition matrices i
  spawn Mult(C11,A11,B11,n/2);
  spawn Mult(C12,A11,B12,n/2);
  spawn Mult(C22,A21,B12,n/2);
  spawn Mult(C21,A21,B11,n/2);
  spawn Mult(T11,A12,B21,n/2);
  spawn Mult(T12,A12,B22,n/2);
  spawn Mult(T22,A22,B22,n/2);
  spawn Mult(T21,A22,B21,n/2);
  sync;
  spawn Add(C,T,n);
  sync;
  return;
}
```

$C = A \times B$

```
cilk void Add(*C, *T, n) {
  h base case & partition matrices i
  spawn Add(C11,T11,n/2);
  spawn Add(C12,T12,n/2);
  spawn Add(C21,T21,n/2);
  spawn Add(C22,T22,n/2);
  sync;
  return;
}
```

$C = C + T$

Table of Contents

- Introduction to Parallelism
- Introduction to Programming Models
- Shared Memory Programming
- Message Passing Programming
- Shared Memory Models
  - Cilk
  - TBB
  - HPF
  - Chapel
  - Fortress
  - Stapl
- PGAS Languages
- Other Programming Models

Threading Building Blocks

- C++ library for parallel programming
- STL-like interface for library components
  - **Algorithms** accept **Ranges** that provide access to **Containers**
- Initial release by Intel in August 2006
- Strongly influenced by Cilk, STAPL, and others

Intel® Threading Building Blocks

Generic Parallel Algorithms
parallel_for
parallel_while
parallel_reduce
pipeline
parallel_sort
parallel_scan

Concurrent Containers
concurrent_hash_map
concurrent_queue
concurrent_vector

Task Scheduler

Low-Level Synchronization Primitives
atomic
spin_mutex
queuing_mutex
reader_writer_mutex
mutex

Memory Allocation
cache_aligned_allocator

Timing
tick_count

The TBB Model

- Execution Model
  - Implicit parallelism
  - Mixed synchronization
    - Locks provided for mutual exclusion
    - Containers provide safe concurrent access
- Productivity
  - Library for an existing language
    - Provides components for reuse
  - Few details of machine available to developer
  - Higher level of abstraction
  - Timing class provided in library for manual tuning
  - Designed to be interoperable with OpenMP and Pthreads

The TBB Model

- Performance
  - Algorithms attempt to generate high degree of parallelism
  - Same work stealing algorithm as Cilk for load balance
- Portability
  - Library implementation must be ported to new platforms
  - Currently requires x86 architecture

TBB Thread Management

- Developer mostly unaware of threads
  - Can specify the desired thread count at TBB initialization
  - Runtime system defaults to single thread per processor
- Developer creates tasks instead of threads
  - Tasks mapped to threads by runtime scheduler as in Cilk
  - TBB algorithms attempt to generate many tasks
- TBB runtime system handles management of threads used to process tasks

TBB Synchronization

Task synchronization

- Tasks are logical units of computation
- Tasks dynamically create new tasks
  - Split-join model applied to child tasks
  - Parent task may specify a task to be executed when all child tasks complete (explicit continuation)
  - Parent task may block and wait on children to complete before it finishes (implicit continuation)
    - Cilk threads use this model
- TBB algorithms generate and manage tasks
  - Use continuations to implement execution pattern

TBB Synchronization

Concurrent Containers

- Allow threads to access data concurrently
- Whole-container methods
  - Modify entire container
  - Must be executed by a single task
- Element access methods
  - Multiple tasks may perform element access/modification
  - Containers use mutexes as needed to guarantee consistency

TBB Synchronization

Low-level Synchronization Primitives

- Atomic template class provides atomic operations
  - Type must be integral or pointer
  - read, write, fetch-and-add, fetch-and-store, compare-and-swap operations provided by class
- Mutexes use scoped locking pattern
  - lock released when variable leaves scope
  - initialization of variable is lock acquisition

```
{
// myLock constructor acquires lock on myMutex
M::scoped_lock myLock( myMutex );
... actions to be performed while holding the lock ...
// myLock destructor releases lock on myMutex
}
```

TBB Synchronization

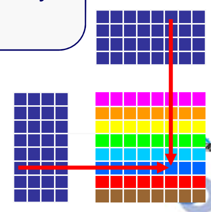
Low-level Synchronization Primitives

Mutex	Implements mutex concept using underlying OS locks (e.g., pthread mutexes)
Spin Mutex	Thread busy waits until able to acquire lock
Queuing Mutex	Threads acquire lock on mutex in the order they request it.
Reader-Writer Mutex	Multiple threads can hold lock if reading. Writing thread must have exclusive lock on mutex

TBB Matrix Multiply

```
class work {
  //data members A,B,C,L,N
public:
  void operator()(const blocked_range<size_t>& r) const {
    for(int i = r.begin(); i != r.end(); ++i) {
      for(int j=0; j<N; ++j) {
        for(int k=0; k<L; ++k) {
          C[i][j] += A[i][k]*B[k][j];
        }
      }
    }
  }
};

task_scheduler_init init;
parallel_for(
  blocked_range<size_t>(0,M,1),
   work(A,B,C,L,M)
);
```

Grainsize parameter determines how many iterations will be executed by a thread at once.



TBB Parallel Sum

```
class sum {
  float* a;
public:
  float sum;

  void operator()(const blocked_range<size_t>& r ) {
    for(size_t i=r.begin(); i!=r.end(); ++i)
      sum += a[i];
  }

  void join(sum& other) { sum += other.sum; }
};

float ParallelSumFoo(float a[], size_t n) {
  sum sum_func(a);
  parallel_reduce(blocked_range<size_t>(0,n,1), sum_func);
  return sum_func.sum;
}
```