Unified Task + Data Parallelism on Manycore Architectures

H. Carter Edwards and Stephen Olivier

SIAM Parallel Processing
February 21, 2014
SAND2014-0898C
Maximize Parallelism for Manycore Architectures; Portable & Performant

- Unified Task + Data Parallelism
  - Task decomposition of heterogeneous computations
  - Data decomposition of homogeneous computations
  - Unify: tasks of data parallel computations

- Integrate and extend Sandia’s Kokkos + Qthreads libraries
  - Kokkos: performance portable manycore data parallelism
    - Multicore CPU, NVidia GPU, Intel Xeon Phi “devices”
    - Parallel_for, parallel_reduce, parallel_scan
    - Multidimensional arrays with device-polymorphic data layout
    + Interface for tasks with internally data parallel operations
  - Qthreads: efficient scheduling of single-thread tasks
    - Multicore CPU
    + Individual tasks that span multiple threads
    + Scheduling for tasks dispatched to GPU
Kokkos: C++ Library / Programming Model

- Portable to Advanced Manycore Architectures
  - Maximize amount of user (application/library) code that can be compiled without modification and run on these architectures
  - Minimize amount of architecture-specific knowledge that a user is required to have
  - Allow architecture-specific tuning to easily co-exist
  - Only require C++1998 standard compliant

- Performant
  - Portable user code performs as well as architecture-specific code
    - Thread scalable – not just thread safety (no locking!)
    - Multidimensional array layout compatible with vectorization

- Usable
  - Small, straight-forward application programmer interface (API)
    - Constraint: don’t compromise portability and performance
Qthreads: C Library / Programming Model

- Efficient scheduling and execution of tasks
  - Lightweight tasks (very little context in comparison to pthreads)
  - NUMA- (non-uniform memory access) and cache-aware scheduling of lightweight tasks onto pinned heavyweight worker threads
  - Synchronization based on software-managed full/empty bits
  - Scalability to many thousands of tasks

- Actively used
  - Multithreaded graph library (MTGL)
  - Cray Chapel tasking layer implementation
  - OpenMP run time system using ROSE XOMP as the front-end
  - Power-aware concurrency management (MAESTRO project with RENCI)
Kokkos + Qthreads: In-progress R&D
(Sandia internal R&D “on a shoestring”)

- Three year project; four months elapsed
  - Draft specification for unified programming model @ today
  - Unit testing on multicore CPU and Xeon Phi @ 12 months
  - Heterogeneous finite element miniapplication @ 15 months
  - Informatics (graph based) miniapplication @ 18 months
  - Sparse solver w/sparse LU miniapplication @ 18 months
  - Unit testing task-data parallelism on GPU @ 21 months
  - Heterogeneous finite element miniapplication on GPU @ 24 months
  - Informatics & Sparse solver/LU miniapplication on GPU @ 30 months
Task Parallelism: Fibonacci example

```cpp
class Fib : public task_serial<long, device> {
public:
    const long n;
    int has_nested_tasks;
    Fib(long arg) : n(arg) {
        has_nested_tasks = 0;
    }
    void apply() {
        if (n < 2) { task_result = n; }
        else {
            Future<long> nested[2];
            if (has_nested_tasks) {
                nested[0] = task_dependence(0);
                nested[1] = task_dependence(1);
                task_result = nested[0].get() + nested[1].get();
            } else {
                nested[0] = spawn(Fib(n - 1));
                nested[1] = spawn(Fib(n - 2));
                has_nested_tasks = 1;
                task_respawn(nested, 2);
            }
        }
    }
};
```

- Walk through simple Fibonacci example
  - Fib(N) = if (N < 2) then N else Fib(N-1) + Fib(N-2)
  - Simple task parallel (not data parallel)
Task Parallelism: Fibonacci example

```cpp
int main() {
    long N; N << std::cin;
    Future<long> f = spawn(Fib<long, device>(N));
    wait(f);
    std::cout << f.get();
}
```

- How it all starts
  - Spawn the root Fibonacci task from “Main” thread
  - “Main” is not a task, can wait for a task to complete
- Future: a reference to a Task

- Simplistic example of a task parallel algorithm
  - There are more efficient algorithms without redundant tasks
  - Presented to illustrate the programming model
  - Do not interpret this example as a recommended pattern
Task Parallelism; Fibonacci example

```cpp
class Fib : public task_serial<long, device> {
public:
    const long n;
    int has_nested_tasks;
    Fib(long arg) : n(arg), has_nested_tasks(0) {}
    void apply()
    {
        if (n < 2)
            task_result = n;
        else {
            Future<long> nested[2];
            if (has_nested_tasks)
                nested[0] = task_dependence(0);
            nested[1] = task_dependence(1);
            task_result = nested[0].get() + nested[1].get();
        } else {
            nested[0] = spawn(Fib(n - 1));
            nested[1] = spawn(Fib(n - 2));
            has_nested_tasks = 1;
            task_respawn(nested, 2);
        }
    }
};
```

- **A Task** is a Functor derived from a parallel pattern
  - “Fib” derived from “task_serial”
  - Data type of the task’s result: “long”
  - Executes on “device”

- **Functor**: C++ class with a required member function
  - User defined data members: “n” and “has_nested_tasks”
  - Execution calls “apply”
Task Parallelism; Fibonacci example

class Fib : public task_serial<long,device> {
public:
    const long n;
    int has_nested_tasks;
Fib( long arg ) : n( arg ) , has_nested_tasks(0) {} 

void apply() {
    if ( n < 2 ) { task_result = n ; }  
    else {
        Future<long> nested[2] ;
        if ( has_nested_tasks ) {
            nested[0] = task_dependence(0);
            nested[1] = task_dependence(1);
            task_result = nested[0].get() + nested[1].get();
        } else {
            nested[0] = spawn( Fib(n-1) );
            nested[1] = spawn( Fib(n-2) );
            has_nested_tasks = 1 ;
            task_respawn( nested , 2 );
        }
    }
}
};

• “Apply” function called from a single thread
• Task sets its “task_result”
  – task_serial<...>::task_result
• Task will be complete when “apply” returns
  – Except when ... , we will come back to this
class Fib : public task_serial<long,device> {
public:
    const long n;
    int has_nested_tasks;
    Fib( long arg ) : n( arg ) , has_nested_tasks(0) {}
    void apply() {
        if ( n < 2 ) { task_result = n ; }
        else {
            Future<long> nested[2] ;
            if ( has_nested_tasks ) {
                nested[0] = task_dependence(0);
                nested[1] = task_dependence(1);
                task_result = nested[0].get() + nested[1].get();
            } else {
                nested[0] = spawn( Fib(n-1) );
                nested[1] = spawn( Fib(n-2) );
                has_nested_tasks = 1 ;
                task_respawn( nested , 2 );
            }
        }
    }
};

• Future : a reference to a Task
• Query completed tasks Fib(N-1) and Fib(N-2)
  – task_serial<...>::task_dependence(#)
  – If these tasks exist ... ; where did they come from?
  – task_result = Fib(N-1) + Fib(N-2)
class Fib : public task_serial<long,device> {
public:
    const long n;
    int has_nested_tasks;
    Fib( long arg ) : n( arg ) , has_nested_tasks(0) {}
    void apply() {
        if ( n < 2 ) { task_result = n ; }
        else {
            Future<long> nested[2] ;
            if ( has_nested_tasks ) {
                nested[0] = task_dependence(0);
                nested[1] = task_dependence(1);
                task_result = nested[0].get() + nested[1].get();
            } else {
                nested[0] = spawn( Fib(n-1) );
                nested[1] = spawn( Fib(n-2) );
                has_nested_tasks = 1 ;
                task_respawn( nested , 2 );
            }
        }
    }
};

• If tasks Fib(N-1) and Fib(N-2) don’t exist then spawn them
  – Future<result_type> spawn( FunctorType );
  – Schedules a new task for parallel execution

• I need to wait for these tasks to complete, but I cannot block
  – GPU: a thread cannot block and spin-waiting is very bad
  – CPU: overhead of context-switch which is bad for performance

• Solution: re-spawn myself
  – “I cannot complete now, call me again after these other tasks complete”
  – task_serial<...>::task_respawn( Future<> depend[] , # );
  – When “apply” returns I am not complete

nested[0] = spawn( Fib(n-1) );
nested[1] = spawn( Fib(n-2) );
has_nested_tasks = 1 ;
task_respawn( nested , 2 );
Task Dependence Directed Acyclic Graph (DAG)

- Dependence not limited to spawner-spawnee (parent-child)
  - E.g., Intel Cilk and OpenMP 3.x
- General task dependence DAG
Task-Data Parallelism: L2 norm example

class Norm2 : public task_reduce<double, device> {
public:
    const double * X;
    const int N;
    int work() const { return N; }

    void operator()( int i, double & v ) const { v += X[i] * X[i]; }
    void init( double & v ) const { v = 0; }
    void join( volatile double & v, volatile const double & w ) const { v += w; }

    void apply() { task_result = sqrt( task_result );

    Norm2( const double * argx, const int argn ) : x(argx), n(argn) {} };

// elsewhere in the “main” thread:
Future<double> f = spawn( Norm2<double, device>(X, N) );
wait( f );
double value = f.get();

• Walk through simple L2 norm example
  – Task with data parallel pattern \[ \sqrt{\sum_{i=0}^{N-1} x_i^2} \]

• Spawn just like a serial task
  – A future for waiting and querying result
class Norm2 : public task_reduce<double,device> {
public:
  const double * X ;
  const int            N ;
  int work() const { return N ; }

void operator()( int i , double & v ) const { v += X[i] * X[i] ; }
void init( double & v ) const { v = 0 ; }
void join( volatile double & v , volatile const double & w ) const { v += w ; }

Norm2( const double * argx , const int argn ) : x(argx), n(argn) {};

// elsewhere in the “main” thread:
Future<double> f = spawn( Norm2(x,n) );
wait( f );
double value = f.get();

• Parallel pattern : “task_reduce”
  – Data parallel with “N” units of work
  – “work” function returns “N”
Task-Data Parallelism: L2 norm example

class Norm2 : public task_reduce<double,device> {
public:
    const double * X ;
    const int N ;

    int work() const { return N ; }
    void operator()( int i , double & v ) const { v += X[i] * X[i] ; }
    void init( double & v ) const { v = 0 ; }
    void join( volatile double & v , volatile const double & w ) const { v += w ; }

    Norm2( const double * argx , const int argn ) : x(argx), n(argn) {} ;
};

// elsewhere in the "main" thread:
Future<double> f = spawn( Norm2(x,n) );
wait( f );
double value = f.get();

• Call “operator()” function N times
  – Work index: \( i \in [0..N) \)
  – Called in parallel from “P” threads; \( P \leq N \)

• Contribute to reduction value “v”
  – Argument “v” is thread-private
class Norm2 : public task_reduce<double,device> {
public:
    const double * X ;
    const int N ;
    int work() const { return N ; }
    void operator()( int i , double & v ) const { v += X[i] * X[i] ; }
    void init( double & v ) const { v = 0 ; }
    void join( volatile double & v , volatile const double & w ) const { v += w ; }
    void apply() { task_result = sqrt( task_result );}
};

// elsewhere in the "main" thread:
Future<double> f = spawn( Norm2(x,n) );
wait( f );
double value = f.get();

- Thread-private values are *partial* contributions
  - Values must be properly *initialized* (might not be zero)
  - Values must be properly *joined* across threads (might not be sum)

- Why “volatile”?  
  - For performance thread private values are joined in-place  
  - Implementation violates threads’ privacy 😲  
  - “volatile” so that compiler is “in the know” 😲
Task-Data Parallelism: L2 norm example

class Norm2 : public task_reduce<double,device> {
public:
    const double * X ;
    const int N ;
    int work() const { return N ; }
    void operator()( int i , double & v ) const { v += X[i] * X[i] ; }
    void init( double & v ) const { v = 0 ; }
    void join( volatile double & v , volatile const double & w ) const { v += w ; }
    void apply() { task_result = sqrt( task_result ); }
};

Future<double> f = spawn( Norm2(x,n) );
wait( f );
double value = f.get();

- Task’s final serial step
  – Opportunity to serially process the task’s result

[Diagram: Data parallel “operator()”]

[Diagram: final serial “apply”]
class Search : public task_serial<void,device> {
public:
    const KeyType       key ;
    const NodeType * node ;

    void apply() {
        Future<NodeType*> group = task_dependence(0);
        if ( node->key == key ) {
            group_complete( group , node );
        } else {
            for ( int i = 0 ; i < node->num_child ; ++i )
                spawn( Search(key, node->child[i]) , & group , 1 );
        }
    }
};

// elsewhere in the “main” thread
Future<NodeType*> group = group_create<NodeType*>(0);
spawn( Search( key , root_node ) , & group , 1 );
wait( group );
NodeType * found_node = group.get();

• Search an n-tree for a node with a given key
  – Spawn a search task at each node
  – Until the “winning” task finds the matching node

• Create a task group, returns a Future
• Spawn search task on the tree’s root node
  – Dependence on the task group
• Wait for the group to complete
class Search : public task_serial<void,device> {
  public:
    const KeyType key ;
    const NodeType * node ;
  void apply() {
    Future<NodeType*> group = task_dependence(0);
    if ( node->key == key ) {
      group_complete( group , node );
    } else {
      for ( int i = 0 ; i < node->num_child ; ++i )
        spawn( Search(key, node->child[i]) , & group , 1 );
    }
  }
};
// elsewhere in the “main” thread
Future<NodeType*> group = group_create<NodeType*>(0);
spawn( Search(key, root_node) , & group , 1 );
wait( group );
NodeType * found_node = group.get();

• Query group’s future, is a dependence
• If node matches complete the group
  – Set the group’s result value
  – An atomic operation: only one task can “win”
• Completing the group cancels execution of all remaining tasks in that group
  – Prevents them from starting, removes them from schedule
Task Groups: Tree Search example

class Search : public task_serial<void,device> {
public:
  const KeyType       key ;
  const NodeType * node ;
void apply() {
  Future<NodeType*> group = task_dependence(0);
  if ( node->key == key ) {
    group_complete( group , node );
  } else {
    for ( int i = 0 ; i < node->num_child ; ++i )
      spawn( Search(key, node->child[i]) , & group , 1 );
  }
}
};

// elsewhere in the “main” thread
Future<NodeType*> group = group_create<NodeType*>(0);
spawn( Search( key , root_node ) , & group , 1 );
wait( group );
NodeType * found_node = group.get();

• If node does not match then spawn tasks for child nodes
  – Spawn as member of the group
  – If another task already “won” then spawned tasks will not execute
• Change from tree search to graph search
  – Atomic update of a “has this node been searched?” flag before spawning a child search task
Wrapping Up (conclusions pending, stay tuned)

- **Unified Task + Data Parallelism**
  - Seamless integration of task and data parallel computations
  - Task decomposition of heterogeneous computations
  - Data decomposition of homogeneous computations

- **Integrate and extend Sandia’s Kokkos + Qthreads libraries**
  - Kokkos: performance portable manycore data parallelism
  - Qthreads: efficient scheduling of single-thread tasks

- **Proof will be in the mini-applications**
  - Heterogeneous finite element
  - Informatics (graph based)
  - Sparse solver with parallel sparse LU factorization & preconditioning
    - Performance portable mini-application code to CPU, Xeon-Phi, GPU