A Unified Futures Interface in C++ for Shared and Distributed Memories

Dimitrios Chasapis

PARSYS - LRI

September 12, 2013

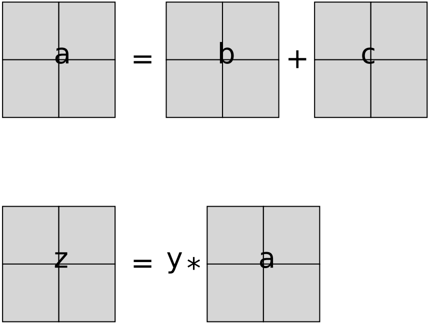
Parallel Programming

- Threads
  - Shared Memory
  - Require explicit synchronization (mutexes, barriers, etc)
- Message Passing
  - Distributed Memory (also available for Shared Memory)
  - Writing code with messages can be difficult
  - Can exploit data localization more naturally and effectively
- Both models can be challenging to use!

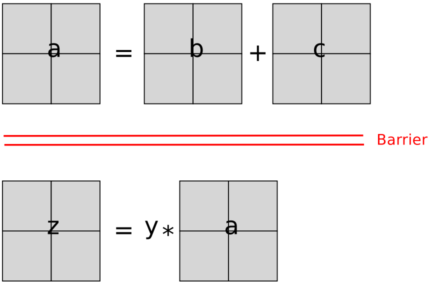
NT2: The Numerical Template Toolbox

- NT2
  - A Matlab like interface in C++
  - High-performance parallel implementation
  - Easier to use than Threads and Message Passing
  - Currently available for Shared Memory (under active development by Pierre Esterie)
  - Distributed Memory version under development (Antoine Tran Tan)

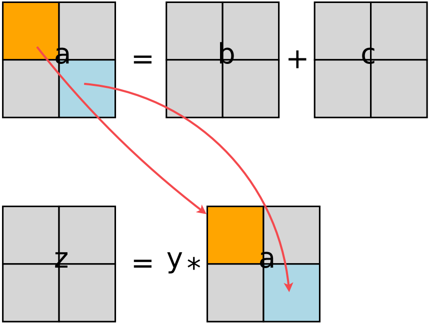
Inter-statement optimization



Inter-statement optimization



Inter-statement optimization



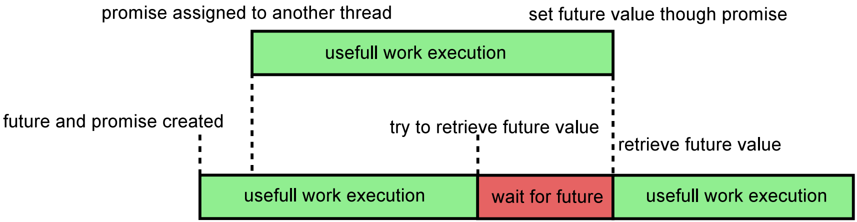
- PGAS (Partitioned Global Address Space) issues with this problem
- C++ is not a PGAS language...

Futures and Promises

- Futures
  - A Future variable encapsulates a data value that may not be available at the time of reference
  - It is used only to read the encapsulated data.
- Promises
  - A Promise variable is associated with a Future and provides the means to set its data value.
  - It is used only to write the encapsulated data

- Available in many languages (Java, C#, C++, Scala, etc)

Futures and Promises:Workflow Example



Futures and Promises

We want a futures interface for Shared and Distributed Memory

- HPX
  - Good performance on shared memory
  - Distributed memory version is still problematic and under development
  - Will take time to support all architectures
- Charm++
  - Good performance for medium grained processes
  - Object oriented model
  - Complicated interface

Futures and Promises

Goal: DIY Futures

- Simple usable interface
- Portable
- Efficient

- In Shared Memory easy to implement

Futures and Promises

Goal: DIY Futures

- Simple usable interface
- Portable
- Efficient

- In Shared Memory easy to implement
- What about Distributed Memory?

Futures and Promises

Goal: DIY Futures

- Simple usable interface
- Portable
- Efficient

- In Shared Memory easy to implement
- What about Distributed Memory?
- Answer:Asynchronous communication

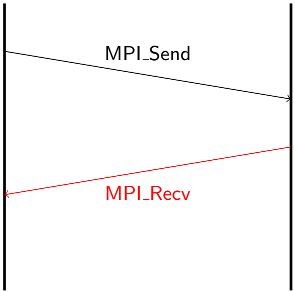
Asynchronous Communication

- Emulate asynchrony
  - Polling for messages
    - Unresponsive it time interval is small
    - Polling can dominate computation if time interval too great
  - Hardware Interrupts
    - High cost
    - Not always available
  - Dedicating a thread for communication
    - Performance depends on thread implementation and OS

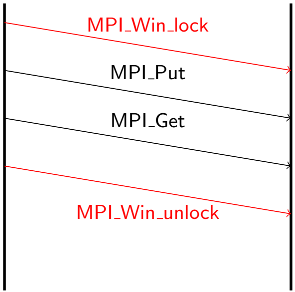
Asynchronous Communication

- Emulate asynchrony
  - Polling for messages
    - Unresponsive it time interval is small
    - Polling can dominate computation if time interval too great
  - Hardware Interrupts
    - High cost
    - Not always available
  - Dedicating a thread for communication
    - Performance depends on thread implementation and OS
- One-sided communication (MPI-2, ARMCI)

MPI-2 One-sided Communication

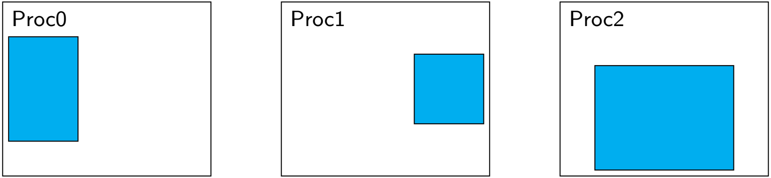


Two-sided communication



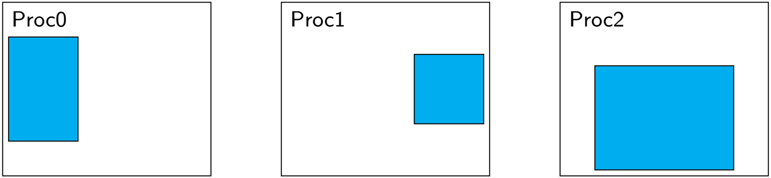
One-sided communication

MPI-2 One-sided Communication



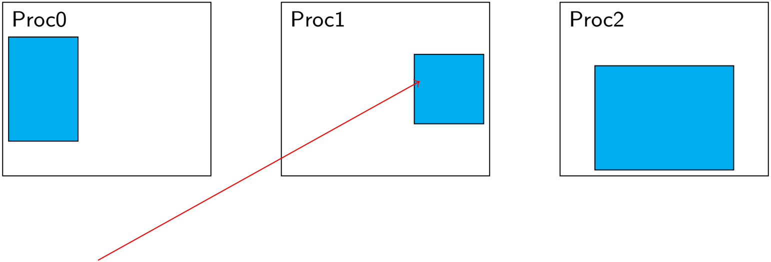
- Local data is exposed through `MPI_Window` objects
- Windows are created by calling the `MPI_Win_create`
- `MPI_Win_create` is a collective operation

MPI-2 One-sided Communication



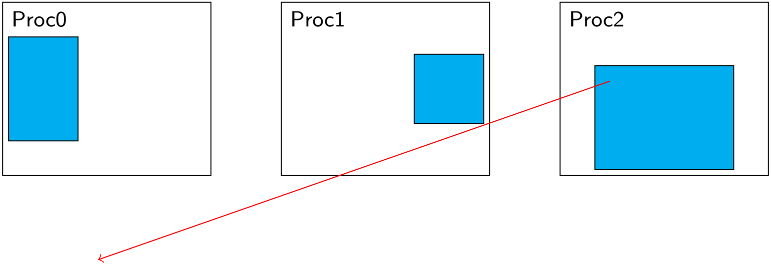
- Remote put/get operations can be performed on memory windows
- These operations must happen in an "epoch"
- An "epoch" is a time frame defined by successive calls of the MPI synchronization primitives
  More on that Later!

MPI-2 One-sided Communication




```
MPI_Put(base_ptr, size, datatype, 1, offset,
        size, datatype, window);
```

MPI-2 One-sided Communication

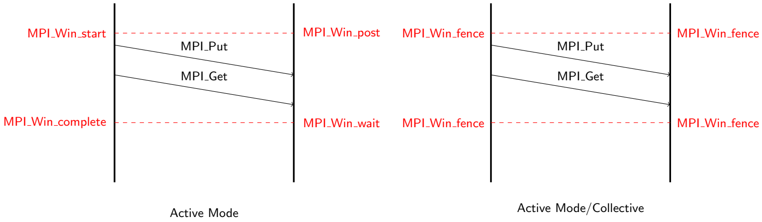



```
MPI_Get(base_ptr, size, datatype, 2, offset,
        size, datatype, window);
```

MPI-2 One-sided Communication

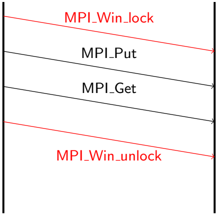
- Put and Get operations need to be synchronized!
  Remember "epochs"?
- Why?
  - Concurrent accesses to the same window and overlapping data are erroneous
  - End of "epoch" marks that message was send or received
- Two modes of synchronization
  - Active mode: All processes are required to take part in the synchronization
  - Passive mode: Only the process initiating the put/get operation needs to synchronize

MPI-2 One-sided Communication



- All processes must be aware of the other processes that take part in the communication
- Communication is not initiated asynchronously

MPI-2 One-sided Communication



Passive Mode

- Really asynchronous!
- Can only write/read data allocated with `MPI_Alloc_mem`
- `MPI_Win_lock` can only be acquired for the whole window
- Not real locks, cannot be used to define critical regions

MPI Futures

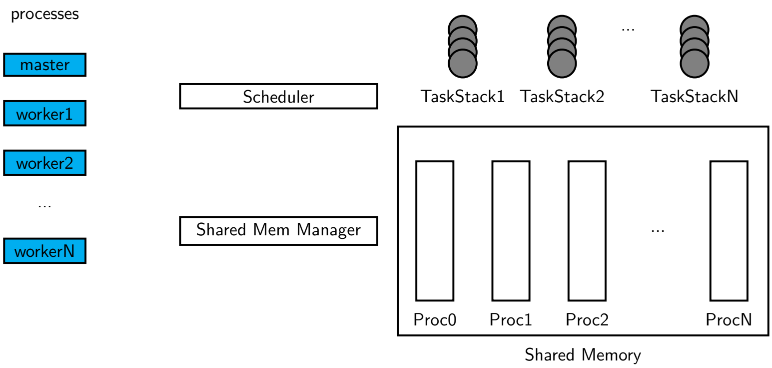
- C++11 standard library future interface over message passing
- Modular design
  - Scheduler:
    - Distributes asynchronous functions (aka *jobs*)
    - Maintains stacks to keep *jobs*, on each process
  - Shared Memory Manager:
    - Provides a virtual shared memory view through a `Shared_ptr` variable
    - Provides the routines to read/write and allocate data on the virtual shared address space
  - Communication Manager:
    - Provides asynchronous message passing
    - Provides a mechanism to expose share local process space
    - Implemented using MPI one-sided communication library

MPI Futures:Interface

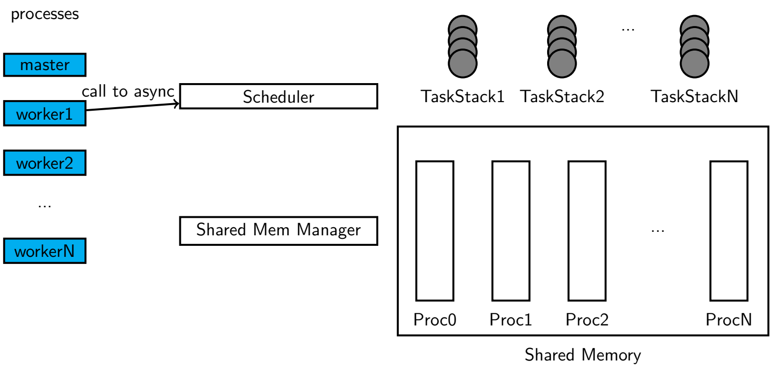
```
class fib {
public:
  fib() {};
  ~fib() {};
  int operator()(int n) {
    if(n == 0) return 0;
    if(n == 1) return 1;
    fib f;
    future<int> fib1 = async(f, n-1);
    future<int> fib2 = async(f, n-2);
    return fib1.get() + fib2.get();;
  };
};

FUTURES_SERIALIZE_CLASS(fib);
FUTURES_EXPORT_FUNCTOR((async_function<fib, int>));
```

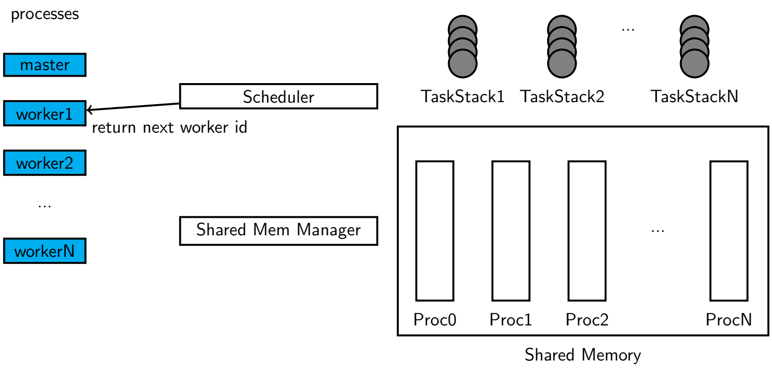
MPI Futures:Implementation



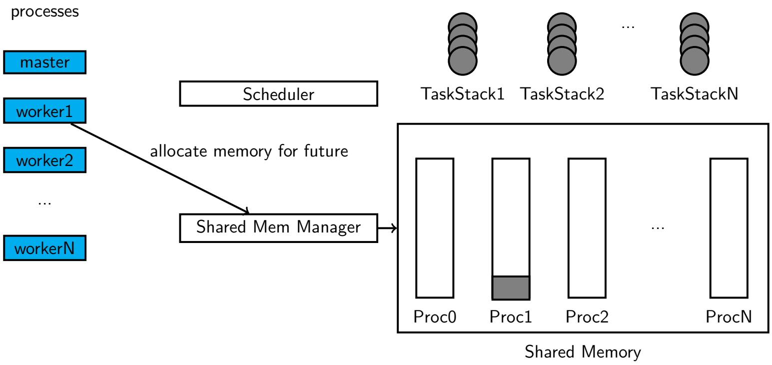
MPI Futures:Implementation



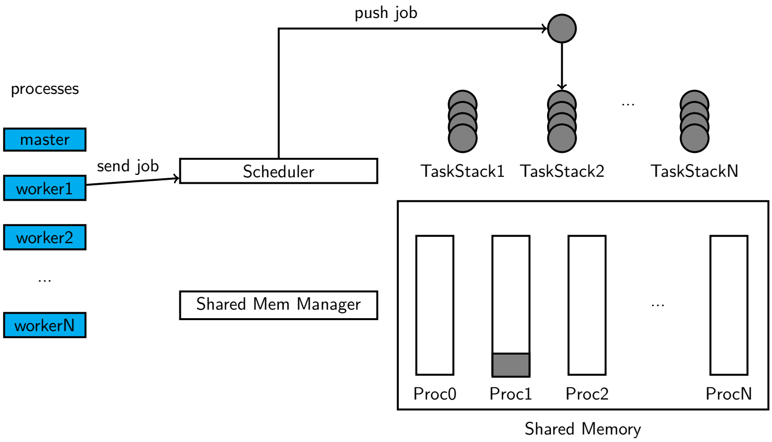
MPI Futures:Implementation



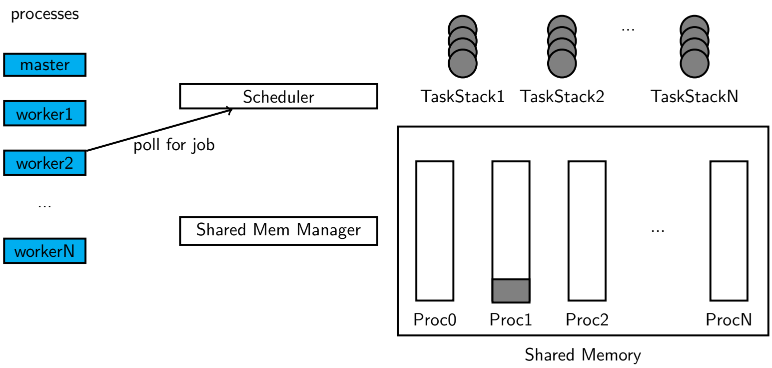
MPI Futures:Implementation



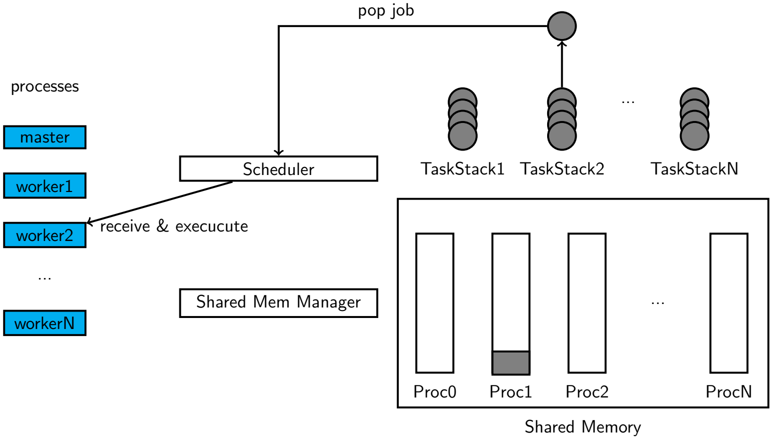
MPI Futures:Implementation



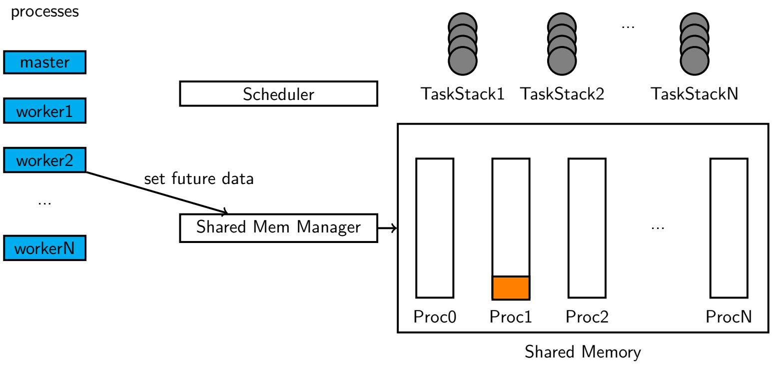
MPI Futures:Implementation



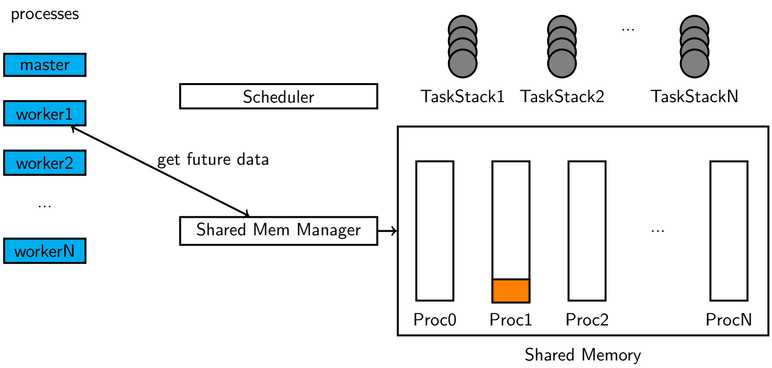
MPI Futures:Implementation



MPI Futures:Implementation



MPI Futures:Implementation



Interface Assesment

```cpp
class fib {
public:
  fib() {};
  ~fib() {};
  int operator()(int n) {
    if(n == 0) return 0;
    if(n == 1) return 1;
    fib f;
    future<int> fib1 = async(f, n-1);
    future<int> fib2 = async(f, n-2);
    return fib1.get() + fib2.get();;
  };
};

FUTURES_SERIALIZE_CLASS(fib);
FUTURES_EXPORT_FUNCTOR((async_function<fib, int>));
```

```cpp
int fibonacci(int n) {
  if(n == 0) return 0;
  if(n == 1) return 1;
  future<int> fib1 = async(fibonacci, n-1);
  future<int> fib2 = async(fibonacci, n-2);
  return fib1.get() + fib2.get();
}
```

- Very close to the C++11 standard!
- Easy to use compared to original message passing code
- Easy to expose functor objects to worker processes

Interface Assesment

Limitations

- Only serializable functor objects can be issued by *async*
- Arguments must also be serializable
- Size of dynamically sized objects as return values must be declared at *async* callsite

Performance Evaluation:Setup

Evaluation setup:

- 2 Intel(R) Xeon(R) CPU E5645@2.40 GHz with 6 available cores on each machine
- Connected through a network socket, but provide shared memory interface
- Compiled using g++ version 4.6.3 with level 3 optimization enabled
- OpenMPI version 1.4.3
- Boost serialization version 1.53

Performance Evaluation:Benchmarks

Three benchmarks:

- Fibonacci:
  - We run it for a $==$ 45
  - For a $<$ 30, we run the sequential code
- Quicksort:
  - We run it for sorting 10,000,000 elements
  - For n $<$ 100,000 we run the sequential code
- Tiled LU

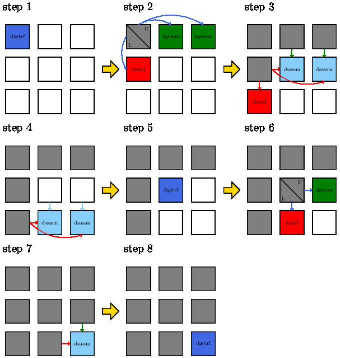
Performance Evaluation:Benchmarks

Three benchmarks:

- Fibonacci:
  - We run it for a $== 45$
  - For a < 30, we run the sequential code
- Quicksort:
  - We run it for sorting 10,000,000 elements
  - For n $<$ 100,000 we run the sequential code
- Tiled LU
  - A little more complicated...

Performance Evaluation:Tiled LU

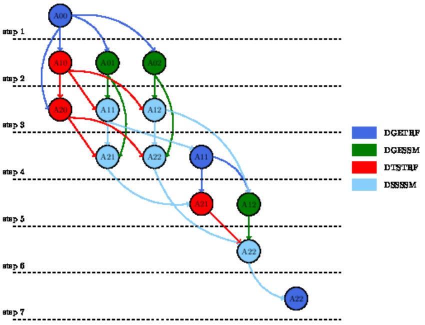
```
for(int k = 0; k < TOTAL_TILES; k++) {
  dgetrf(A[k][k], P[k][k]);
  for(int n = k+1; n < TOTAL_TILES; n++) {
    dgessm(A[k][n], A[k][k], P[k][k])
  }
  for(int m = k+1; m < TOTAL_TILES; m++) {
    dtstrf(A[k][k], A[m][k] , P[m][k]);
    for(int n=k+1; n < TOTAL_TILES; n++) {
      dssssm(U[k][n], A[m][n], L[m][k],
                      A[m][k], P[m][k]);
    }
  }
}
```



- Originally implemented with PLASMA

Performance Evaluation:Tiled LU

```
for(int k = 0; k < TOTAL_TILES; k++) {
  A[k][k] = cpldAU[k][k].get().A;
  dgetrf(A[k][k], P[k][k]);
  for(int n = k+1; n < TOTAL_TILES; n++) {
    A[k][n] = cpldAU[k][n].get().A;
    fA[k][n] = async(dgessm, A[k][n],
                     A[k][k], P[k][k]);
  }
  for(int m = k+1; m < TOTAL_TILES; m++) {
    A[m][k] = cpldAU.get().A;
    dtstrf(A[k][k], A[m][k].get() , P[m][k]);
    for(int n=k+1; n < TOTAL_TILES; n++) {
      if(m == k+1)
        A[k][n] = fA[k][n].get();
      else
        A[k][n] = cpldAU.get().U;
      A[m][n] = cpldAU.get().A;
      cpldAU[m][n] = async(dssssm, A[k][n], A[m][n],
                           L[m][k], A[m][k], P[m][k]);
    }
  }
}
```

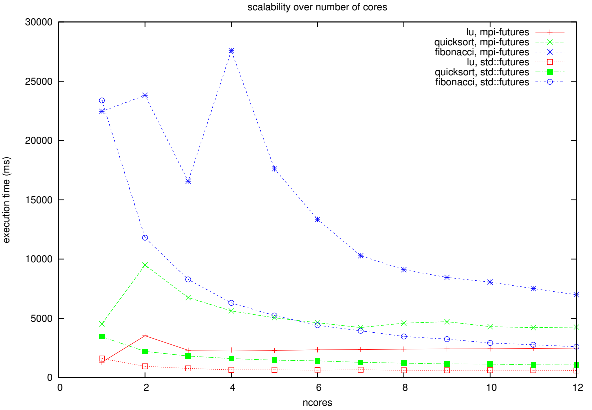


Performance Evaluation:Benchmarks

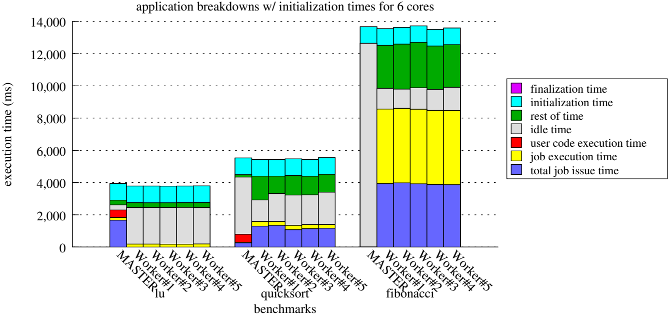
Three benchmarks:

- Fibonacci:
  - We run it for a $==$ 45
  - For a $<$ 30, we run the sequential code
- Quicksort:
  - We run it for sorting 10,000,000 elements
  - For n $<$ 100000, we run the sequential code
- Tiled LU
  - We apply LU factorization on an 2,000x2,000 matrix
  - Block size is 200x200 elements

Performance Evaluation



Performance Evaluation

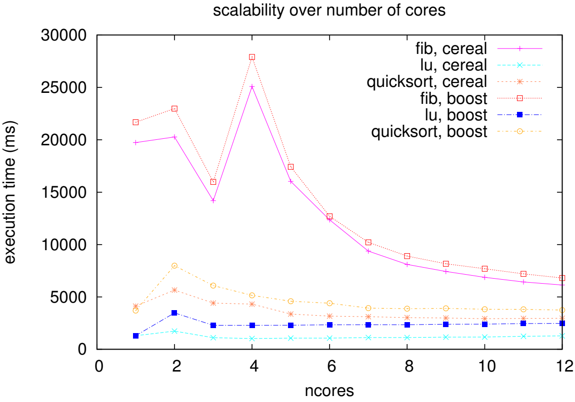


- Fibonacci spends 30% trying to acquire lock in scheduler
- Quicksort spends 25% trying to acquire lock and 15% on vector copying and 16% on serialization
- In LU master spends 70% on serialization routines

Improving Performance

- Boost Serialization is a dominant source of overhead
- Use an alternative serialization library with better performace
    - CEREAL serialization library!

Improving Performance



- Performance still troublesome, but we get some speedup!

Conclusions

- Implementing the C++11 standard future's interface over MPI is possible!
  - Very usable
  - Sensible limitations

- Two major "Deal breakers" regarding performance
  - Locks over MPI
  - Boost Serialization

Conclusions

- MPI-2 one-sided communication not as versatile but usable
  - `MPI_Win_create` can only be created collective
    Not possible to share data dynamically in an asynchronous manner
  - Active mode synchronization cannot be initiated asynchronously
  - Passive mode cannot handle static data
  - Lacks synchronization primitives (Active and Passive mode synchronize message completion, they do not define critical regions)

Future Work

- Fix performance
- Use coarser grain applications for benchmarking
- Try a hybrid approach.
- Implement secondary C++11 library features (exceptions, timeouts, etc)