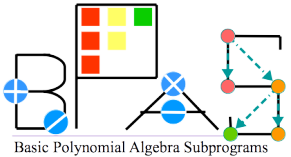
Multithreaded Programming in the BPAS Library

Building Object-Oriented (and Optional) Parallelization from C++ Primitives

Alexander Brandt

Faculty of Comptuer Science, Dalhouise University, Canada
Ontario Research Centre for Computer Algebra, University of Western Ontario

July 22, 2026



Introduction

```
1  void mergeSort(int* A, int i, int j) {
2      if (j <= i) {
3          return;
4      }
5      int k = i + (j-1)/2;
6      mergeSort(A, i, k);
7      mergeSort(A, k, j);
8      merge(A, i, k, j);
9  }
```

→ Where recursive calls are independent, those function calls can be executed **concurrently**

→ **Fork** the execution control flow and then **join** or **sync** them

→ Hardware must support parallelism with multi-core or multi-processors

Compiler-Level Automatic Parallelization

→ CILK and OPENMP provide automatic parallelization though compiler extensions

→ Very easy but flexibility more challenging

```
void mergeSort(int* A, int i, int j) {
  //... base case, k
  cilk_spawn mergeSort(A, i, k);
  mergeSort(A, k, j);
  cilk_sync
  merge(A, i, k, j);
}
```

```
void mergeSort(int* A, int i, int j) {
  //... base case, k
  #pragma omp parallel num_threads(2)
  {
    #pragma omp sections {
      #pragma omp section {
        mergeSort(A, i, k);
      }
      #pragma omp section {
        mergeSort(A, k, j);
      }
    }
  }
  merge(A, i, k, j);
}
```

Dynamic Multithreading with BPAS

→ Object-oriented, Standard C++11, No compiler extensions

→ Supports composition of parallel regions

→ **Dynamic Multithreading**: programmers specify opportunities for concurrency, runtime decides parallel execution

```
1  void mergeSort(int* A, int i, int j) {
2      //... base case, k
3      threadID id;
4      ExecutorThreadPool& pool = ExecutorThreadPool::getThreadPool();
5
6      pool.obtainThread(id);
7      pool.executeTask(id, std::bind(mergeSort, A, i, k));
8      mergeSort(A, k, j);
9      pool.waitForThread(id);
10
11     merge(A, i, k, j);
12 }
```

Threading Primitives

C++11 introduced the *Thread Support Library*

- `std::thread`
  - C++ class encapsulating a thread (often a `pthread`) and its low-level spawn and join
- `std::mutex`
  - shared object between threads to indicate *mutual exclusion* to a **critical region**.
  - `mutex` is *locked* or *owned* by at most one thread at a time.
- `std::lock_guard`, `std::unique_lock`
  - temporary object wrapping a `mutex` whose object lifetime automatically locks and unlocks the `mutex`.
  - the constructor **blocks** and only returns once the shared `mutex` is successfully owned by the calling thread.
- `std::condition_variable`
  - blocks the current thread and temporarily releases a lock
  - receives notification from another thread to awaken the blocked thread

std::function

Functors, function objects, callable objects

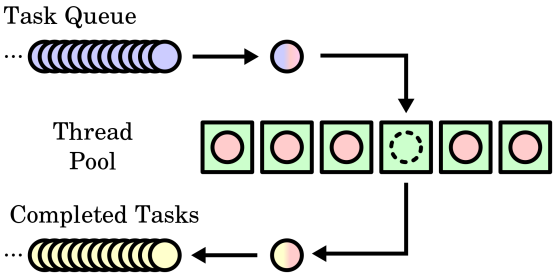
- → First-class objects which are callable using normal function syntax
- → Are often constructed by passing function names, function pointers
- → `std::bind` binds arguments to a function or function object, returning a function object which requires fewer arguments

```
1  void printInteger(int a) {
2      std::cout << a << std::endl;
3  }
4
5  //Function object from function name
6  std::function<void(int)> f_printInt(printInteger);
7  f_printInt(12);
8
9  //Function object binding arguments to function name
10 std::function<void()> f_print42 = std::bind(printInteger,42);
11 f_print42();
```

Outline

Thread Pools

- → Highly parallel programs benefit for a thread pool: avoiding thread spawn overhead, **resource contention**
- → Threads spawn only once at beginning of program
- → Threads remain active throughout program lifetime



Function Executor Thread

`FunctionExecutorThread`

- → Internal `std::thread` lifetime bound to object lifetime
- → Executes `std::function` objects asynchronously
- → `sendRequest(std::function<void()>)`: non-blocking submit task
- → `waitForThread()`: blocking call until all tasks are complete
- → Results available through object references or pointers

Function Executor Thread: Implementation

```
1  class FunctionExecutorThread {
2
3      AsyncObjectStream<std::function<void()>> requestQueue;
4      std::thread m_worker;
5
6      std::mutex m_mutex;
7      std::condition_variable m_cv;
8
9      FunctionExecutorThread() {
10         //member functions require pointer to member
11         m_worker = std::thread(
12             &FunctionExecutorThread::eventLoop, this);
13     }
14
15     void eventLoop();
16
17     void sendRequest(std::function<void()>);
18
19     void waitForThread();
20 }
```

Function Executor Thread: Implementation Details

```
1  void FunctionExecutorThread::eventLoop() {
2      std::function<void()> task;
3      while(requestQueue.getNextObject(task)) {
4          task();
5          std::unique_lock<std::mutex> lk(m_mutex);
6          bool notify = requestQueue.streamEmpty();
7          lk.unlock();
8          if (notify) m_cv.notify_all();
9      }
10 }
11
12 void FunctionExecutorThread::sendRequest(std::function<void()> f) {
13     std::lock_guard<std::mutex> lk(m_mutex);
14     requestQueue.addResult(f);
15 }
16
17 void FunctionExecutorThread::waitForThread() {
18     std::unique_lock<std::mutex> lk(m_mutex);
19     while (!requestQueue.streamEmpty()) {
20         m_cv.wait(lk);
21     }
22 }
```

Object Streams

The `AyncObjectStream` class provides:

1. a queue for tasks, or any object, and
2. a blocking mechanism to keep the `FunctionExecutorThread` alive and idle when waiting for tasks

→ Actually a class template for any kind of object being passed between two threads

→ Implements a queue satisfying the **producer-consumer problem**

→ A **std::queue** combined with a mutex and condition variable

AsyncObjectStream Interface

```
1  template <class Object>
2  class AsyncObjectStream {
3
4      std::queue<Object> retObjs;
5      std::mutex m_mutex;
6      std::condition_variable m_cv;
7      bool finished; //is the stream still open?
8
9      //Producer: add an object to the queue
10     void addResult(Object&& res);
11
12     //Producer: close the producer end of stream,
13     //          no more objects to produce
14     void resultsFinished();
15
16     //Consumer: pop an object from the queue, return true
17     //          iff stream is open and objects available
18     bool getNextObject(Object& res);
19
20     //Consumer: determine if queue is currently empty
21     bool streamEmpty();
22 };
```

AsyncObjectStream: getNextObject

```
1  bool getNextObject(Object& res) {
2      std::unique_lock<std::mutex> lk(m_mutex);
3      if (finished && retObjs.empty()) {
4          lk.unlock();
5          return false;
6      }
7
8      //Wait in a loop in case of spurious wake ups
9      while (!finished && retObjs.empty() {
10         m_cv.wait(lk);
11     }
12
13     if (finished && retObjs.empty()) {
14         lk.unlock();
15         return false;
16     } else {
17         res = retObjs.front();
18         retObjs.pop();
19         lk.unlock();
20         return true;
21     }
22 }
```

ExecutorThreadPool

- → A pool of **FunctionExecutorThreads**
- → An internal queue of tasks and queue of threads
- → When threads are busy, they are temporarily removed from the pool
- → When all threads busy, tasks are added to task queue

```
1  class ExecutorThreadPool {
2
3  private:
4      std::deque<FunctionExecutorThread*> threadPool;
5      std::deque<std::function<void()>> taskPool;
6      std::mutex m_mutex;
7      std::condition_variable m_cv; // waitForThreads
8      std::vector<FunctionExecutorThread*> reservedThreads; // obtainThreads
9
10 public:
11     void addTask(std::function<void()> f);
12     void waitForThreads();
13 }
```

ExecutorThreadPool: Flexible Usage (1/2)

→ In support of certain **parallel patterns**, clients can (temporarily) obtain exclusive ownership of threads from the pool, rather than using `addTask`

→ Abstract away actual threads through **thread IDs**

→ Once thread obtained, repeat Steps 2–3 as often as necessary

```
1  class ExecutorThreadPool {
2
3      //Step 1: obtain a thread's ID, removing it from the pool
4      void obtainThread(threadID& id);
5
6      //Step 2: execute a task on a particular thread
7      void executeTask(threadID id, std::function<void()>& f);
8
9      //Step 3 (optional): wait for thread to become idle
10     void waitForThread(threadID id);
11
12     //Step 4: wait for thread and then return thread to pool
13     void returnThread(threadID id);
14 }
```

ExecutorThreadPool: Flexible Usage (2/2)

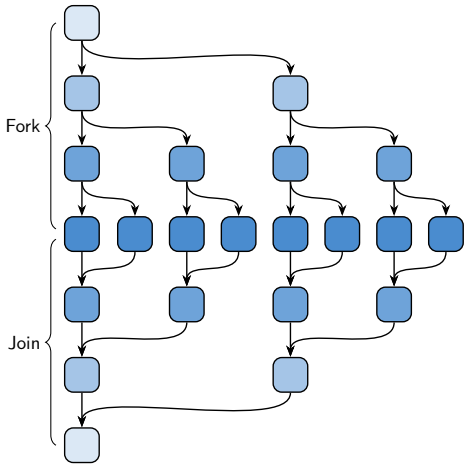
→ In support of certain **parallel patterns**, clients can (temporarily) obtain exclusive ownership of threads from the pool, rather than using `addTask`

→ Can obtain multiple threads at a time

```
1  class ExecutorThreadPool {
2
3      //Step 1: obtain threadIDs, removing them from the pool
4      void obtainThreads(std::vector<threadID>& ids);
5
6      //Step 2: execute a task on a particular thread
7      void executeTask(threadID id, std::function<void()>& f);
8
9      //Step 3 (optional): wait for threads to become idle
10     void waitForThreads(std::vector<threadID>& ids);
11
12     //Step 4: wait for threads and then return threads to pool
13     void returnThreads(std::vector<threadID>& ids);
14 }
```

Outline

Fork-Join



→ **Fork**: divide problem and execute separate calls in parallel

→ **Join**: merge parallel execution back into serial

→ Recursively applying fork-join can easily parallelize a divide-and-conquer algorithm

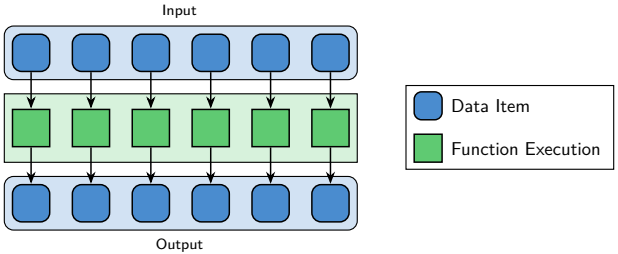
Fork-Join with ExecutorThreadPool

```
1  void mergeSort(int* A, int i, int j) {
2      if (j <= i) { return; }
3      int k = i + (j-1)/2;
4      mergeSort(A, i, k);
5      mergeSort(A, k, j);
6      merge(A, i, k, j);
7  }
```

```
1  void mergeSort(int* A, int i, int j) {
2      if (j <= i) { return; }
3      int k = i + (j-1)/2;
4      threadID id;
5      ExecutorThreadPool& pool = getThreadPool();
6
7      pool.obtainThread(id);
8      pool.executeTask(id, std::bind(mergeSort, A, i, k));
9      mergeSort(A, k, j);
10
11     pool.returnThread(id);
12     merge(A, i, k, j);
13 }
```

Map

- → Simultaneously execute a function on each data item in a collection
- → If more data items than threads, apply the pattern block-wise: partition the collection and apply one thread to each partition
- → Often simplified as just a `parallel_for` loop
- → Where multiple map steps are performed in a row, they should operate in lockstep



Map with ExecutorThreadPool

```
1  //Apply f to each item of A, returning results in B
2  void MapExample(int* B, int* A, int n, std::function<void(int*,int*)> f) {
3
4      for (int i = 0; i < n; ++i) {
5          f(&B[i], &A[i]);
6      }
7  }
```

```
1  void MapExample(int* B, int* A, int n, std::function<void(int*,int*)> f) {
2
3      ExecutorThreadPool& pool = getThreadPool();
4      std::vector<threadID> ids;
5      pool.obtainThreads(n-1, ids); //assume n-1 threads avail.
6
7      for (int i = 0; i < n-1; ++i) {
8          pool.executeTask(ids[i], std::bind(f, &B[i], &A[i]));
9      }
10     f(&B[n-1], &A[n-1]); //use main thread for one call
11
12     pool.returnThreads(ids); //also waits for threads
13 }
```

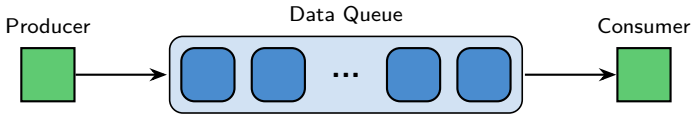
Producer-Consumer

→ Two functions connected by a queue

→ The producer produces data items, pushing them to the queue

→ The consumer processes data items, pulling them from the queue

→ Producer and consumer execute simultaneously; must avoid **deadlock**



→ In some circumstances, the producer may be considered as an **iterator** or **generator**

Generators

→ Generators are special kinds of **coroutines** which **yield** data items one at a time, rather than many as a collection

→ A yield pauses execution of the function, and allows computations to resume from that point at the next function call

```
1  void FibonacciGen(int n) {
2      int Fn_1 = 0;
3      int Fn = 1;
4      for (int i = 0; i < n; ++i) {
5          yield Fn_1;
6          Fn = Fn + Fn_1;
7          Fn_1 = Fn - Fn_1;
8      }
9  }
```

→ Where the generation of data items is itself expensive, generators may execute asynchronously, following the producer-consumer pattern: `AsyncGenerator`

AsyncGenerator Example

```
1  void FibonacciGen(int n, AsyncGenerator<int>& gen) {
2      int Fn_1 = 0;
3      int Fn = 1;
4      for (int i = 0; i < n; ++i) {
5          gen.generateObject(Fn_1); //yield Fn_1 and continue
6          Fn = Fn + Fn_1;
7          Fn_1 = Fn - Fn_1;
8      }
9      gen.setComplete();
10 }
11
12 void Fib() {
13     int n;
14     std::cin >> n;
15     AsyncGenerator<int> gen(FibonacciGen, n);
16
17     int fib;
18     //get one integer at a time until generator is finished
19     while (gen.getNextObject(fib)) {
20         std::cerr << fib << std::endl;
21     }
22 }
```

AsyncGenerator Implementation

```
template <class Object>
class AsyncGenerator {
    AsyncObjectStream<Object> stream; //Objects instead of std::function
    FunctionExecutorThread t;

    //Create a generator from a function and its arugments
    template <class Function, class... Args>
    AsyncGenerator(Funcion& f, Args&...args) {

        //pass reference to generator object to producer function
        std::function<void()> boundF = std::bind(f, args..., std::ref(*this));
        t.sendRequest(boundF);
    }

    //delegate to stream's methods
    void generateObject(Object& obj) { stream.addResult(obj); }
    void setComplete()               { stream.resultsFunished(); }
    bool getNextObject(Object& obj)  { return stream.getNextObject(obj); }

}
```

Outline

ExecutorThreadPool Optional Parallelism

- → If all threads already reserved, `obtainThread(id)` may fail to find a thread
- → In this case, return special value `const threadID notAThread`
- → `executeTask(id, task)` behaves serially if `id == notAThread`
- → Calls to `executeTask()` are a *suggestion* for parallelism, depending on the current state of the thread pool

```
1  class ExecutorThreadPool {
2      void obtainThread(threadID& id);
3
4      void executeTask(threadID id, std::function<void()>& f);
5
6      void waitForThread(threadID id);
7
8      void returnThread(threadID id);
9  }
```

ExecutorThreadPool Optional Parallelism (2/2)

```
1  void obtainThread(threadID& id) {
2      std::lock_guard<std::mutex> lk(m_mutex);
3      if (threadPool.empty()) {
4          id = ExecutorThreadPool::notAThread;
5      } else {
6          FunctionExecutorThread* t = threadPool.front();
7          threadPool.pop_front();
8          occupiedThreads.push_back(t);
9          id = t->get_id();
10     }
11 }
12
13 void executeTask(threadID id, std::function<void()>& f) {
14     if (id == ExecutorThreadPool::notAThread) {
15         f();
16     } else {
17         std::lock_guard<std::mutex> lk(m_mutex);
18         for (FunctionExecutorThread* t : occupiedThreads) {
19             if (t->get_id() == id) { t.sendRequest(f); break; }
20         }
21     }
22 }
```

AsyncGenerator Optional Parallelism

Can use thread pool rather than directly using a `FunctionExecutorThread`

→ If all threads in the pool are busy, execute the function serially instead

```
1  template <class Object>
2  class AsyncGenerator {
3
4      //Create a generator from a function and its arugments
5      template <class Function, class... Args>
6      AsyncGenerator(Funcion& f, Args&...args) {
7          auto boundF = std::bind(f, args..., std::ref(*this));
8
9          ExecutorThreadPool& pool = getThreadPool();
10         if (pool.allThreadsBusy()) {
11             boundF();
12         } else {
13             pool.addTask(f);
14         }
15     }
16 }
```

Cooperative Parallelism

With simultaneous clients of `ExecutorThreadPool`, some tasks should be given **priority**.

→ Some tasks are more **coarse-grained**, offer more potential speed-up

→ Processing certain tasks may expose more parallelism and thus should be executed first

Solution: `pool.addPriorityTask()`

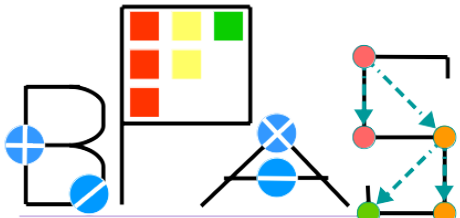
→ If there are idle threads, priority task behaves as normal task

→ If there are no idle threads:

1. temporarily allow over-subscription and spawn a new **priority thread**
2. assign the priority task to the new priority thread
3. the next thread returned to the pool is **retired** to recover a state without over-subscription

→ Do not spawn a priority thread if all current threads are priority threads

Thank you!



www.bpaslib.org

Questions?

ExecutorThreadPool Singleton

→ To avoid *over-subscription*, a program should not use multiple thread pools

→ All areas of code should share the same thread pool

→ Use a classic singleton pattern

```
1  class ExecutorThreadPool {
2
3      private:
4      //pool size defaults to 1 less than hardware allows,
5      //the main thread counts as 1
6      ExecutorThreadPool(int nthreads =
7      std::thread::hardware_concurrency() - 1;);
8
9      public:
10     static ExecutorThreadPool& getThreadPool() {
11         static ExecutorTreadPool pool;
12         return pool;
13     }
14 }
```

ExecutorThreadPool: addTask

```
1  void addTask(std::function<void()> f) {
2      std::unique_lock<std::mutex> lk(m_mutex);
3      taskPool.push_back(f);
4      lk.unlock();
5      tryPullTask();
6  }
7
8  void tryPullTask() {
9      std::unique_lock<std::mutex> lk(m_mutex);
10
11     if (!taskPool.empty() && !threadPool.empty()) {
12         FunctionExecutorThread* worker = threadPool.front();
13         threadPool.pop_front();
14
15         std::function<void()> f = taskPool.front();
16         taskPool.pop_front();
17         worker->sendRequest(f);
18     }
19
20     lk.unlock();
21 }
```

FunctionExecutorThread Callback (1/2)

How does a worker thread notify the thread pool that it has become idle?

→ A callback function inserts the thread itself back into the pool

```
1  ExecutorThreadPool(int nthreads) {
2      // for i = 0..nThreads
3      FunctionExecutorThread* t = new FunctionExecutorThread();
4      t->setCallback( std::bind(&ExecutorThreadPool::putBackThread, this) );
5  }
6
7  void putBackThread(FunctionExecutorThread* t) {
8      std::unique_lock<std::mutex> lk(m_mutex);
9      if (!taskPool.empty()) {
10         std::function<void()> f = taskPool.front();
11         taskPool.pop_front();
12         worker->sendRequest(f);
13     } else {
14         threadPool.push_back(t);
15     }
16     lk.unlock();
17     m_cv.notify_all(); //notify waitForThreads()
18 }
```

FunctionExecutorThread Callback (2/2)

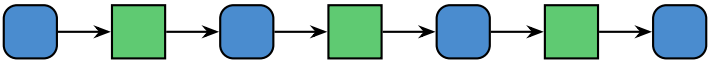
How does a worker thread notify the thread pool that it has become idle?

→ A callback function inserts the thread itself back into the pool

```
class FunctionExecutorThread {

    std::function<void(FunctionExecutorThread*)> cb;

    void eventLoop() {
        std::function<void()> task;
        while(requestQueue.getNextObject(task)) {
            task();

            if (cb) cb((FunctionExecutorThread*) this);

            std::unique_lock<std::mutex> lk(m_mutex);
            bool notify = requestQueue.streamEmpty();
            lk.unlock();
            if (notify) m_cv.notify_all();
        }
    }
}
```

Pipeline

- → A sequence of stages where the output of one stage is used as the input to another
- → Two consecutive stages form a producer-consumer pair
- → Internal stages are both producer and consumer
- → Typically, a pipeline is constructed statically through code organization
- → Pipelines can be created dynamically and implicitly with `AsyncGenerators` and the callstack



Pipelines with AsyncGenerator

```
1  void intSequence(AsyncGenerator<int>& prevStage, AsyncGenerator<int>&
       nextStage) {
2      int i;
3      while(prevStage.getNextObject(i)) {
4          nextStage.generateObject(i);
5      }
6  }
7
8  std::vector<int> A = {1,2,3,4,5,6,7,8,9};
9  AsyncGenerator<int> stageOne(A);
10
11 AsyncGenerator<int>   stageTwo(intSequence, stageOne);
12 AsyncGenerator<int> stageThree(intSequence, stageTwo);
13 AsyncGenerator<int>  stageFour(intSequence, stageThree);
14
15 //consume from last stage of pipeline
16 int i;
17 while(stageFour.getNextObjext()) {
18     std::cout << i << std::endl;
19 }
```