Stream Parallelism in Multi-Cores

PhD Course of Perspectives in Parallel Programming

Dalvan Griebler
Outline

❖ Introduction
❖ Programming Interfaces for Multi-Cores
❖ Expressing Stream Parallelism (TBB vs FastFlow)
❖ Application
❖ Performance Evaluation
❖ Conclusions
❖ References
Introduction

❖ This presentation is about
  ➢ Stream Paradigm
  ➢ Multi-Core Programming Interfaces
  ➢ Multi-Core applications
  ➢ Stream Parallelism
  ➢ FastFlow vs TBB usage
  ➢ Performance evaluation
Introduction

Stream Paradigm

- Stream processing
- Composed by stream (set of data) and kernels (sequential operations on each element).
- A stream can be bounded or unbounded
Introduction

❖ **Stream Paradigm**

➤ Stream systems

- Structured parallel programming by using a functional and building block style
- Kernels are sequential compute units
- In terms of expressiveness, they try to avoid from user lock mechanisms
- Communication through FIFO queue channels

```plaintext
A[i]

for(i=0;i<N;i++)

B[i]

sum[i] = A[i]+B[i]

stdout(sum[i])
```
Stream Parallelism

❖ Skeletons for stream parallelism
  ➢ Pipeline
  ➢ Farm
  ➢ LoopBack

S1 ➔ S2 ➔ Sn ➔ W1 ➔ Wn ➔ Sch. ➔ Gether

LoopBack
Introduction

❖ Other stream paradigms
  ➢ Reactive systems

=\text{A1}+\text{A2}
Introduction

❖ Other streams paradigms

➢ DataFlow systems
  ■ Stream and DataFlow are very similar
  ■ In general, streams are able to express DataFlow graphs
  ■ DataFlow tries to explore opportunities of parallelism when data dependencies are explicitly declare in the code.
  ■ On the architecture view: the behaviour is indeterministic
  ■ On the programming view: you will have a graph with data operations dependencies.
  ■ In general, the systems are implemented using data structures
Introduction

❖ Other stream paradigms
  ➢ Data Streams
    ■ Data transfer
    ■ Centralized Broker
    ■ Fault Tolerance
    ■ Applies filters to real time data

Apache Storm

Google Cloud DataFlow

Spark

samza
Introduction

❖ **Why should I use stream paradigm**
  ➢ To simplify parallel programming
  ➢ To avoid locks
  ➢ To increase the throughput of my application
  ➢ Implement efficient and expert runtimes
  ➢ Enforce structured programming
Introduction

❖ Multi-Core context

➢ Scenarios
➢ Limitations
➢ Problems
➢ What are the current applications?
Introduction

- Multi-Core context

Workstations

Supercomputers

- Hardware?
- Computations?
- Stream sources?
Introduction

❖ Multi-Core context

Hardware?

- 8 Cores HT with 3.0 GHz
- 5750 CUDA cores with 700MHz
- 61 cores with 1.3 GHzv
- 32GB
Introduction

❖ Multi-Core context

Computations?
Introduction

❖ Multi-Core context
Introduction

❖ Multi-Core context

HP ProLiant DL580 Gen9 Server

- 4 Processors (Intel E7-8890) with a total of 144 threads
- 4TB RAM capacity (NUMA)
- 10 HD capacity SAS/SSD/SATA
- 10GB Network
- Slots for Accelerators

Intel E7-8890

18 cores with HT 2.5GHz
Introduction

❖ Multi-Core context
Introduction

❖ Multi-Core context
Programming Interfaces for Multi-Cores

- OpenMP
- Cilk
- TBB
- FastFlow
Programming Interfaces for Multi-Cores

- OpenMP
Programming Interfaces for Multi-Cores

❖ OpenMP

```c
void process_in_parallel() {  
    #pragma omp parallel
    #pragma omp single
    {
        int x = 1;
        ...
        for (int i = 0; i < T; ++i) {
            #pragma omp task shared(x, ...) depend(out: x) // T1
            preprocess_some_data(...);
            #pragma omp task shared(x, ...) depend(in: x) // T2
            do_something_with_data(...);
            #pragma omp task shared(x, ...) depend(in: x) // T3
            do_something_independent_with_data(...);
        }
    } // end omp single, omp parallel
}
```

T1 has to be completed before T2 and T3 can be executed.

T2 and T3 can be executed in parallel.
Programming Interfaces for Multi-Cores

Intel CilkPlus
- Work Stealing model
- Recursive parallelism
- Keywords
  - cilk_for
  - cilk_spawn
  - cilk_sync
- Runtime Support
  - #pragma cilk grainsize = expression
  - cilk::reducer_opadd<int> sum;
  - System Functions (set and get threads info.)
bool done = false;
int iter_counter = 0;
cilk_pipe_while(!done) { // Each iteration starts executing in Stage 0.
    int i = iter_counter++;
    done = stage0(i);
    ilk_stage(1); // Advance to Stage 1 (parallel stage)
    stage1(i);
    ilk_stage_wait(2); // Advance to Stage 2 (serial stage)
    stage2(i);
    ilk_stage(3); // Advance to Stage 3 (parallel stage)
    stage3(i);
    ilk_stage_wait(4); // Advance to Stage 4 (serial stage)
    stage4(i);
}
Programming Interfaces for Multi-Cores

Intel TBB (Threading Building Blocks)
- Work Stealing Pattern
- Provides Several Patterns (for loop, map, reduce, pipeline)
- Provides recursive parallelism
- There is no real stream
- Threads cross over all stages.
- Only Multi-core
- Flow Graph
# Programming Interfaces for Multi-Cores

**Intel TBB (Threading Building Blocks)**

<table>
<thead>
<tr>
<th></th>
<th>task graphs</th>
<th>pipeline / parallel_pipeline</th>
<th>flow graph</th>
<th>pipeline / parallel_pipeline</th>
<th>flow graph</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Expressiveness</strong></td>
<td>Can express acyclic dependency graphs.</td>
<td>Can express linear pipelines.</td>
<td>Can express acyclic dependency graphs as well as acyclic and cyclic messaging graphs.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Ease-of-use</strong></td>
<td>Requires low-level bookkeeping code and explicit spawning of tasks.</td>
<td>A concise, type safe interface</td>
<td>More verbose than parallel_pipeline, but does not require explicit book-keeping or task spawning.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Persistence</strong></td>
<td>Graphs are executed destructively. Cannot be re-executed.</td>
<td>Can be executed multiple times.</td>
<td>Can be executed multiple times.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Scheduling</strong></td>
<td>Uses spawned tasks, which optimize for cache locality.</td>
<td>Uses spawned tasks, which optimize for cache locality.</td>
<td>Uses enqueued tasks, which are fairness-oriented and support fire-and-forget use cases.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Performance</strong></td>
<td>Very low overhead since it is built directly on tasks and is executed destructively.</td>
<td>Overhead is comparable to flow graph.</td>
<td>Overhead is comparable to pipeline and parallel_pipeline.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Graph object](image)

**Functional**
- source_node
- continue_node
- function_node
- multioutput_function_node

**Buffering**
- buffer_node
- queue_node
- priority_queue_node
- sequencer_node

**Split / Join**
- queueing join
- reserving join
- tag matching join
- split_node
- or_node

**Other**
- broadcast_node
- write_once_node
- overwrite_node
- limiter_node
Programming Interfaces for Multi-Cores

❖ Intel TBB (Threading Building Blocks)

➢ Pipeline vs Flow Graph Options

```cpp
#include <tbb/pipeline.h>

int main() {
    tbb::pipeline pipeline;
    MyInputFilter input_filter( input_file );
    pipeline.add_filter( input_filter );
    MyTransformFilter transform_filter;
    pipeline.add_filter( transform_filter );
    MyOutputFilter output_filter( output_file );
    pipeline.add_filter( output_filter );
    pipeline.run( nthreads*4 );
    return 0;
}
```
Programming Interfaces for Multi-Cores

FastFlow

- Similar to TBB, but a framework for research
- Provides high-level and low-level patterns
- New patterns can be built
- Communication with lock-free queues
- Provides C++ template for stream-based skeletons (farm and pipeline)
- Threads do not cross stages
Programming Interfaces for Multi-Cores

❖ FastFlow

➢ Some Skeletons

```c
Stage0 s0; Stage1 s1; Stage2 s2; Stage3 s3;

ff_Pipe<long> pipe1(s0, s1);
pipe1.wrap_around();

ff_Pipe<long> pipe2(pipe1, s2);
pipe2.wrap_around();

ff_Pipe<long> pipe(pipe2, s3);
pipe.wrap_around();

if (pipe.run_and_wait_end() < 0) error("running pipe");
```

Fancy pipeline

```c
ff_Farm<long> farm(F, 3);
farm.remove_collector(); // removes the default collector
// the scheduler gets in input the internal load-balancer
Sched S(farm.getlb());
farm.add_emitter(S);
// adds feedback channels between each worker and the scheduler
farm.wrap_around();
// creates a node from a function
ff_node_F<long> stage(Stage0);
// creates the pipeline
ff_Pipe<> pipe(stage, farm);
if (pipe.run_and_wait_end() < 0) error("running pipe");
```
Programming Interfaces for Multi-Cores

Programming Interfaces Summary

- TBB
- FastFlow
- OpenMP
- Cilk
Expressing Stream Parallelism (TBB vs. FastFlow)

❖ Building the compute units

**FastFlow**

```c
struct Stage0: ff_monode_t<long> {
    int svc_init() { counter=0; return 0; }
    long *svc(long *task) {
        if (task==nullptr) {
            for (long i=1;i<=NUMTASKS;++i)
                ff_send_out((void*)i);
            return GO_ON;
        }
        printf("Stage0 has got task %ld\n", (long)task);
        ++counter;
        if (counter == NUMTASKS) return EOS;
        return GO_ON;
    }
    long counter;
};
```

```c
struct Stage2: ff_monode_t<long> {
    long *svc(long *task) {
        // sends back even tasks less than ...
        if (((long)task <= (NUMTASKS/2))
            ff_send_out_to(task, 0);
        else ff_send_out_to(task, 1);
        return GO_ON;
    }
};
```

**TBB**

```c
class MyInputFilter: public tbb::filter {
public:
    MyInputFilter( FILE* input_file_ );
    ~MyInputFilter();
private:
    FILE* input_file;
    TextSlice* next_slice;
    /*override*/ void* operator()(void*);
};
MyInputFilter::MyInputFilter( FILE* input_file_ ) :
    filter(serial_in_order), input_file(input_file_),
    next_slice( TextSlice::allocate( MAX_CHAR_PER_INPUT_SLICE ) ) {};
MyInputFilter::~MyInputFilter() {next_slice->free();}
void* MyInputFilter::operator()(void*) {
    // Read characters into space that is available in the next slice.
    size_t m = next_slice->avail();
    size_t n = fread( next_slice->end(), 1, m, input_file );
    if (!n && next_slice->size()==0 ) {
        // No more characters to process
        return NULL;
    }else{
        // Have more characters to process.
        TextSlice& t = *next_slice;
        next_slice = TextSlice::allocate( MAX_CHAR_PER_INPUT_SLICE );
        char* p = t.end()+n;
        if (n==m ) {
            // Might have read partial number. If so, transfer characters
            while( p>t.begin() & isdigit(p[-1]) )
                --p;
            next_slice->append( p, t.end()+n );
        }t.set_end(p);
        return &t;
    }
}
```
Expressing Stream Parallelism (TBB vs. FastFlow)

❖ Building the stream computation

FastFlow

```cpp
ff_Farm<long> farm(F, 3);
farm.remove_collector(); // removes the default collector
// the scheduler gets in input the internal load-balancer
Sched S(farm.getlb());
farm.add_emitter(S);
// adds feedback channels between each worker and the scheduler
farm.wrap_around();
// creates a node from a function
ff_node_F<long> stage(Stage0);
// creates the pipeline
ff_Pipe<> pipe(stage, farm);
if (pipe.run_and_wait_end() == 0) error("running pipe");
```

TBB

```cpp
tbb::flow::graph g;
tbb::flow::limiter_node limiter( g, nthreads*4 );
tbb::flow::sequencer_node< TextSlice * > sequencer(g, sequencer_body() );
tbb::flow::source_node input( g, MyInputFilter(input_file), false );
tbb::flow::function_node transform( g, tbb::flow::unlimited, MyTransformFilter() );
tbb::flow::function_node output( g, tbb::flow::serial, MyOutputFilter( output_file ) );
tbb::flow::make_edge( input, limiter );
tbb::flow::make_edge( limiter, transform );
tbb::flow::make_edge( transform, sequencer );
tbb::flow::make_edge( sequencer, output );
tbb::flow::make_edge( output, limiter.decrement );
input.activate();
go.wait_for_all();
```
Application

❖ Sobel Application Picture

Read Images → Apply Sobel → Write Images
Application

Possible Strategies (Which is the best?)

- Read Images
  - Apply Sobel
    - Write Images
  - Apply Sobel
    - Write Images

- Read Images
  - Apply Sobel
    - Write Images
  - Apply Sobel
    - Write Images

- Read Images
  - Apply Sobel
    - Write Images
  - Apply Sobel
    - Write Images

- Read Images
  - Apply Sobel
    - Write Images
  - Apply Sobel
    - Write Images
Application

Possible Strategies (Which is the best?)
Application

Possible Strategies (Which is the best?)

Approach 1 (Farm)
- Read Images
  - Apply Sobel
    - Write Images

Approach 2 (Pipe)
- Read Images
  - Apply Sobel
- Apply Sobel
  - Write Images
Applications (Code Examples)

❖ Sobel Application (pseudo source code)

```c
//global declaration
int main(int argc, char *argv[]){
    //open directory ...
    DIR *dptr = opendir(...);
    struct dirent *dfptr;
    while((dfptr = readdir(dptr)) != NULL){
        //preprocessing
        if (file_extension == "bmp"){
            //Reads the image ...
            tot_img++;
            //Reads the image ...
            image = read(filename,height,width);
            //Applies the Sobel ...
            new_image=sobel(image,height,width);
            //Writes the image ...
            write(new_image,height,width);
        }else{
            tot_not++;
        }
    }
    //end of stream computing
    return 0;
}
```
Applications (Code Examples)

❖ Sobel (farm)

```c
struct Stream {
    ...
};

struct Stage0: ff_node_t<Stream> {
    Stream *svc(Stream *){
        DIR *dptr = opendir(...);
        struct dirent *dfptr;
        while((dfptr = readdir(dptr)) != NULL){
            //preprocessing
            if (file_extension == "bmp"){
                //Reads the image ...
                tot_img++;
                Stream *stream = new Stream(...);
                ff_send_out(stream);
            }else{
                tot_not++;
            }
        } //end of
        return EOS;
    }
};

Stream *Stage1(Stream *in,ff_node*const){
    //Reads the image ...
    image = read(in->filename,height,width);
    //Applies the Sobel ...
    new_image=sobel(image,height,width);
    //Writes the image ...
    write(new_image,height,width);
    delete in;
    return (Stream*)GO_ON;
}
```

TBB

```c
struct Stream {
    ...
};

class Stage0: public tbb::filter {
    public:
        ...; void *operator()(void *); ....;
};

Stage0::Stage0(...):tbb::filter(serial_in_order),...){
    void * Stage0::operator()(void *) {
        DIR *dptr = opendir(...);
        struct dirent *dfptr;
        while((dfptr = readdir(dptr)) != NULL){
            //preprocessing
            if (file_extension == "bmp"){
                //Reads the image ...
                tot_img++;
                Stream *stream = new Stream(...);
                return stream;
            }else{
                tot_not++;
            }
        } //end of
        return NULL;
    }
};

class Stage1: public tbb::filter {
    public:
        Stage1(); void * operator()( void *input );
};

Stage1::Stage1(): tbb::filter(parallel){
    void *Stage1::operator()(void *input){
        Stream *in = static_cast<Stream*>(input);
        //Reads the image ...
        image = read(in->filename,height,width);
        //Applies the Sobel ...
        new_image=sobel(image,height,width);
        //Writes the image ...
        write(new_image,height,width);
        delete in;
        return NULL;
    }
```

FastFlow
Applications (Code Examples)

❖ Sobel (farm)

```c
int main(int argc, char const *argv[]){
    Stage0 S0(...);
    ff_Farm<Stream> S1(Stage1,workers);
    S1.add_emitter(S0);
    S1.set_scheduling_on-demand();
    S1.remove_collector();
    if (S1.run_and_wait_end()<0) { // e.
        error("Running farm\n");
        return -1;
    }
    return 0;
}
```

```
int main(int argc, char const *argv[]){
    tbb::pipeline pipeline;
    Stage0 S0(....);
    pipeline.add_filter(S0);
    Stage1 S1;
    pipeline.add_filter(S1);
    pipeline.run(workers);
}
```
Applications (Code Examples)

❖ Sobel (pipe)

```cpp
struct Stream {
    ...;
};
struct Stage0: ff_node_t<Stream> {
    Stream *s = Stream(*);
    DIR *dp = opendir(...);
    struct dirent *dpf;
    while ((dpf = readdir(dp)) != NULL) {
        // preprocessing
        if (file_extension == "bmp") {
            // Reads the image...
            tot_img++;
            // Reads the image...
            image = read(filename, height, width);
            Stream *stream = new Stream(...);
            return stream;
        } else {
            tot_not++;
        }
    }
    // end of return EOS;
}
Stage0(Stream *in, ff_node<const>
    // Applies the Sobel...
    in->new_image = sobel(in->image, in->height, in->width);
    return in;
};
Stage1(Stream *in, ff_node<const>
    // Applies the Sobel...
    in->new_image = sobel(in->image, in->height, in->width);
    return in;
};
Stage2(Stream *in, ff_node<const>
    // Writes the image...
    write(in->new_image, in->height, in->width);
    delete in;
    return (Stream*)GO_ON;
```
Applications (Code Examples)

❖ Sobel (pipe)

```c
int main(int argc, char const *argv[]){
    Stage0 S0(...);
    ff_Farm<Stream> S1(Stage1,workers);
    S1.add_emitter(S0);
    S1.set_scheduling_on_demand();
    S1.remove_collector();
    struct MultiInput_STAGE: ff_minode_t<Stream> { 
        Stream *svc(Stream *stream) {
            return Stage2(stream, this);
        }
    };
    MultiInput_STAGE S2;
    ff_Pipe<> pipe(S1,S2);
    if (pipe.run_and_wait_end()<0) { // executes the error
        error("Running pipeline\n");
        return -1;
    }
    return 0;
}
```

FastFlow

```c
int main(int argc, char const *argv[]){
    tbb::pipeline pipeline;
    Stage0 S0(...);
    pipeline.add_filter(S0);
    Stage1 S1;
    pipeline.add_filter(S1);
    Stage1 S2;
    pipeline.add_filter(S2);
    pipeline.run(workers);
}
```

TBB
Performance Evaluation

Environment of Tests

- Machine
  - Intel(R) Core(TM) i7-4700HQ CPU @ 2.40GHz, 4 Cores and 4 Logical
  - 16GB memory
  - 240GB SSD hard disk
- Default compilation
  - g++-5.2.1 -std=c++1y -O3 -finline-functions -DNDEBUG
- FastFlow
  - Blocking mode
  - 2015 version
- TBB
  - 2015 version
- Input
  - 8000 images of 1.44MB
Performance Evaluation

Metrics
- Number of System Threads
- Completion Time
- Performance (Speed-Up and Efficiency)
- Memory Usage
- Cache-misses
- Context Switches
- Latency
- Throughput
- Energy Consumption
Performance Evaluation

- Number of system threads

![System Threads Graph](image)
Performance Evaluation

❖ Completion Time

![Graph: Execution Times Sobel Application](image-url)
Performance Evaluation

- Performance

![Performance Chart]

- Speed-Up
- Cores
- TBB (farm)
- FF (farm)
- TBB (pipe)
- FF (pipe)
Performance Evaluation

- Efficiency

![Graph showing efficiency vs cores](image-url)
Performance Evaluation

Memory Usage

[Graph showing Memory Usage (VmHWM) with different lines representing TBB, FF, TBB (pipe), and FF (pipe) across 8 cores.]
Performance Evaluation

❖ Cache-misses
Performance Evaluation

❖ Context Switches (Voluntary)
Performance Evaluation

❖ Context Switches (NonVoluntary)
Performance Evaluation

❖ Throughput
Performance Evaluation

❖ Latency
Performance Evaluation

- Energy Consumption (from socket)
Performance Evaluation

- Energy Consumption (from memory controller)
Performance Evaluation

- Energy Consumption (only the cores)
Conclusions

❖ Stream Parallelism Challenges
❖ Machine Limitations
❖ Energy Consumption
❖ Scalability limitations
❖ Cilk and OMP are not suitable for stream parallelism
❖ In pipeline computations is expected to reduce memory consumption.
❖ Perspectives for accelerators that are part of the multi-core context. Only FastFlow is taking in to account.
❖ Flow Graph must be explorer (TBB states that the performance is comparable to pipeline)
Bibliography

- OpenMP <http://openmp.org/>
- FastFlow <http://calvados.di.unipi.it/>
- TBB <https://www.threadingbuildingblocks.org/>
- Cilk <https://www.cilkplus.org/>
Thank you!

Questions?

Dalvan Griebler