Distributed systems: paradigms and models

μMDF: A minimal Macro Data Flow framework

Andrea Maggiordomo
mggndr89[at]gmail.com

2016

1 Introduction

The project consists in the design and implementation of μMDF, a parallel framework supporting the execution of Macro Data Flow graphs. The framework is implemented in C++ and relies only on the standard library support for concurrency introduced in the 2011 revision of the language.

2 Framework overview

This section covers the design choices made for the project from a user's perspective, and then describes how a μMDF program can be defined using the framework interface.

2.1 Features

From a user perspective, the framework must allow the definition of a Macro Data Flow graph that describes the computation through the dataflow model. This is done by instantiating a `mdf::Graph` object to which the user can add instruction nodes and connections (where connections link the output of an instruction to the input of another).

The generation of the data that must be processed through the MDF graph is delegated to a *streamer* object that feeds to the graph a stream of input tokens. The input tokens generated by a streamer will trigger one or more fireable instructions in the graph and start the computation on a new graph instance. Streamer objects are implemented as concepts: they are simply objects that define a `Next()` method with a suitable signature. A MDF graph can have more than one entry point for the data generated by a streamer.

Once the data has flowed through the graph, it is processed by a *drainer* object. A drainer is a user defined callable object that is invoked on the output token produced by an interpretation and can perform "clean-up" duties. Note that only MDF graphs with exactly one exit point are supported.

A MDF interpreter is an instance of the `mdf::Mdf<D>` template class, where the template type `D` is the drainer type. An interpreter is constructed by passing an instance of the MDF graph which will be used as model, the number of workers that the interpreter will use (that is, the parallelism degree of the interpreter), and a `unique_ptr` to a drainer type instance. Once the interpreter has been constructed, a user will simply start the interpretation passing a streamer object to the interpreter and waiting for its completion.

2.2 Graph definition

A MDF graph is initially defined as empty:

```
mdf::Graph g{};
```

Once the instance has been constructed, an instruction node can be added by calling the `AddInstruction` method, a variadic template method which takes as input a callable type, and a list of `mdf::ParamDecl<T>` objects that tell the framework the type and name of the parameters taken by the function. The type `T` is used by the framework to cast each token to the correct type, while the parameter names are required in order to forward the output of a node to a specific destination. Supposing we had a function `int Foo(int,double)`, we could add it to the graph as

```
mdf::NodeId idFoo = g.AddInstruction(
    &Foo, mdf::ParamDecl<int>{"p1Foo"}, mdf::ParamDecl<double>{"p2Foo"});
```

Note that the order of the parameter types is relevant. The method returns an instruction node identifier which can be used to specify connections. For example, assuming we also had a function `int Bar(int)`:

```
mdf::NodeId idBar = g.AddInstruction(&Bar, mdf::ParamDecl<int>{"p1Bar"});
g.Connect(idFoo, idBar, "p1Bar");
```

would connect the output of `Foo` – which is an `int` – to the first parameter of `Bar`.

Another way of connecting two instructions is by declaring a dependency between them. This is useful if an instruction must be executed after another has run because a parameter is modified by side effect. This mechanism can be employed to obtain some degree of data parallelism by passing pointers and declaring dependencies between instruction nodes. The statement

```
g.DeclareDependency(idFoo, idBar);
```

would instruct the framework to fire `Bar` only after the execution of `Foo` has taken place. Dependencies have no name.

2.3 Streamer concept

A streamer object must implement a `Next()` method with the following signature:

```
vector<mdf::InputTokenContainer> Next();
```

The `InputTokenContainer` type is used by the framework to assign an input token (that is, a token generated by the streamer) to a macro instruction; it is constructed from a `mdf::NodeId` and a `string` (the destination instruction and parameter name), and a `mdf::TokenHandle` object, which is an erasure type required by the interpreter to handle heterogeneous types. One can obtain a `TokenHandle` object by wrapping a value `v` of type `T` with the helper function `WrapValue`:

```
auto handle = mdf::WrapValue<T>{v};
```

As an example, to feed input parameters to the `Foo` instruction one could do the following:

```
vector<InputTokenContainer> Streamer::Next() {
  vector<InputTokenContainer> input;
  if (!endOfInput()) {
    input.emplace_back(InputTokenContainer{
        idFoo, "p1Foo", mdf::WrapValue<int>(NextInt())});
    input.emplace_back(InputTokenContainer{
        idFoo, "p2Foo", mdf::WrapValue<double>(NextDouble())});
  }
  return input;
}
```

By convention returning an empty vector tells the framework that the input stream has no more incoming data.

2.4 Drainer concept

A drainer object is simply a callable object that can receive as input a `TokenHandle` parameter. Any token produced by an instruction node that has no outgoing connections and/or dependencies (*output* nodes) is delivered to the drainer. A MDF graph must have a single output node, but this requirement is not enforced. Accesses to the drainer are protected against race conditions, so it is safe to perform operations on critical resources like writing to a file. Unfortunately, when dealing with an output token, one must downcast explicitly. Suppose we want to drain the output produced by the `Bar` function:

```
void Drainer::operator()(TokenHandle token) {
 mdf::ValueHandle<int> out = dynamic_pointer_cast<mdf::Value<int>>(token);
  if (out) {
    int r = out->GetValue();
    // Do something with r
  } else ... // Downcast failed: something is wrong with the graph definition
}
```

The `ValueHandle<T>` type is a pointer type to a `Value<T>` object, the actual value container used by the framework.

2.5 Starting an interpreter

The MDF interpreter constructor takes as arguments a graph instance, the parallelism degree of the interpreter, and a `unique_ptr` to a drainer object:

```
mdf::Mdf<Drainer> interpreter{g, tn, unique_ptr<Drainer>{new Drainer}};
```

Once the interpreter has been constructed, it is run by calling the `Start()` method, passing a unique pointer to a streamer object as parameter (the method is templated on the streamer type):

```
unique_ptr<Streamer> streamer{new Streamer{idFoo}};
streamer = engine.Start(move(streamer));
```

The `Start()` method is blocking, and it returns once the computation has been completed.

3 The MDF interpreter internals

This section offers a quick overview of the internal architecture of the MDF interpreter.

3.1 Graph instancing

When a streamer generates new input data, a new `mdf::GraphHandle` object is constructed in order to keep track of the state of the new graph instance. A `GraphHandle` object contains an `instanceId` field, a copy of the original MDF graph and a thread-safe `mdf::ConcurrentMap` that stores the state of each instruction node in the graph (the tokens available to an instruction, the number of instructions it depends on that have already been executed, and a boolean `fired` flag). The `ConcurrentMap` is implemented as a hash table, with bucket-level lock granularity.

3.2 Task scheduling

The `Mdf` interpreter features a global task queue, as well as local queues assigned to each worker thread. When new input is available from the streamer, any instruction that can be immediately fired is scheduled on the global queue; any instruction fired by a worker thread is instead scheduled on its own local queue.

Since a streamer might generate input data at a rate much higher than the one at which the interpretations are performed by the workers, the global queue is bounded in size. If the queue reaches its maximum capacity, the streamer will block until a sufficient amount of tasks have been removed.

Job stealing is employed in an attempt to achieve some kind of load balancing: when a thread can't find any task neither in its local nor in the global queue, it attempts to steal a task from some other thread. Only if it fails in such an attempt it actually yields the execution.

It is worth mentioning that in this case yielding is a rather aggressive strategy, and if the shortage of tasks is prolonged suspending the thread for a brief amount of time could be beneficial to the other busy workers.

4 Example files

A few examples are provided in order to show how the framework can be used:

`hello.cpp` The complete implementation of the example used in this document to describe the framework features.

`sinloops.cpp` This example instantiates an artificial MDF graph used to test the framework and perform the experiments. The graph contains six nodes that simulate a somewhat expensive computation by iteratively computing the sine of a parameter. The executable takes as argument the number of items streamed, the parallelism degree and the number of iterations performed at each node.

`mandelbrot.cpp` This example generates a visual representation of the Mandelbrot set based on the histogram of the number of iterations required to reject each point. Each pixel coordinate is mapped to a complex number and membership to the Mandelbrot set is rejected up to a fixed number of iterations. The computation is organized as follows: the image is split into a sequence of blocks which are generated by the streamer; each block is further divided into slices, with each slice (composed by a number of block lines) handled by a MDF instruction; since the image generation requires the maximum reached iteration count to map iterations to pixel intensities, each MDF instruction returns the maximum reached iteration, which is then reduced using a "reverse binary tree" topology. To avoid dealing with boundary cases, the size of the image is a power of 2 (1024 by 1024), as well as the size of each block (64 by 64) and the number of lines in each slice (4). This means that the MDF graph is composed by an initial stage of 16 macro instructions that compute the iterations, and $\log_2(16) = 4$ stages where the maximum number of iterations of the block gets computed by reducing the output of each slice. The executable file can take the parallelism degree as argument.

`dependencies.cpp` This example uses dependencies among nodes (rather than connections) to implement the data parallel computation of a stream of matrix-vector multiplications. The graph declares an array of "worker" instructions, and the streamer assigns a slice of the input matrix to each instruction. The executable accepts as input the dimension of the matrix, the number of tasks and the number of items streamed.

Other than these files, the sequential versions of the loops and Mandelbrot examples (`seq_sinloops.cpp` and `seq_mandelbrot.cpp`) are also provided, as well as the files used to run the experiments (`*_exp.cpp`).

Since the framework is implemented as a simple set of header files the samples can be compiled directly at the command line (passing `-std=c++11` and `-pthread` as options), otherwise the `build.sh` and `build_mic.sh` scripts can be used:

`./build.sh src exe compiler` builds the source file `src` with the specified compiler (either `g++` or `icpc`) and generates the `exe` binary.

`./build_mic.sh src exe` uses `icpc` and compiles with the `-mmic` flag.

5 Experimental results

The experiments were performed using the `sinloops` and `mandelbrot` examples. Each experiment was replicated 10 times, taking the average completion time with a given parallelism degree after discarding the minimum and maximum values measured. The experiments were run both on the Xeon PHI coprocessor and on the host machine.

The `sinloops` test was performed by streaming 1000 input items and performing 50000 iterations in each node: results for completion time, speedup, scalability and efficiency are reported in figures 1 and 2.

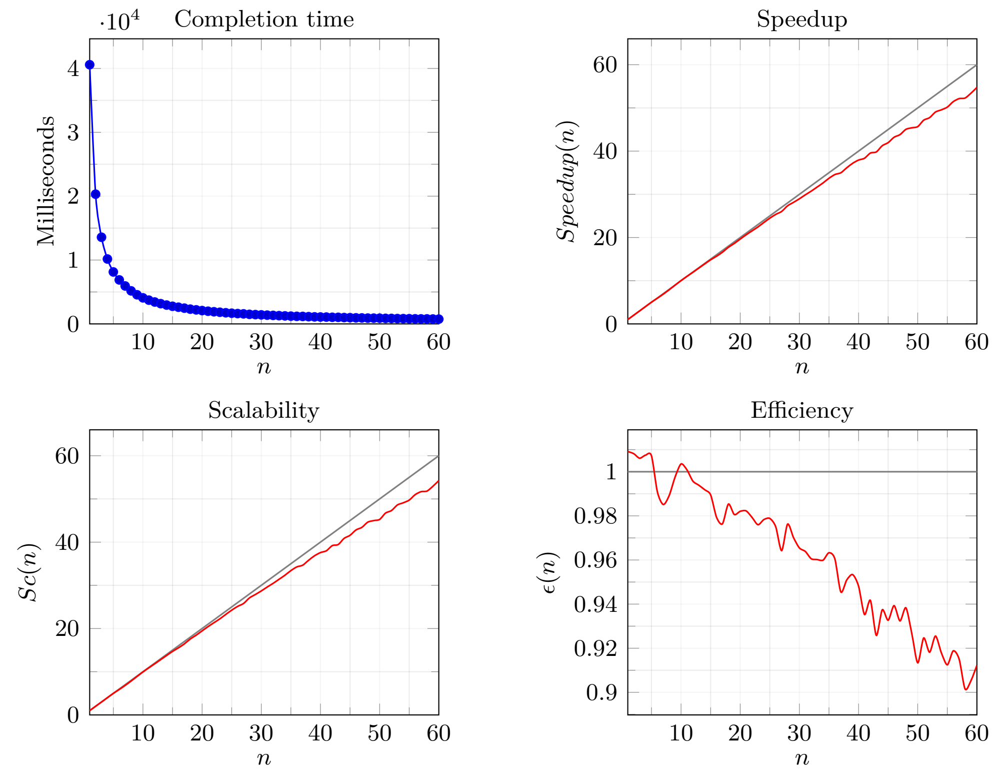

Figure 1: Results for the `sinloops` test – Xeon PHI

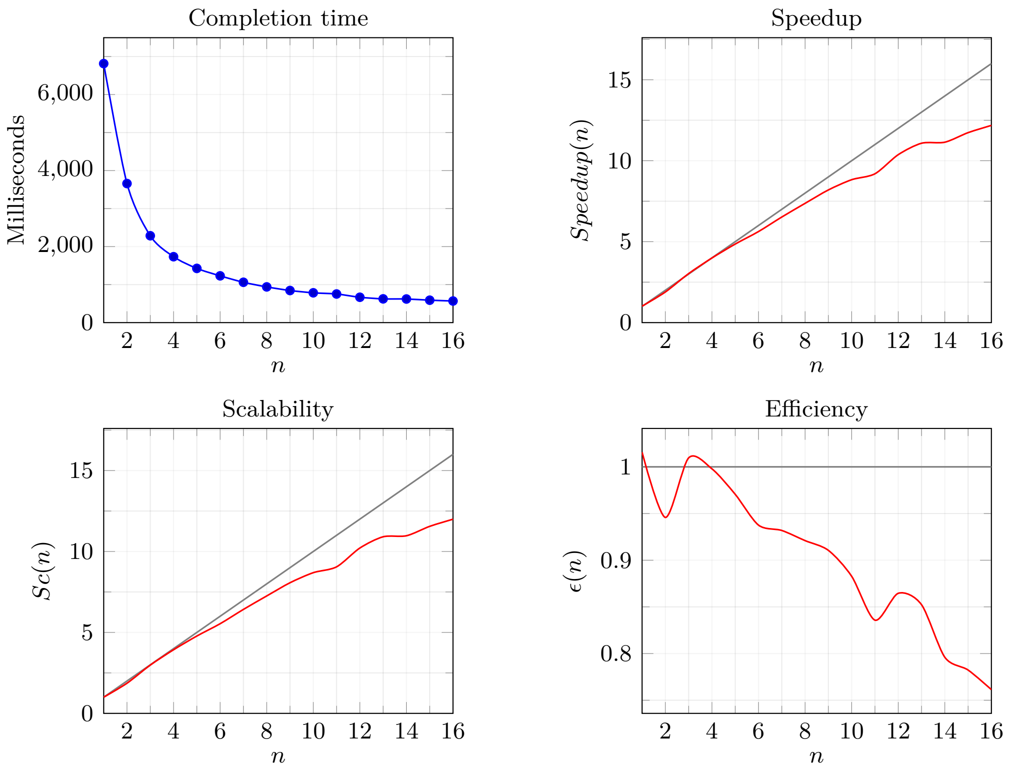

Figure 2: Results for the `sinloops` test – Host machine

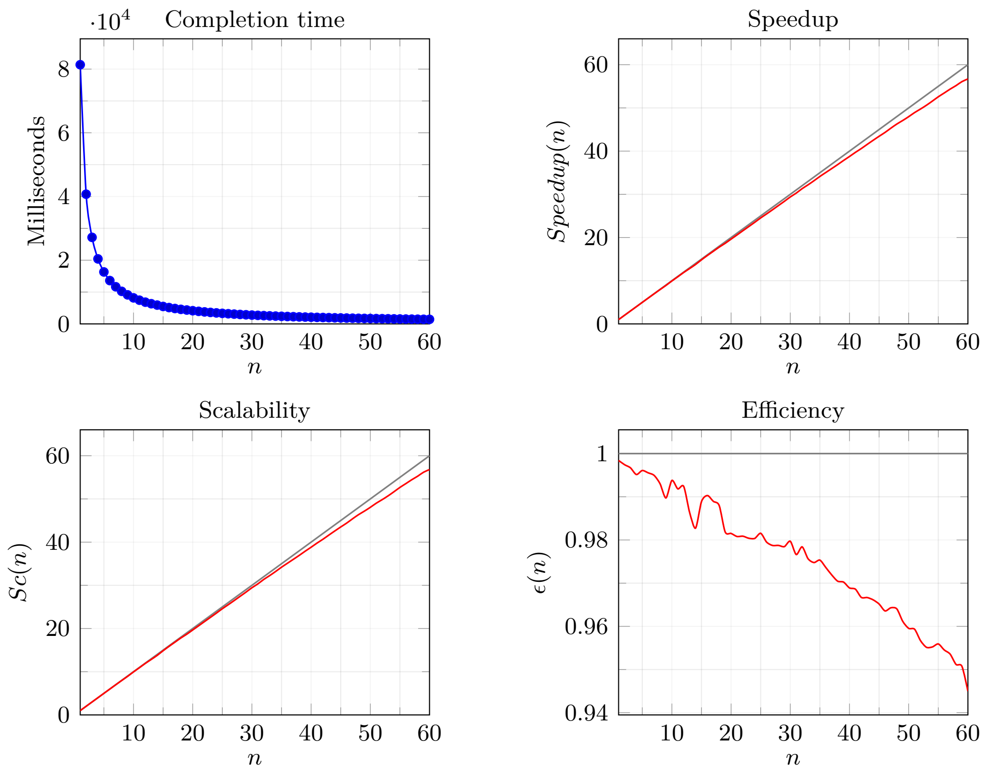

Figure 3: Results for the `mandelbrot` test – Xeon PHI

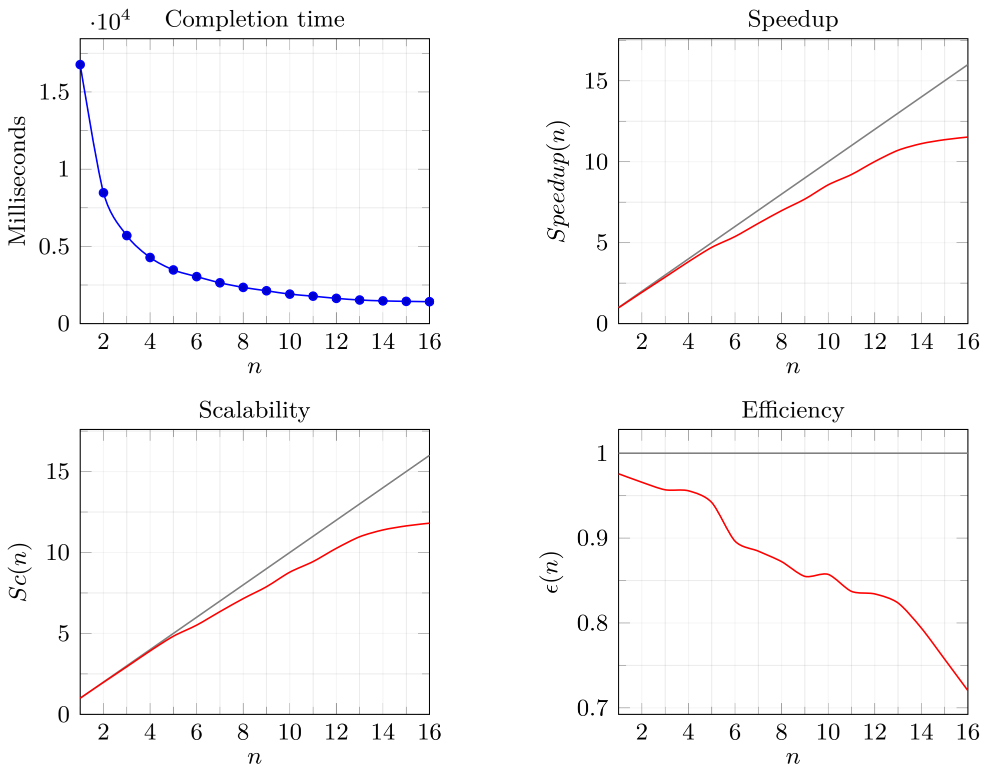

Figure 4: Results for the `mandelbrot` test – Host machine

The `mandelbrot` test employed a stream of 256 blocks to generate an image of 1024 by 1024 pixels; experimental results are reported in figures 3 and 4 and do not take into account the time required to write the image file to disk.

The plots show reasonable performance measures for the test that were run on the Xeon PHI coprocessor with good efficiency levels in both applications, while the results were more underwhelming for the tests performed on the host machine.

6 Conclusions

This report presented a minimal framework to support the execution of Macro Data Flow graphs, and showed experimental results that seem somewhat encouraging. Possible improvements could be made by performing more strict type checking on the instruction arguments in order to prevent user errors, and more importantly by checking the validity of the graph topology: a simple visit would be enough to detect macroscopic errors like cycles, multiple exit nodes, or instructions that have more incoming edges than the number of parameters they can accept.