Communicating Process Architectures in Light of Parallel Design Patterns and Skeletons

Dr Kevin Chalmers

School of Computing
Edinburgh Napier University
Edinburgh

k.chalmers@napier.ac.uk

Overview

- I started looking into patterns and skeletons when I wrote some nice helper functions for C++11 CSP
  - `par_for`
  - `par_read`
  - `par_write`
- I started wondering what other helper functions and blocks I could develop
- Which led me to writing the paper, which I've done some further thinking about
- So, I'll start with my proposals to the CPA community and add in some extra ideas not in the paper

Outline

Outline

1. Creating Patterns and Skeletons with CPA
2. CSP as a Descriptive Language for Skeletal Programs

Outline

Outline

1. Creating Patterns and Skeletons with CPA
2. CSP as a Descriptive Language for Skeletal Programs
3. Using CCSP as a Lightweight Runtime
4. Targeting Cluster Environments

Outline

Outline

Table: Mapping Catanzaro's and Massingill's view of parallel design patterns.

Catanzaro	**Massingill**
Not Covered	Finding Concurrency
Structural	Supporting Structures
Computational	Not Covered
Algorithm Strategy	Algorithm Structures
Implementation Strategy	Supporting Structures
Concurrent Execution	Implementation Mechanisms

Common Patterns Discussed in the Literature

- Pipeline (or pipe and filter).
- Master-slave (or work farm, worker-farmer).
- Agent and repository.
- Map-reduce.
- Task-graph.
- Loop parallelism (or parallel for).
- Thread pool (or shared queue).
- Single Program - Multiple Data (SPMD).
- Message passing.
- Fork-join.
- Divide and conquer.

Slight Aside - The 7 Dwarves (computational problem patterns)

- Structured grid.
- Unstructured grid.
- Dense matrix.
- Sparse matrix.
- Spectral (FFT).
- Particle methods.
- Monte Carlo (map-reduce).

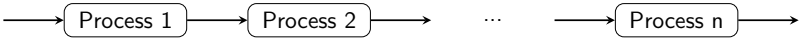

Figure: Pipeline Design Pattern.

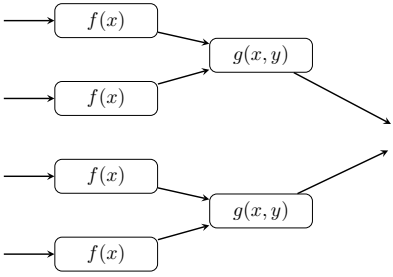

Figure: Map-reduce Design Pattern.

- Pipeline.
- Master-slave.
- Map-reduce.
- Loop parallelism.
- Divide and conquer.
- Fold.
- Map.
- Scan.
- Zip.

Data Transformation - How Functionals Think

- I'll come back to this again later
- Basically many of these ideas come from the functional people
- Everything in their mind is a data transform
- Having been to a few with functional people (Scotland has a lot of Haskellers) they see every parallel problem as a map-reduce one
  - This has real problems for scalability

Creating a Pipeline with FastFlow

```
int main()
{
    // Create a vector of two workers
    vector<ff_node*> workers = {new worker, new worker};
    // Create a pipeline of two stages and a farm
    ff_pipe<fftask_t> pipeline(new stage_1, new stage_2, new
        ff_farm<>(workers));
    // Execute pipeline
    pipeline.run_and_wait_end();
    return 0;
}
```

Plug and Play with CPA

- We've actually been working with "skeletons" for a long time
- The plug and play set of processes capture some of the ideas - but not quite in the same way
- Some of the more interesting processes we have are:
  - Paraplex (gather)
  - Deparaplex (scatter)
  - Delta
  - Basically any communication pattern
- So we already think in this way. We just need to extend our thinking a little.

Some GLSL

```
// Incoming / outgoing values
layout (location = 0) in vec3 position;
layout (location = 0) out float shade;
// Setting the value
shade = 5.0;
// Emitting vertices and primitives
for (i = 0; i < 3; ++i)
{
    // .. do some calculation
    EmitVertex();
}
EndPrimitive();
```

Task Interface in C++11 CSP

```
void my_task(chan_in<input_type> input, chan_out<output_type>
    out)
{
     while (true)
     {
          // Read input
          auto x = input();
          // ...
          // Write output
          output(y);
     }
}
```

- Unlike a pipeline task we can match arbitrary input to arbitrary output

Creating a Pipeline in C++11 CSP

```
// Plug processes together directly
task_1.out(task_2.in());
// Define a pipeline (pipeline is also a task)
pipeline<input_type, ouput_type> pipe1
{
    task_1,
    task_2,
    task_3
};
// Could also add processes together
task<input_type, output_type> pipe2 = task_1 + task_2 + task_3;
```

Programmers not Plumbers

- These potential examples adopt a different style to standard CPA
- Notice that we don't have to create channels to connect tasks together
  - Although the task method uses channels
- This "pluggable" approach to process composition is something I tried away back in 2006 with .NET CSP
  - I don't think it was well received however

Outline

Descriptive Languages

- Descriptive languages have been put forward to describe skeletal programs
- Limited set of base skeletons used to describe further skeletons
- Aim is to describe the structure of a parallel application using this small set of components
- The description can then be "reasoned" about to enable simplification
  - In other words examine the high level description and determine if a different combination of skeletons would provide the same "behaviour" which would be faster (less communication, reduction, etc.)

- Describes a collection of general purpose blocks
  - **Wrappers** describe how the function is to be run (e.g. sequentially or in parallel)
  - **Combinators** describes communication between blocks
    - **1-to-N** or a deparaplex
    - **N-to-1** or a paraplex
    - **policy** for example unicast, gather, scatter, etc.
  - **Functionals** run parallel computations (e.g. spread, reduce, pipeline)

Example - Task Farm

$$TaskFarm(f) = \triangleleft_{Unicast(Auto)} \bullet [|\Delta|]_n \bullet \triangleright_{Gather}$$

Reading from left to right:

$\triangleleft_{Unicast(Auto)}$ denotes a *1-to-N* communication using a *unicast* policy that is *auto* selected. *auto* means that work is sent to a single available node to process from the available processes.

- is a separator between stages of the pipeline.

$[|\Delta|]_n$ denotes that *n* computations are occurring in parallel. Δ is the computation being undertaken, which is f in the declaration $TaskFarm(f)$.

- is a separator between stages of the pipeline.

$\triangleright_{Gather}$ denotes a *N-to-1* communication using a *gather* policy.

- Uses a functional approach to composition
- For example map

$$map_L(f, [x_1, x_2, \ldots, x_n]) = [f(x_1), f(x_2), \ldots, f(x_3)]$$

- And reduce

$$reduce_L(\oplus, [x_1, x_2, \ldots, x_n]) = x_1 \oplus x_2 \oplus \cdots \oplus x_n$$

- We can therefore describe a Monte Carlo π computation as:

$$pi(points) = result$$
$$\textbf{where}$$
$$f(x, y) = sqr(x) + sqr(y) <= 1$$
$$result = \textbf{reduce}_L(+, \textbf{map}_L(f, points))/n$$

Thinking about CSP as a Description Language

- OK, I've not thought about this too hard
  - I'll leave this to the CSP people
- However I see the same sort of terms used in the descriptive languages
  - Description
  - Reasoning
  - Communication
  - etc.
- Creating a set of CSP "blocks" that could be used to describe skeleton systems could be interesting

Outline

What Doesn't Work for Parallelism

- There has been discussion around what doesn't work for exploiting parallelism in the wide world (don't blame me, blame the literature)
  - automatic parallelization.
  - compiler support is limited to low level optimizations.
  - explicit technologies such as OpenMP and MPI require too much effort.
- The creation of new languages is also not considered a viable route (again don't shoot the messenger)
- So how do we use what we have?

CCSP as a Runtime

- To quote Peter - *"we have the fastest multicore scheduler"*
- So why isn't it used elsewhere?
- I would argue we need to use the runtime as a target platform for existing ideas

OpenMP Parallel For

```
#pragma parallel for num_threads(n)
for (int i = 0; i < m; ++i)
{
    //... do some work
}
```

- Pre-processor generates necessary code
- OpenMP is restrictive on n above - usually 64 max
- A CCSP runtime could overcome this

Outline

Skeletons Aimed at MPI

- Just one slide!
- Most skeleton frameworks use MPI under the hood
  - Help exploit parallelism using MPI
- This is something any CPA skeleton framework would have to look into supporting
  - Handily I'm working on that just now
- As we consider communication more it shouldn't be that difficult.

Outline

Summary

- This work is really about pointing to some potential future directions for CPA
- I have put forward four proposals:

  To the community at large The description and implementation of parallel design patterns and skeletons with CPA techniques

  To the CSP people The use of CSP as a description language for these skeletons

  To the CCSP developers The use of CCSP as a runtime to support parallel execution (such as OpenMP)

  To the distributed runtime developers The use of these ideas in distributed computing to better target cluster computing
- We also need to disseminate these ideas to the wider parallel community if we want them to use these techniques

Questions?