Communicating Connected Components: Extending Plug and Play to Support Skeletons

Kevin Chalmers, Jon Kerridge, Jan Bækgaard Pedersen

School of Computing
Edinburgh Napier University
Edinburgh

k.chalmers@napier.ac.uk
j.kerridge@napier.ac.uk

Department of Computer Science
University of Nevada
Las Vegas

matt.pedersen@unlv.edu

Outline

Outline

Last year...

- I proposed that we should be investigating algorithmic skeletons within our techniques.
- Algorithmic skeletons are a technique for non-parallel programmers (domain experts) to exploit parallelism. An example skeleton is a pipeline which provides a template into which functions can be placed by the programmer.
- A number of such skeleton libraries exist – *eSkel* [Cole, 2004], *Muesli* [Ciechanowicz and Kuchen, 2010], *Skandium* [Leyton and Piquer, 2010], and *SkeTo* [Matsuzaki et al., 2006].

Wrappers describe how a function is to run (e.g. *sequential*, *parallel*).

Combinators describe communication between blocks – *N-to-1*, *1-to-N* and *feedback*. *N-to-1* and *1-to-N* include a communication policy to determine, such as *unicast*, *gather*, etc. Feedback describes a feedback loop with a given condition.

Functionals run parallel computations. Included are *parallel*, *Multiple Instruction, Single Data*, *pipeline*, *spread*, and *reduce*.

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

Reading from left to right:

$\triangleleft_{Unicast(Auto)}$ a *1-to-N* communication using an *auto* selected *unicast* policy.

$\bullet$ separates pipeline stages.

$[|\Delta|]_n$ denotes *n* Δ computations in parallel. Δ is F in $TaskFarm(F)$.

$\bullet$ separates pipeline stages.

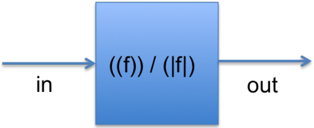
$\triangleright_{Gather}$ a *N-to-1* communication using a *gather* policy.

Outline

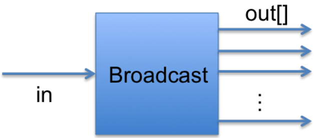
Blocks

- Wrapper
- Combinators 1-to-N
  - Broadcast
  - Scatter
  - Unicast Round Robin
  - Unicast Auto
- Combinators N-to-1
  - Gather
  - Gatherall
- Feedback
- Functionals
  - Parallel
  - Pipeline
  - Spread
  - Reduction

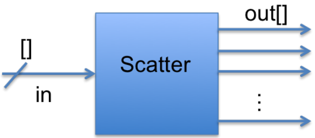
procedure WRAPPER(F, in<X>, out<Y>)
    while true **do**
        in ? value
        out ! F(value)
    end while
end procedure



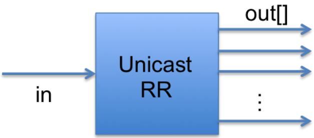
```
procedure BROADCAST(in<X>, out<X>[n])
    while true do
        in ? value
        par for i in 0..n-1 do
            out[i] ! value
    end while
end procedure
```



```
procedure SCATTER(in<X[n]>, out<X>[n])
    while true do
        in ? value
        par for i in 0..n-1 do
            out[i] ! value[i]
    end while
end procedure
```

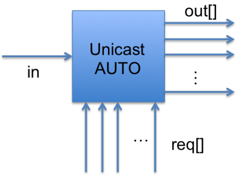


```
procedure UNICAST_RR(in<X>, out<X>[n])
    while true do
        for i in 0..n-1 do
            in ? value
            out[i] ! value
        end for
    end while
end procedure
```

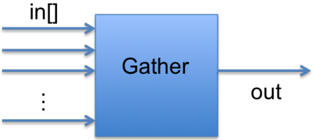


Unicast (Auto)

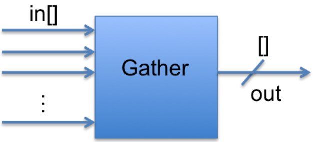
```
procedure UNICAST_AUTO(in<X>, req<ℕ>, out<X>[n])
    while true do
        in ? value
        req ? idx
        out[idx] ! value
    end while
end procedure

procedure UNICAST_AUTO_GUARDED(in<X>, out<X>[n])
    while true do
        in ? value
        select chan from out
            chan ! value
    end while
end procedure
```



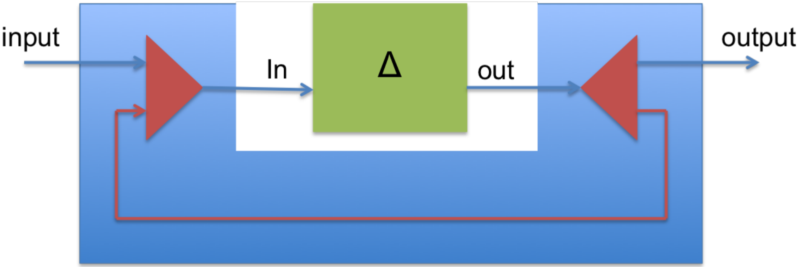
```
procedure GATHER(in<X>[n], out<X>)
    while true do
        for i in 0..n-1 do
            in[i] ? value
            out ! value
        end for
    end while
end procedure
```



```
procedure GATHERALL(in<X>[n], out<X[n]>)
    X value[n]
    while true do
        par for i in 0..n-1 do
            in[i] ? value[i]
        out ! value
    end while
end procedure
```



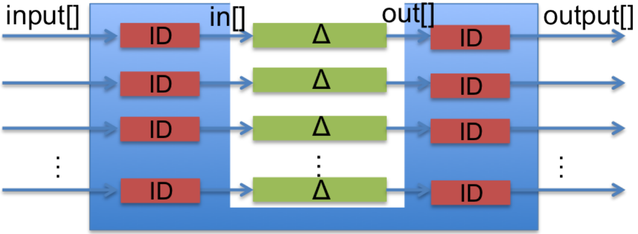
Feedback



Feedback

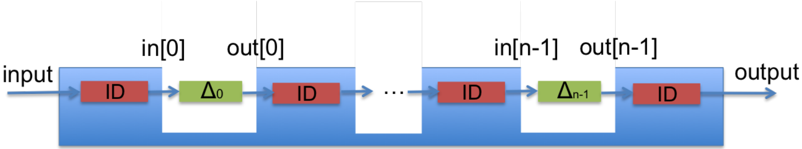
```
procedure MERGE(in<X>, to_block<X>, from_block<X>, out<X>, cond)
    while true do
        in ? value
        to_block ! value
        from_block ? value
        while cond(value) do
            to_block ! value
            from_block ? value
        end while
        out ! value
    end while
end procedure

procedure FEEDBACK(BLOCK, cond, in<X>, out<X>)
    to_block<X>
    from_block<X>
    par
        BLOCK(to_block, from_block)
        MERGE(in, to_block, from_block, out, cond)
end procedure
```

procedure PAR(BLOCK, in<X>[n], out<Y>[n])
  par for i in 0..n-1 **do**
    BLOCK(in[i], out[i])
end procedure

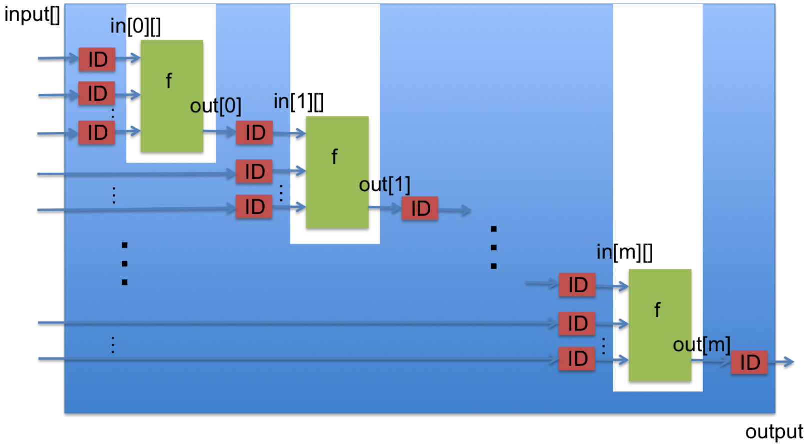


- May also work with a range of processes (i.e., BLOCK[n] - MIMD)

procedure PIPELINE(block[n], in<X>, out<Y>)
    internal[n - 1]
    par
        block[0](in, internal[0])
        par for i in 1..n-2 **do**
            block[i](internal[i - 1], internal[i])
        block[n-1](internal[n - 2], out)
end procedure



```
procedure SPREADER(F, param, k, out<X>[n])
    value ← F(param)                                ▷ value has arity k
    if k = n then
        par for i in 0..n-1 do
            out[i] ! value[i]
    else
        par for i in 0..n-1 do
            SPREADER(F, value[i], k, out[n/k * i]...out[n/k * (i + 1)])
    end if
end procedure
procedure SPREAD(F, k, in<X>, out<X<[n])
    while true do
        in ? value
        SPREADER(F, value, k, out)
    end while
end procedure
```


input[]
in[0][]
ID
ID
ID
f
out[0]
in[1][]
ID
ID
ID
f
out[1]
ID
in[m][]
ID
ID
ID
f
out[m]
ID
output

```
procedure REDUCER(f, k, params[n])
    if k = n then
        return f(params)
    end if
    X values[n/k]
    par for i in 0..(n/k) - 1 do
        values[i] ← reducer(f, k, params[n/k * i]..params[n/k * (i + 1)])
    return f(values)
end procedure
procedure REDUCE(f, k, in<X>[n], out<X>)
    X values[n]
    par for i in 0..n-1 do
        in[i] ? values[i]
    out ! reducer(f, k, values)
end procedure
```

Outline

Concordance

- Given a text, extract the location of equal word strings for strings of words of lengths 1..N in terms of the starting location of the word string in the text, provided the word string is repeated a minimum number of times.
- For example, search the Bible for seven word strings will pull out "And God saw that it was good" in multiple locations.

- Two solutions - parallel grouping of pipelines, or pipelining of parallel groups
- Group of Pipelines (GoP)

$$GoP = ((emit)) \bullet \triangleleft_{Unicast(Auto)} \bullet [|2 \bullet 3 \bullet 4 \bullet 5|]_n$$

- Pipeline of Groups

$$PoG = ((emit)) \bullet \triangleleft_{Unicast(Auto)} \bullet [|2|]_n \bullet [|3|]_n \bullet [|4|]_n \bullet [|5|]_n$$

Concordance Results

Groups	Time (ms)	Speedup
1	24281.5	1.181
2	23765.5	1.207
3	22211	1.292
4	21695.5	1.322

Groups	Time (ms)	Speedup
1	24430	1.174
2	22984	1.248
3	21883	1.311
4	21734.5	1.320

Outline

Conclusions

- We have demonstrated that taking a process orientated view to skeleton block definition and composition provides a simple understanding of input and output typing, and the potential parallel behaviour within a block.
- We have also provided results of a concordance application using these blocks within a message passing Groovy library.
  - Jon did a presentation (here) to the Groovy community.
  - Jon's writing another Groovy book on using this approach.
- Future work
  - We aim to take these definitions and implement them in other message passing languages and libraries.
  - We aim to utilise C++ variadic templates to provide simple skeleton composition to the application programmer.

References

Ciechanowicz, P. and Kuchen, H. (2010).
Enhancing Muesli's Data Parallel Skeletons for Multi-core Computer Architectures.
In *2010 12th IEEE International Conference on High Performance Computing and Communications (HPCC)*, pages 108–113.

Cole, M. (2004).
Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming.
Parallel Computing, 30(3):389–406.

Leyton, M. and Piquer, J. M. (2010).
Skandium: Multi-core Programming with Algorithmic Skeletons.
pages 289–296. IEEE.

Matsuzaki, K., Iwasaki, H., Emoto, K., and Hu, Z. (2006).
A Library of Constructive Skeletons for Sequential Style of Parallel Programming.
In *Proceedings of the 1st International Conference on Scalable Information Systems*, InfoScale '06, New York, NY, USA. ACM.