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
1. Background

2. Skeletal Components

3. Some Experimental Results

4. Conclusions
1. Background

2. Skeletal Components

3. Some Experimental Results

4. Conclusions
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\( (F) = \langle Unicast(Auto) \rangle \bullet [|\Delta|]_n \bullet \triangleright Gather \)

Reading from left to right:

\( \langle Unicast(Auto) \rangle \) a 1-to-N communication using an auto selected unicast policy.

- separates pipeline stages.

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

- separates pipeline stages.

\( \triangleright Gather \) a N-to-1 communication using a gather policy.
1 Background

2 Skeletal Components

3 Some Experimental Results

4 Conclusions
- 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 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 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
procedure UNICAST_AUTO\(\text{in}\langle X\rangle, \text{req}\langle \mathbb{N}\rangle, \text{out}\langle X\rangle[n]\))
    while true do
        in ? value
        req ? idx
        out[idx] ! value
    end while
end procedure

procedure UNICAST_AUTO_GUARDED\(\text{in}\langle X\rangle, \text{out}\langle X\rangle[n]\))
    while true do
        in ? value
        select chan from out
        chan ! value
    end while
end procedure
procedure \textsc{gather}(\text{in}<X>[n], \text{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 while
end procedure
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)  \(\triangleright\) 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
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
1 Background

2 Skeletal Components

3 Some Experimental Results

4 Conclusions
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.
Solution - Groovy Parallel Library

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

\[ GoP = ((\text{emit})) \odot \text{Unicast(Auto)} \odot [[2 \odot 3 \odot 4 \odot 5]]_n \]

- Pipeline of Groups

\[ PoG = ((\text{emit})) \odot \text{Unicast(Auto)} \odot [|2|_n \odot [|3|_n \odot [|4|_n \odot [|5|_n} \]
<table>
<thead>
<tr>
<th>Groups</th>
<th>Time (ms)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>24281.5</td>
<td>1.181</td>
</tr>
<tr>
<td>2</td>
<td>23765.5</td>
<td>1.207</td>
</tr>
<tr>
<td>3</td>
<td>22211</td>
<td>1.292</td>
</tr>
<tr>
<td>4</td>
<td>21695.5</td>
<td>1.322</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Groups</th>
<th>Time (ms)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>24430</td>
<td>1.174</td>
</tr>
<tr>
<td>2</td>
<td>22984</td>
<td>1.248</td>
</tr>
<tr>
<td>3</td>
<td>21883</td>
<td>1.311</td>
</tr>
<tr>
<td>4</td>
<td>21734.5</td>
<td>1.320</td>
</tr>
</tbody>
</table>
1 Background

2 Skeletal Components

3 Some Experimental Results

4 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

Enhancing Muesli’s Data Parallel Skeletons for Multi-core Computer Architectures.

Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming.

Skandium: Multi-core Programming with Algorithmic Skeletons.
pages 289–296. IEEE.

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.