Parallelizing Maximal Clique Enumeration in Haskell

Andres Löh
(joint work with Toni Cebrián)

Well-Typed LLP

7 February 2012
Background:
The Parallel GHC Project
The Parallel GHC Project

- Currently the largest project we have at Well-Typed LLP.
- Funded by Microsoft Research in Cambridge (GHC HQ).
- Runs for two years (until June 2012).
The Parallel GHC Project

Goals

- polish GHC’s support for parallel programming,
- demonstrate the parallel programming in Haskell works and scales,
- develop and improve tools that support parallel programming in Haskell,
- develop tutorials and information material.
Los Alamos National Labs (USA)
Monte Carlo algorithms for particle and radiation simulation
The Parallel GHC Project
Participating organizations

- **Los Alamos National Labs** (USA)
  Monte Carlo algorithms for particle and radiation simulation

- **Dragonfly** (New Zealand)
  Implementation of a fast Bayesian model fitter
The Parallel GHC Project
Participating organizations

- **Los Alamos National Labs** (USA)
  Monte Carlo algorithms for particle and radiation simulation

- **Dragonfly** (New Zealand)
  Implementation of a fast Bayesian model fitter

- **Internet Initiative Japan** (Japan)
  High-performance network servers
The Parallel GHC Project

Participating organizations

- **Los Alamos National Labs** (USA)
  Monte Carlo algorithms for particle and radiation simulation

- **Dragonfly** (New Zealand)
  Implementation of a fast Bayesian model fitter

- **Internet Initiative Japan** (Japan)
  High-performance network servers

- **Telefonica R+D** (Spain)
  Parallel/distributed graph algorithms
The Parallel GHC Project

Participating organizations

- **Los Alamos National Labs** (USA)
  Monte Carlo algorithms for particle and radiation simulation

- **Dragonfly** (New Zealand)
  Implementation of a fast Bayesian model fitter

- **Internet Initiative Japan** (Japan)
  High-performance network servers

- **Telefonica R+D** (Spain)
  Parallel/distributed graph algorithms

Each partner organization has a Haskell project involving parallelism they want to implement.
The Parallel GHC Project
Workflow

- Organizations discuss their project plans with us.
- We jointly develop implementation goals and the design of the programs.
- The organizations develop the programs, with our assistance.
- We identify potential problems and stumbling blocks.
- We spark off separate mini-projects in order to fix such problems.
- We communicate ideas for further improvements to the GHC developers.
- We collect results and experiences and extract it into regular project digests, and later into new tutorial material.
Mini-projects so far

- A web portal for parallel programming in Haskell.
- A monthly newsletter on parallel programming in Haskell.
- Fixing hidden limits in the GHC IO manager.
- A Haskell binding for MPI.
- Better visualizations in ThreadScope.
- Parallel PRNGs in Haskell.
- ...
A case study: trying to (re)implement parallel Maximal Clique Enumeration in Haskell.
Maximal Clique Enumeration
Maximal Clique Enumeration

Definitions

Clique

A **clique** in an undirected graph is a complete subgraph, i.e., a subgraph where every two vertices are connected.
Maximal Clique Enumeration

Definitions

**Clique**

A *clique* in an undirected graph is a complete subgraph, i.e., a subgraph where every two vertices are connected.

**Maximal Clique**

A clique in a graph is called *maximal* if there is no larger clique containing it.
Maximal Clique Enumeration

Definitions

**Clique**

A *clique* in an undirected graph is a complete subgraph, i.e., a subgraph where every two vertices are connected.

**Maximal Clique**

A clique in a graph is called *maximal* if there is no larger clique containing it.

**Maximal Clique Enumeration (MCE)**

Given an undirected graph, determine all maximal cliques in that graph.
Problem is exponential in the worst case as there are graphs with exponentially many maximal cliques (in the size of vertices).

There are several MCE algorithms that perform well in practice.

We’re going to look at the Bron-Kerbosch (BK) algorithm (1973) – good combination of performance and simplicity.
Bk state

Bk maintains a state of three sets of vertices:

- **compsub**: active clique
- **cand**: candidates for extending the active clique
- **excl**: possible extensions of the active clique that would lead to duplication (originally called not)
BK state

BK maintains a state of three sets of vertices:

- **compsub**: active clique
- **cand**: candidates for extending the active clique
- **excl**: possible extensions of the active clique that would lead to duplication (originally called **not**)

Initial state (given graph $G = (V, E)$):

- **compsub** := $\emptyset$
- **cand** := $V$
- **excl** := $\emptyset$
BK in imperative pseudocode

```
bk (compsub, cand, excl) :
    if null cand && null excl then report compsub
    foreach v in cand :
        bk (compsub ∪ {v}, cand ∩ N(v), excl ∩ N(v))
        cand := cand \ {v}
        excl := excl ∪ {v}
```

where \( N(v) \) are the neighbours of vertex \( v \).
**BK in Haskell**

```haskell
type Clique = [Vertex]
bk :: Clique → [Vertex] → [Vertex] → [Clique]
bk compsub cand excl =
  if null cand && null excl then [compsub]
  else loop cand excl

where
  loop :: [Vertex] → [Vertex] → [Clique]
  loop [] _ = []
  loop (v : cand’) excl =
    bk (v : compsub) (cand’ ‘res’ v) (excl ‘res’ v) ++
    loop cand’ (v : excl)

where vs ‘res’ v removes the vertices that are not connected
to v from vs .
```
We should abstract over an input graph.

```haskell
type Vertex = Int

class Graph g where
    size :: g → Int
    vertices :: g → [Vertex]
    connected :: g → Vertex → Vertex → Bool
```
Bron-Kerbosch

bronKerbosch :: Graph g ⇒ g → [Clique]
bronKerbosch g = bk [] (vertices g) [] -- initial state

where
  bk = . . . -- as before
  res :: [Vertex] → Vertex → [Vertex]
  res vs v = filter (connected g v) vs
Example

\[
\text{gr } = \text{edgesToGraph}
\]

\[
[(1, 2), (1, 3), (2, 3), (2, 4), (2, 5), (3, 4), (4, 5), (4, 6), (5, 6)]
\]

\[
\text{test } = \text{bronKerbosch gr } \equiv [[3, 2, 1], [4, 3, 2], [5, 4, 2], [6, 5, 4]]
\]
Example

\[\text{gr} = \text{edgesToGraph} \\]
\[
[(1, 2), (1, 3), (2, 3), (2, 4), (2, 5), (3, 4), (4, 5), (4, 6), (5, 6)]
\]

\[\text{test} = \text{bronKerbosch} \text{ gr} \equiv [[3, 2, 1], [4, 3, 2], [5, 4, 2], [6, 5, 4]]\]
Example

gr = edgesToGraph
    [(1, 2), (1, 3), (2, 3), (2, 4), (2, 5), (3, 4), (4, 5), (4, 6), (5, 6)]

test = bronKerbosch gr == [[3, 2, 1], [4, 3, 2], [5, 4, 2], [6, 5, 4]]
Example

\[ \text{gr} = \text{edgesToGraph} \]
\[ [(1, 2), (1, 3), (2, 3), (2, 4), (2, 5), (3, 4), (4, 5), (4, 6), (5, 6)] \]

\[ \text{test} = \text{bronKerbosch gr} = [(3, 2, 1), (4, 3, 2), (5, 4, 2), (6, 5, 4)] \]
Example

gr = edgesToGraph
    [(1, 2), (1, 3), (2, 3), (2, 4), (2, 5), (3, 4), (4, 5), (4, 6), (5, 6)]

test = bronKerbosch gr = [[3, 2, 1], [4, 3, 2], [5, 4, 2], [6, 5, 4]]

Found a maximal clique; backtrack.
Example

gr = edgesToGraph
   [(1, 2), (1, 3), (2, 3), (2, 4), (2, 5), (3, 4), (4, 5), (4, 6), (5, 6)]

test = bronKerbosch gr = [[3, 2, 1], [4, 3, 2], [5, 4, 2], [6, 5, 4]]

Excluded vertices prevent reporting the same clique again.
**Example**

\[
\text{gr} = \text{edgesToGraph} \\
\quad [(1, 2), (1, 3), (2, 3), (2, 4), (2, 5), (3, 4), (4, 5), (4, 6), (5, 6)] \\
\text{test} = \text{bronKerbosch} \quad \text{gr} = [(3, 2, 1), (4, 3, 2), (5, 4, 2), (6, 5, 4)]
\]

Excluded vertices prevent reporting the same clique again.
Example

\[
\begin{align*}
gr = \text{edgesToGraph} \\
\quad [(1, 2), (1, 3), (2, 3), (2, 4), (2, 5), (3, 4), (4, 5), (4, 6), (5, 6)] \\
test = \text{bronKerbosch gr} \equiv [[3, 2, 1], [4, 3, 2], [5, 4, 2], [6, 5, 4]]
\end{align*}
\]

Excluded vertices prevent reporting the same clique again.
Example

gr = edgesToGraph
    [(1, 2), (1, 3), (2, 3), (2, 4), (2, 5), (3, 4), (4, 5), (4, 6), (5, 6)]

test = bronKerbosch gr =: [[3, 2, 1], [4, 3, 2], [5, 4, 2], [6, 5, 4]]

Another maximal clique found.
Improving the (sequential) algorithm

Some minor modifications help making BK more efficient:
- pick a suitable graph representation (connected should be efficient),
- not traverse all the elements of cand; instead, pick the most connected candidate \( p \) first, and subsequently only consider candidates that are not connected to \( p \).
Strategies for Deterministic Parallelism
Parallelism using annotations

Overview

- In Haskell, we can annotate computations for parallel execution.
- Annotations create **sparks**.
- When cores are idle, the Haskell RTS will steal sparks and run them.
- All low-level details are managed by the RTS.
- Due to Haskell’s purity, using annotations does not affect the result of a program (speculative, deterministic parallelism).
Parallelism using annotations

Interface

```haskell
data Eval a -- (abstract), annotated terms
instance Monad Eval -- we can combine such terms

type Strategy a = a \to Eval a -- a strategy annotates a term

dot :: Strategy a \to Strategy a \to Strategy a -- composition of strategies

using :: a \to Strategy a \to a -- applying a strategy
```
Basic strategies

--- evaluation:

\[ r_0 :: \quad \text{Strategy a} \quad -- \text{none} \]

\[ r_{\text{seq}} :: \quad \text{Strategy a} \quad -- \text{WHNF} \]

\[ r_{\text{deepseq}} :: \text{NFData} ~ a \Rightarrow \text{Strategy a} \quad -- \text{NF} \]

\[ r_{\text{par}} :: \quad \text{Strategy a} \quad -- \text{WHNF in parallel} \]

Names start with “r”: think “reduce”.

\[ r_0 = \text{return} \]

The first three strategies determine how much of a term is evaluated. The \textsf{rpar} strategy introduces a spark.
Strategies are datatype-oriented

Given a datatype, it’s easy to define strategy combinators.

For example:

\[
\text{evalList, parList} :: \text{Strategy } a \to \text{Strategy } [a] \\
\text{evalList } s \ [\ ] = \text{return } [\ ] \\
\text{evalList } s \ (x : xs) = \text{do} \\
\quad r \leftarrow s \ x \\
\quad rs \leftarrow \text{evalList } s \ xs \\
\text{return } (r : rs) \\
\text{parList } s = \text{evalList } (rpar \ ‘dot’ \ s)
\]
Strategies are datatype-oriented

Given a datatype, it’s easy to define strategy combinators.

For example:

```haskell
 evalList, parList :: Strategy a -> Strategy [a]
 evalList s [] = return []
 evalList s (x : xs) = do
  r ← s x
  rs ← evalList s xs
  return (r : rs)

 parList s = evalList (rpar ‘dot‘ s)
```

Similarly for all members of Traversable.
Parallelizing BK

- BK is a recursive algorithm.
- Parallelization via the data we operate on does not seem suitable.
Parallelizing BK

BK is a recursive algorithm.
Parallelization via the data we operate on does not seem suitable.
Instead, we’d like to parallelize on the call tree.
Parallelizing BK

- BK is a recursive algorithm.
- Parallelization via the data we operate on does not seem suitable.
- Instead, we’d like to parallelize on the call tree.
- We can just turn the call tree into a data structure.
BK revisited

\[
\text{bronKerbosch' :: Graph } g \Rightarrow g \rightarrow [\text{Clique}]
\]
\[
\text{bronKerbosch'} \ g = \text{bk } [] \ (\text{vertices } g) \ [ ] \quad \text{-- initial state}
\]

where

\[
\text{bk compsub cand excl } = \ldots
\]

\[
\text{if null cand } \&\& \text{ null excl then } [\text{compsub}]
\]

else loop cand excl

where

\[
\text{loop } [ ] \quad _\quad = [ ]
\]
\[
\text{loop } (v : \text{cand'}) \ \text{excl } =
\]
\[
\text{bk } (v : \text{compsub}) \ (\text{cand'} \ 'res' \ v) \ (\text{excl} \ 'res' \ v) \ +
\]
\[
\text{loop } \text{cand'} \ (v : \text{excl})
\]
\[
\text{res vs } v = \text{filter } (\text{connected } g \ v) \ \text{vs}
\]
**BK revisited**

<table>
<thead>
<tr>
<th>type</th>
<th>BKState = (Clique, [Vertex], [Vertex])</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>BKTree = Fork BKState [BKTree]</td>
</tr>
</tbody>
</table>

\[
\text{bronKerbosch}': \text{Graph } g \Rightarrow g \rightarrow [\text{Clique}]
\]

\[
\text{bronKerbosch}': g = \text{bk} [] (\text{vertices } g) [] \quad \text{-- initial state}
\]

**where**

\[
\text{bk compsub cand excl} = \ldots
\]

\[
\text{if null cand } && \text{null excl} \text{ then } [\text{compsub}]
\]

\[
\text{else loop cand excl}
\]

**where**

\[
\text{loop [] } = []
\]

\[
\text{loop } (v : \text{cand'}) \text{ excl} =
\]

\[
\text{bk } (v : \text{compsub}) (\text{cand'} 'res' v) (\text{excl} 'res' v) +
\]

\[
\text{loop cand'} (v : \text{excl})
\]

\[
\text{res vs v} = \text{filter} (\text{connected } g v) \text{ vs}
\]
BK revisited

type BKState = (Clique, [Vertex], [Vertex])
data BKTree = Fork BKState [BKTree] | Report Clique

bronKerbosch' :: Graph g ⇒ g → BKTree
bronKerbosch' g = bk [] (vertices g) [] -- initial state
   where
      bk compsub cand excl = ...
      if null cand && null excl then [compsub]
         else loop cand excl

      where
         loop [] _ = []
         loop (v : cand') excl =
            bk (v : compsub) (cand' 'res' v) (excl 'res' v) ++
            loop cand' (v : excl)

         res vs v = filter (connected g v) vs
BK revisited

\textbf{type} \ BKState \ = \ (\text{Clique}, [\text{Vertex}], [\text{Vertex}])

\textbf{data} \ BKTree \ = \ Fork \ BKState \ [BKTree] \ | \ Report \ Clique

bronKerbosch' :: Graph \ g \ \Rightarrow \ g \ \rightarrow \ BKTree
bronKerbosch' \ g \ = \ bk \ [] \ (\text{vertices} \ g) \ [] \ -- \ initial \ state

\textbf{where}

\begin{align*}
\text{bk} \ \text{compsub} \ \text{cand} \ \text{excl} \ &= \ Fork \ (\text{compsub}, \ \text{cand}, \ \text{excl}) \\
\text{if} \ \text{null} \ \text{cand} \ \&\& \ \text{null} \ \text{excl} \ \text{then} \ [\text{compsub}] \\
\text{else} \ \text{loop} \ \text{cand} \ \text{excl}
\end{align*}

\textbf{where}

\begin{align*}
\text{loop} \ [\ ] \ &= \ [\ ] \\
\text{loop} \ (v : \ \text{cand'}) \ \text{excl} &= \\
&= \ bk \ (v : \ \text{compsub}) \ (\text{cand'} \ 'res' \ v) \ (\text{excl} \ 'res' \ v) \ \# \\
&\quad \text{loop} \ \text{cand'} \ (v : \ \text{excl})
\end{align*}

\text{res} \ vs \ v \ = \ \text{filter} \ (\text{connected} \ g \ v) \ vs
BK revisited

\textbf{type} BKState = (Clique, [Vertex], [Vertex])
\textbf{data} BKTree = Fork BKState [BKTree] | Report Clique

bronKerbosch' :: Graph g ⇒ g → BKTree
bronKerbosch' g = bk [] (vertices g) [] -- initial state

\textbf{where}

bk compsub cand excl = Fork (compsub, cand, excl) $
\textbf{if} \text{ null cand} \&\& \text{ null excl} \textbf{ then} [\text{Report compsub}]
\textbf{else} \text{ loop cand excl}

\textbf{where}

loop [] = []
loop (v : cand') excl =
bk (v : compsub) (cand' 'res' v) (excl 'res' v) ++
loop cand' (v : excl)
res vs v = filter (connected g v) vs
BK revisited

**type** BKState = (Clique, [Vertex], [Vertex])

**data** BKTree = Fork BKState [BKTree] | Report Clique

bronKerbosch' :: Graph g ⇒ g → BKTree
bronKerbosch' g = bk [] (vertices g) [] -- initial state

where

bk compsub cand excl = Fork (compsub, cand, excl) $
  \text{if null cand && null excl then } [\text{Report compsub}]$

else loop cand excl

where

loop [] _ = []
loop (v : cand') excl =
  bk (v : compsub) (cand' 'res' v) (excl 'res' v) :
  loop cand' (v : excl)

res vs v = filter (connected g v) vs
Extracting the cliques

\[
\begin{align*}
\text{extract} :: \text{BKTree} & \rightarrow [\text{Clique}] \\
\text{extract} \ (\text{Fork } \_ \ \text{xs}) & = \text{concat} \ (\text{map} \ \text{extract} \ \text{xs}) \\
\text{extract} \ (\text{Report } \text{c}) & = [\text{c}]
\end{align*}
\]
Extracting the cliques

\[
\text{extract} :: \text{BKTree} \rightarrow [\text{Clique}]
\]
\[
\text{extract} (\text{Fork } xs) = \text{concat} (\text{map} \text{ extract} \ xs)
\]
\[
\text{extract} (\text{Report c}) = [c]
\]

\[
\text{property } g = \text{extract} (\text{bronKerbosch}' g) ::= \text{bronKerbosch } g
\]
A strategy for call trees

Ideally, this one would do:

```haskell
strategy :: Strategy BKTree
strategy (Fork s xs) = fmap (Fork s) (parList strategy xs)
strategy (Report c) = fmap Report (rdeepseq c)
```
A strategy for call trees

Ideally, this one would do:

```haskell
strategy :: Strategy BKTree
strategy (Fork s xs) = fmap (Fork s) (parList strategy xs)
strategy (Report c) = fmap Report (rdeepseq c)
```

Problems:

- Too many sparks created in too little time (spark pool overflows).
- Too many sparks that are too small to do any good.
- Sequential optimizations interfere with parallelisation.
Options

- Reduce the number of sparks, by chunking the lists.
- Increase granularity, also by chunking the lists.
- Limit the depth of parallelization (but that’s not good due to the imbalanced nature of the call trees).
- Don’t create sparks for leaves.

... 

All of these can be achieved just by changing the strategy. Nothing else in the program has to be touched. Thus:

- getting some form of speedup even for an algorithm that isn’t trivial to parallelize is actually not a lot of work;
- the call tree technique is widely applicable and extensible.
We have found strategies that provide reasonable speedups up to eight cores, but:

- these strategies aren’t dynamic enough;
- some graphs can usually be found that work bad with a given strategy, but better with others;
- the speedup is not linear;
- current tests indicate that things get slower again from eight cores up.
On the other hand:

- Schmidt et al. “A scalable, parallel algorithm for maximal clique enumeration” use a similar technique (which in fact inspired us) in an imperative/distributed setting and report linear speedups up to 2048 cores.
- There’s (relatively speaking) much more effort involved in implementing the technique.
Future work

- More testing and examples.
- Strategies should be more dynamic.
- Provide more information in the call tree.
- More control over RTS needed after all?
- Overhead for collecting cliques in deterministic order?
- Another approach to deterministic parallelism: Par monad, with user-implementable schedulers.
- Also try distributed systems via “Cloud Haskell”.
- Try more graph algorithms.
- ...