Parallelization of Spider Planner

Morten Smedsrud, SINTEF ICT
SPIDER - A Generic VRP Solver

- Designed to be widely applicable
- Based on generic, rich model
- Predictive route planning
- Plan repair, reactive planning
- Dynamic planning with stochastic model
- Framework for VRP research
SPIDER - Generalisations of CVRP

- Heterogeneous fleet
  - Capacities
  - Equipment
  - Arbitrary tour start/end locations
  - Time windows
  - Cost structure
- Linked tours with precedences
- Mixture of order types
- Multiple time windows, soft time windows
- Capacity in multiple dimensions, soft capacity
- Alternative locations, tours and orders
- Arc locations, for arc routing and aggregation of node orders
- Alternative time periods
- Non-Euclidean, asymmetric, dynamic travel times
- Compatibility constraints
- A variety of constraint types and cost components
  - driving time restrictions
  - visual beauty of routing plan, non-overlapping
<table>
<thead>
<tr>
<th>Year</th>
<th>Processor</th>
<th>Clock Frequency</th>
<th>Max Power</th>
<th>Cores(Threads)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1978</td>
<td>8086 / 8088</td>
<td>5-8MHz</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1982</td>
<td>80286</td>
<td>6-25MHz</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1985</td>
<td>80386</td>
<td>12-40MHz</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1989</td>
<td>80486</td>
<td>16-100MHz</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1993</td>
<td>Pentium</td>
<td>60-233MHz</td>
<td>17W</td>
<td>1</td>
</tr>
<tr>
<td>1995</td>
<td>Pentium Pro</td>
<td>150-200MHz</td>
<td>35W</td>
<td>1</td>
</tr>
<tr>
<td>1997</td>
<td>Pentium 2</td>
<td>233-450Mhz</td>
<td>27W</td>
<td>1</td>
</tr>
<tr>
<td>1999</td>
<td>Pentium III</td>
<td>450-1400MHz</td>
<td>32W</td>
<td>1</td>
</tr>
<tr>
<td>2000-2008</td>
<td>Pentium IV</td>
<td>1.3-3.8GHz</td>
<td>115W</td>
<td>1(2)</td>
</tr>
<tr>
<td>2005-</td>
<td>Athlon X2</td>
<td>1-3.2GHz</td>
<td>125W</td>
<td>2</td>
</tr>
<tr>
<td>2006-</td>
<td>Intel Core 2</td>
<td>1.06-3.33GHz</td>
<td>130W</td>
<td>2(4)</td>
</tr>
<tr>
<td>2008-</td>
<td>Intel Core i7</td>
<td>1.6-3.33GHz</td>
<td>130W</td>
<td>4-6(8-12)</td>
</tr>
</tbody>
</table>
The shared memory model on PCs

- All data available to all CPU cores in the same address space
- Bandwith is shared and CPU cores must maintain cache coherency
- Potential race conditions
- Software tools: events, wait functions and critical sections.
- Deadlocks
- Hardware tool: Atomic operations
SPIDER characteristics

- Original design back in 1996-1998
- Written in C++, heavy use of STL and smart pointers for memory management
- Based on iterated local search
- 5 different initial constructors
- 15 different types of operators / neighbourhoods
- 14 objective / 8 constraint types
- More than 300k lines of code
- Profiling showed majority of the runtime spend on evaluating neighbours
Move / neighbourhood architecture

- Neighbourhoods present moves sequentially
- Neighbourhood can be set up to filter away many obviously infeasible moves (default on)
- Moves present the changes they represent to constraints and objectives by applying the changes (and undoing them)
- Moves also report what parts of the solution they change
Parallel implementation

- 1 Master / multiple slave threads
- To avoid race conditions each thread maintains and works on its own copy of the solution
- Each thread takes turns on getting moves to evaluate from the sequential neighbourhood (through critical section)
- Each move gets a serial number from its neighbourhood, to get deterministic behavior
- Make code thread safe by the use of critical sections
- Using atomic operations on smart pointers to remove the overhead of critical sections
- Some previously static allocated arrays allocated dynamically to make functions reentrant
Relocate results

![Graph showing Relocate results with lines for 1T Relocate, 2T Relocate, and 4T Relocate.]
2 opt results
Or exchange results
Insert results

![Graph showing Insert results with lines for 1T insert, 2T insert, and 4T insert. The x-axis represents time in steps, and the y-axis represents the value. The graph shows the comparison of different insert results over time.]
Possible explanation

- Sequential neighbourhood generation taking too much time
- Too many critical sections somewhere
- Slowed by reallocation of previous statically allocated arrays
Possible improvements

- Redo the neighbourhood generation, generate smaller independent sub neighbourhoods that can be evaluated and filtered in parallel
- Reimplementing thread safe version of some code optimizations that was removed because it was not thread safe
- Change the way moves present changes, without applying them. Would reduce the need to maintain copy of solution data for each thread.
- If neighbourhood size can be estimated in advance, use sequential evaluation on small neighbourhoods
- Re implement cache arrays to be statically allocated, 1 per thread
Preliminary Conclusions?

- Non deterministic execution can make debugging shared memory applications difficult.
- Converting old sequential code to thread safe code turned out to be a lot more costly than expected.
- Problem cases probably need to be of a certain size before parallel evaluation makes sense.
- Care must be taken to avoid unnecessary thread synchronisation as it can be costly.
- This implementation needs more tuning and testing. Currently does not appear to scale past 2 threads.