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

PC Microprocessor development

Year	Processor	Clock Frequency	Max Power	Cores(Threads)
1978	8086 / 8088	5-8MHz		1
1982	80286	6-25MHz		1
1985	80386	12-40MHz		1
1989	80486	16-100MHz		1
1993	Pentium	60-233MHz	17W	1
1995	Pentium Pro	150-200MHz	35W	1
1997	Pentium 2	233-450Mhz	27W	1
1999	Pentium III	450-1400MHz	32W	1
2000-2008	Pentium IV	1.3-3.8GHz	115W	1(2)
2005-	Athlon X2	1-3.2GHz	125W	2
2006-	Intel Core 2	1.06-3.33GHz	130W	2(4)
2008-	Intel Core i7	1.6-3.33GHz	130W	4-6(8-12)

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
- Potentional 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

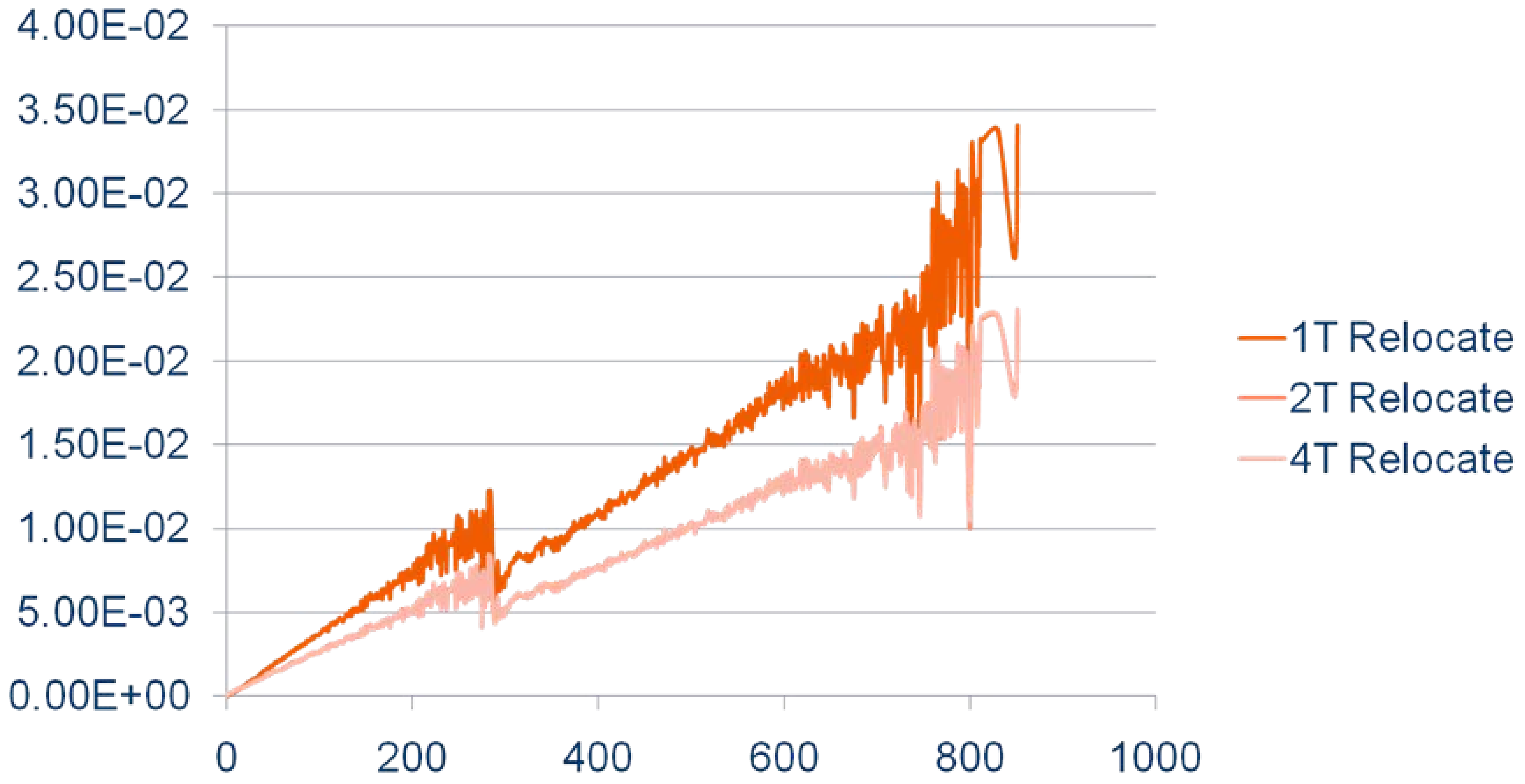
Move / neighbourhood architecture

- 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

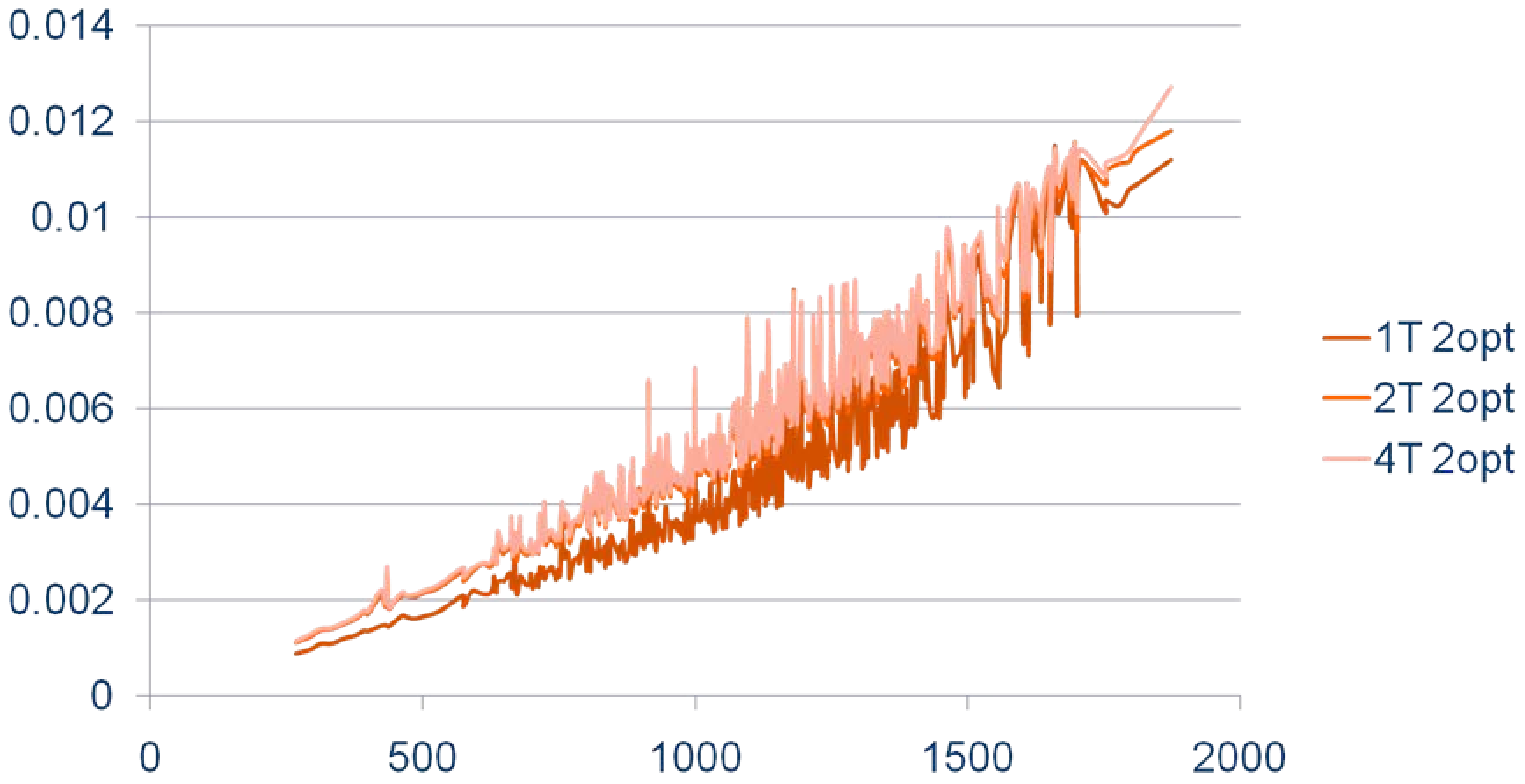
Parallel implementation

- 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

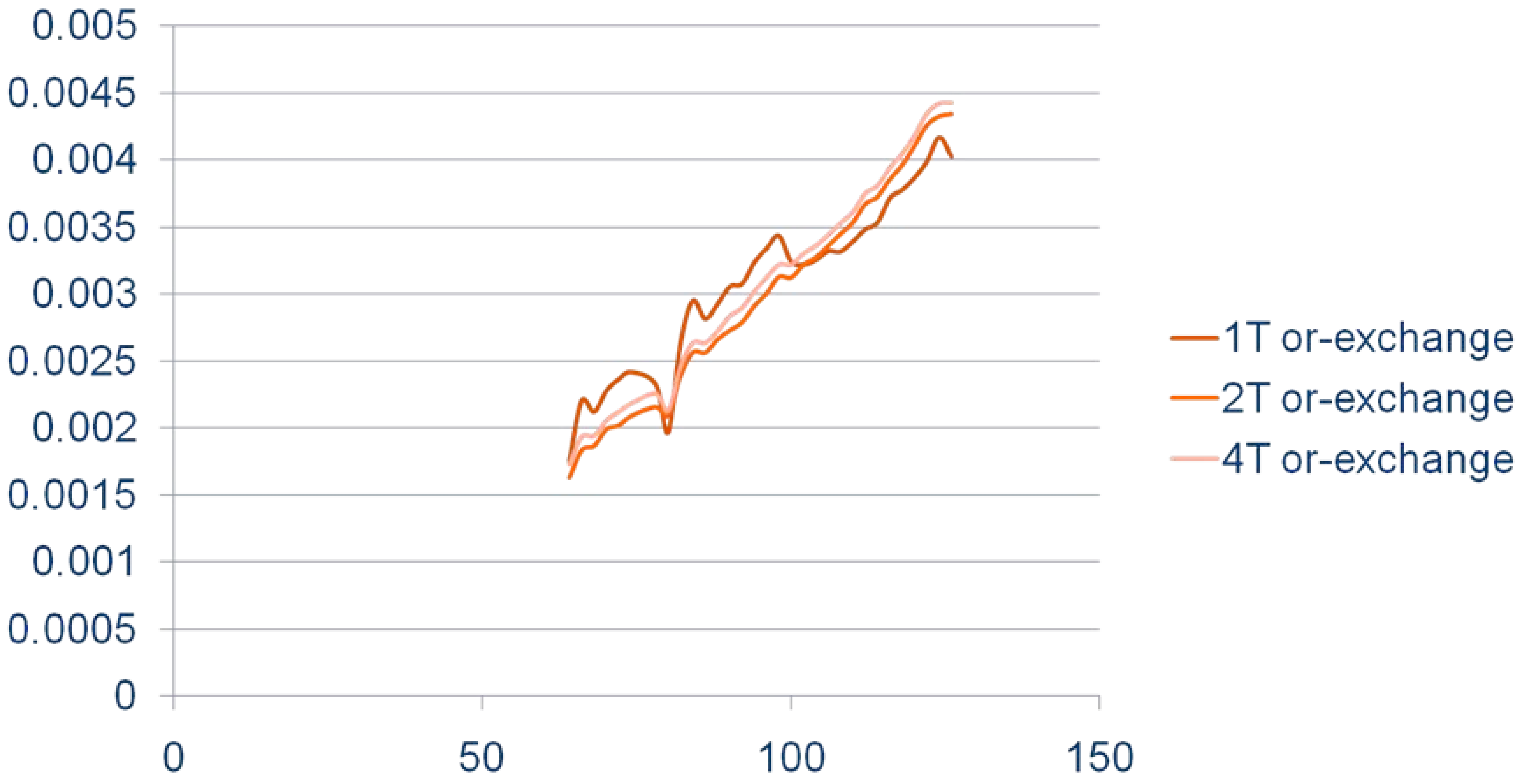
Relocate results



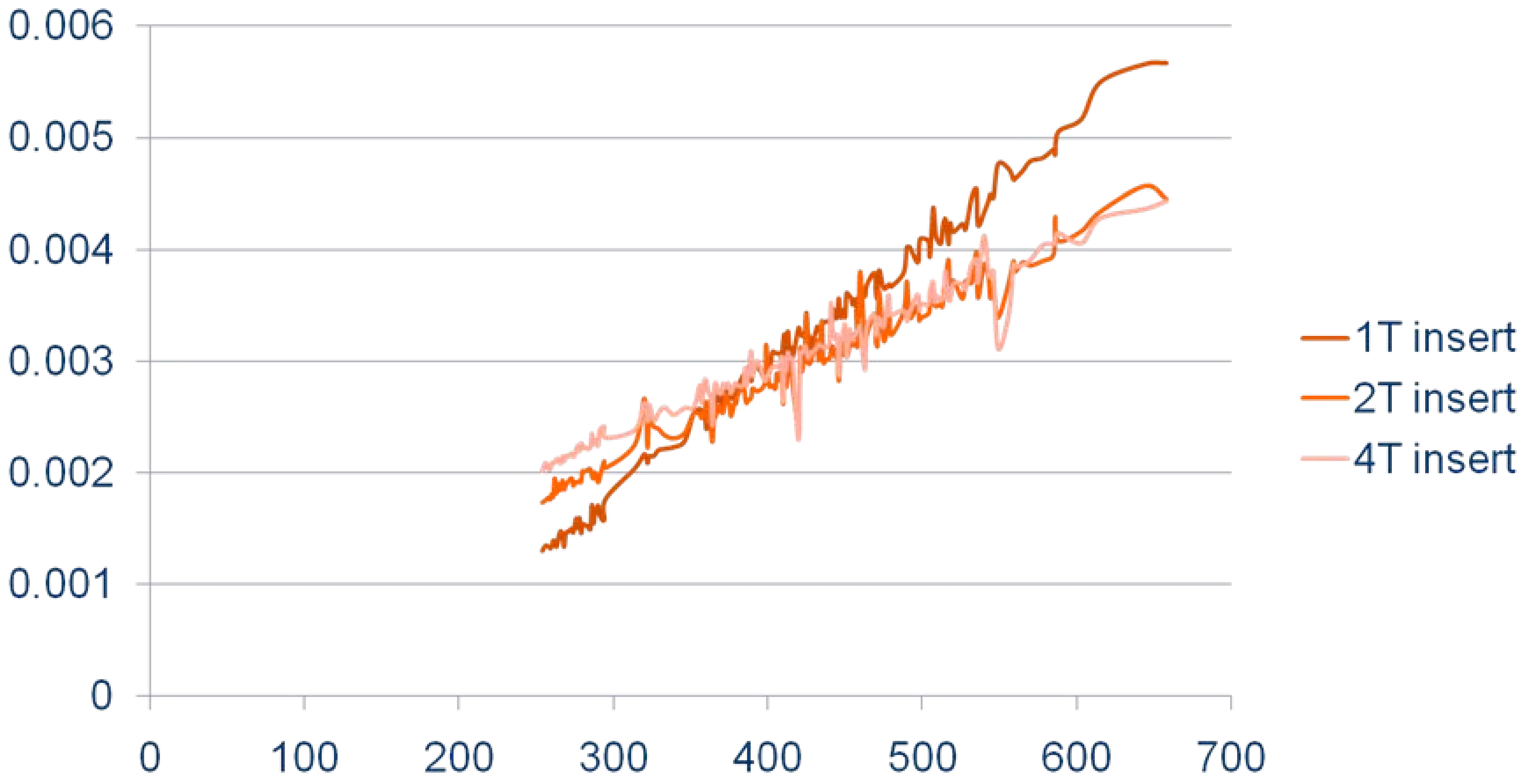
2 opt results



Or exchange results



Insert results



Possible explanation

- Sequential neighbourhood generation taking too much time
- Too many critical sections somewhere
- Slowed by reallocation of prevous 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 unecessary thread synchronisation as it can be costly.
- This implementation needs more tuning and testing. Currently does not appear to scale past 2 threads.