

# Batch Oriented State Space Generation

Stefan Blom

Computational Logic  
Institute of Computer Science  
University of Innsbruck

16. 11. 2005

# Outline

- 1 Introduction
- 2 State Space Generation
- 3 Implementation
- 4 Conclusion

# Outline

- 1 Introduction
- 2 State Space Generation
- 3 Implementation
- 4 Conclusion

# Distributed State Space Generation

## Problems

- It uses many workers for a long time.
- The size is limited by the available memory.

## Solution

- Cut the task into many small independent jobs.
- Use on-disk generation to increase capacity.

# Outline

- 1 Introduction
- 2 State Space Generation**
- 3 Implementation
- 4 Conclusion

# State Space Generation

```
todo := {initial state}
visited := todo
while not empty(todo)
  maybe_new := explore(todo)
  todo := maybe_new \ visited
  visited := visited union maybe_new
```

# Classical Distributed State Space Generation

- Assumes a globally unique hash function on states.
- States are assigned to workers by means of the hash function.
- Workers use a hash table in memory to decide if a state is new.

# Batch Oriented State Space Generation

- Assumes a globally unique order on states.
- States can be assigned based on pivots.
- The database of visited states is an *ordered* list.

# Basic operations

## explore Takes

- list of states

## Produces

- list of transitions
- sorted lists of destination states

## zip Takes

- old sorted list of visited states
- sorted lists of destination states

## Produces

- new sorted list of visited states
- lists of new states

# Algorithm

```
visited:=[initial state]
todo:=[[initial state]]
while not empty(todo)
    dest := map explore todo
    (visited,todo) := zip(visited,dest)
```

# Outline

- 1 Introduction
- 2 State Space Generation
- 3 Implementation**
- 4 Conclusion

# Prototype

- Uses Torque batch scheduler.
- Limited to single visited state list.
- In each round:
  - Submit the explore jobs.
  - Submit the zip job, holding for explore jobs to finish.

## Statistics - Lift 6

states	33,949,609
transitions	165,318,222
Begin-End	31 hours
Total size	575MB
Transition files	415
size of state list	55,651,215 bytes
time need for last zip	521.33 sec
Hardware	XP2400/512M

# Scalability issues

- For efficient IO, we need a large block size.
- Sorting random lists works best if all items are in memory.
- Splitting databases means many input/output files.
- On-disk nature makes new-state check very expensive.

# Number of files

- Make explore and zip  $K$ -way distributed.
- If the visited list is  $N$ -way split then:
  - The number of files needed is  $\frac{N}{K}$  (one file per job).
  - Number of open files per process is  $\frac{N}{K^2}$ .

# Improving the new-state check

- Use B-Tree rather than a list to make searching more efficient.
- Using bit-hashing to detect new states might reduce the number of times the list has to be zipped.

# Outline

- 1 Introduction
- 2 State Space Generation
- 3 Implementation
- 4 Conclusion**

# Summary

## Disadvantages

- On-disk nature makes new-state check very expensive.
- Batch scheduler adds latency.

## Advantages

- Scales up to very large clusters.
- Can deal with any size of state space.
- Short jobs allow efficient use of resources.
- Might be possible to use on a GRID.

Questions?