Parallel Fuzzy Qualitative Reasoning

Allan M. Bruce and George M. Coghill
Department of Computing Science
University of Aberdeen
Aberdeen AB24 3UE
e-mail: {abruce, gcoghill}@csd.abdn.ac.uk

Abstract

Qualitative Reasoning offers an approach to analyse system behaviours where standard numerical techniques are impossible or impractical. The theory behind Qualitative Reasoning is very simple, however current implementations are not very efficient. We have developed a Qualitative Reasoning engine with a novel parallel abstract architecture. Recognising several stages within the engine to be executed in parallel allows the engine to be run on multiprocessor machines or in distributed computing environments and is scalable. This should allow the engine to be executed much faster and make qualitative reasoning an option for future applications. Results of the parallelisations show that execution time has been decreased by an order of two in these stages when run on a small multi-processor machine.
Keywords: Qualitative Reasoning, Parallel Systems, Fuzzy Systems.

1 Introduction

The development of the GRID [1] and the ongoing evolution of web services has meant that distributed computing now has the potential to be carried out on a global scale. This has presented a great opportunity for the construction of AI tools utilising these resources, which is being actively pursued. Web services offer a portable cross-platform interface which allows tools to be easily accessed remotely.

In this paper we present a novel software architecture for Fuzzy Qualitative Reasoning, instantiated as a system named JMorven. JMorven is designed to capitalise on these developments; both by making direct use of the GRID for parallel processing and providing a web resource for use in larger systems such as model learning, diagnosis or planning.

The system described in this paper was inspired by the work of Platzner and Rinner [2–4] on parallelising QSIM [5]. While developing a C version of QSIM in order to achieve a more optimal version, they noted that certain parts of QSIM would benefit from parallelisation, therefore they developed a dedicated hardware architecture which resulted in what is effectively a QSIM machine. While dedicated hardware is unlikely to provide a realistic way forward, the design ideas are sound and JMorven builds on these both by abstracting the engine and extending the degree of parallelisation.

This paper is organised as follows: Section 2 introduces qualitative reasoning and Morven [6]. Work undertaken by Platzner and Rinner to optimise QSIM is detailed in section 3. Section 4 outlines the inspiration for implementing the qualitative reasoner and how parallelisations were achieved. Results of the parallelisations are shown and discussed in section 5. Section 6 draws some conclusions from the work completed, and finally some future work is proposed in section 7.

2 Fuzzy Qualitative Reasoning

Mathematical analysis has been used for centuries to describe the behaviour of systems. This makes use of precise quantities which need to be well specified, however these precise quantities are not always known. Qualitative Reasoning [5–9] (QR) offers an approach to analysing systems with incomplete or imprecise knowledge. As such, QR can be used when quantitative mathematical techniques cannot. There are several domains in which QR has been used including diagnosis, learning, planning and control [10–14].

The simplest representation in QR is when quantities fall into one of three ranges: positive, zero, and negative. To include extra information about the behaviour of variables over time, derivative information can also be included. For example, if a single tank full of water has the plug removed, then initially the volume of water would be qualitatively [+, -], i.e. there would be a positive volume of water but the rate of change of volume is negative. Eventually this system would equilibriate to [0, 0] i.e. there would be no water in the tank and there would be no change in the volume.

2.1 QSIM

One of the most popular qualitative reasoning systems is Qualitative Simulation (QSIM) developed by Kuipers [5]. QSIM is a constraint based QR package using *Qualitative Differential Equations* to specify the constraints. Variables in QSIM are represented by a $<qmag, qdir>$ pair where *qmag* denotes the qualitative magnitude of the variable which consists of either a landmark value or an interval within the given range. The rate of change of the variable is expressed by *qdir* which can take one of the three values, *inc* for increasing, *dec* for decreasing or *std* for steady. The *Constraint-filter* is used to ensure the qualitative states created are consistent with the

constraints and a pairwise Waltz-filter [15] is used to ensure the model is consistent across all constraints which discards any tuples conflicting across any pair of constraints. To allow QSIM to simulate a model's behaviour over time, transition rules are used describing how variables transit depending on their magnitude and direction.

2.2 FuSim

QSIM was the main inspiration for the development of a Fuzzy Qualitative Simulation package called FuSim [8]. Fuzzy reasoning deals with uncertainty whereas QR deals with imprecision therefore combining the two approaches is thought to increase the scope of such a system. FuSim represents fuzzy numbers as paramaterised four-tuples as detailed below:

$$\mu_A(x) = \begin{cases} 0 & x < a - \alpha \\ \alpha^{-1}(x - a + \alpha) & x \in [a - \alpha \quad a] \\ 1 & x \in [a \quad b] \\ \beta^{-1}(b + \beta - x) & x \in [b \quad b + \beta] \\ 0 & x > b + \beta \end{cases}$$

These fuzzy four tuples are used to create *Fuzzy Quantity Spaces*. A quantity space in FuSim is a set of overlapping four-tuples which span a finite range as shown in figure 1. An α-cut is used in FuSim to aid simulation - this is where a fuzzy quantity space is converted to a non-overlapping crisp quantity space by selecting a 'typical' membership value, α.

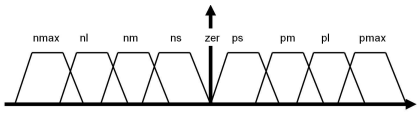

Figure 1: Fuzzy quantity space showing nine quantities.

To use fuzzy four-tuples the standard arithmetic operators have been defined as shown in table 1. Once these arithmetic operators have been applied to fuzzy numbers, the resulting propagated value needs to be mapped back to a relevant quantity space (the predicted values). The *Approximation Principle* is used to do this, which merely states that any quantities in the quantity space overlapping the propagated fuzzy value are an approximation to it. Obviously some values may be more suited to this approximation, so a *distance metric* is used to determine how close the approximation is for a given propagated value mapped back into the quantity space. This technique also allows the prioritisation of quantities which is utilised during simulation.

FuSim uses a similar variable representation to QSIM. However the derivative is not restricted to three values as in QSIM, it can instead take on any value in the quantity space specified (or a specific quantity space purely for that derivative if desired). This allows the model to be analysed more precisely over time and also allows FuSim to make temporal calculations. Due to the finite number of quantities in each quantity space, FuSim creates a state to describe the model at a given time. A behaviour is described as a set of these states in a tree with each node representing a valid state and each edge a valid transition.

Operation	Result	Conditions
$-n$	$(-d, -c, \delta, \gamma)$	all n
$\frac{1}{n}$	$\left(\frac{1}{d}, \frac{1}{c}, \frac{\delta}{d(d+\delta)}, \frac{\gamma}{c(c-\gamma)}\right)$	$n >_0 0, n <_0 0$
$m + n$	$(a + c, b + d, \tau + \gamma, \beta + \delta)$	all m, n
$m - n$	$(a - d, b - c, \tau + \delta, \beta + \gamma)$	all m, n
$m \times n$	$(ac, bd, a\gamma + c\tau - \tau\gamma, b\delta + d\beta + \beta\delta)$	$m >_0 0, n >_0 0$
	$(ad, bc, d\tau - a\delta + \tau\delta, -b\gamma + c\beta - \beta\gamma)$	$m <_0 0, n >_0 0$
	$(bc, ad, b\gamma - c\beta + \beta\gamma, -d\tau + a\delta - \tau\delta)$	$m >_0 0, n <_0 0$
	$bd, ac, -b\delta - d\beta - \beta\delta, -a\gamma - c\tau + \tau\gamma)$	$m <_0 0, n <_0 0$
	$m = [a, b, \tau, \beta], n = [c, d, \gamma, \delta]$	

Table 1: Arithmetic primitives used in FuSim

FuSim, like QSIM, is a non-constructive QR system, i.e. the algorithm uses the transition rules to determine the set of successor values and then filters these with the constraints and the pair-wise filter.

2.3 Morven

Morven [6], formerly known as Mycroft, is a qualitative reasoning framework built on ideas developed in FuSim and adding several novel features. Morven uses a constructive approach to qualitative analysis which lends itself better toward simulation and helps reduce spurious behaviour generation. Due to its being constructive, Morven requires that constraints are causally ordered. This is where a constraining variable must not appear before it has been constrained in the ordering.

One major limit of QSIM and FuSim was the use of only one derivative per variable. Morgan introduced the concept of *Qualitative Vectors* [16] which inspired *Vector Envisionment* [17] which allows a non-fixed number of derivatives to be used, including reasoning purely with the magnitude of a variable. Morven built on this and introduced *Fuzzy Vector Envisionment* [6] which reasons about a non-fixed number of fuzzy derivatives. With the inclusion of multiple derivatives, a system has to incorporate a method to be able to constrain these extra derivatives. *Differential Planes* [18] are used to add these extra constraints. Differential planes also have the advantage that the model complexity may be reduced for higher derivatives if the extra detail is not required. An example of differential planes is shown below for the single tank system:

plane 0:	C0:	$q_o = M^+(V)$
	C1:	$V' = q_i - q_o$
plane 1:	C0:	$q'_o = M^+(V')$
	C1:	$V'' = q'_i - q'_o$

Where V is the volume of water in the tank and q_i, q_o represent the inflow and outflow of water respectively.

3 Parallel QSIM

One disadvantage of Qualitative Reasoning is that current implementations are not very efficient [2] and can take a long time to analyse the behaviour of complex models. Platzner and Rinner decided to try and optimise the popular package QSIM to make it more efficient and therefore appeal to a wider audience. QSIM was originally developed in LISP

so their first optimisation achieved was when porting QSIM to C. They found that this typically decreased execution time by approximately three to four times for models running on the C version over the LISP version on the same hardware setup. These results were encouraging so they sought more optimisations.

The Constraint-filter iterates through all tuples from the constraints and only consistent and valid tuples remain. Each constraint is considered in turn and for every possible combination of values a variable may take, the Tuple-filter checks for consistency with the single constraint. If a combination is found to be inconsistent that tuple is discarded, however if the tuple is found to be consistent then the Tuple-filter does not discard it. The Tuple-filter and a Waltz-filter together make up the Constraint-filter.

The form-all-states stage takes all valid tuples from the constraint filter and generates all possible unique states for the total set of constraints.

Platzner and Rinner decided to use a dedicated hardware machine which would execute the large number of instructions more quickly than a standard processor. During the design and implementation of the tuple filter, they found that the tuples could be filtered independently of each other. This would mean that all tuples could be constrained in their own parallel stage which would speed up execution greatly. The Tuple-filter was redesigned for the dedicated parallel hardware platform using up to seven DSP co-processors as shown in figure 2 [3].

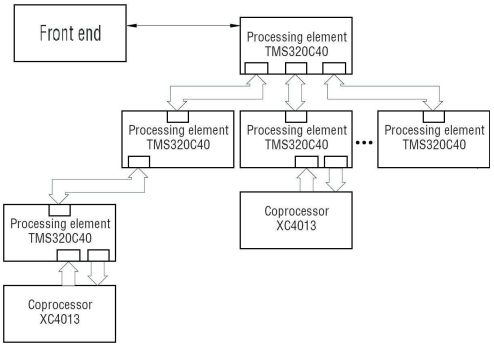

Figure 2: The Architecture of Parallel QSIM.

The form-all-states stage was also parallelised, but using a different technique which partitions the search space into smaller sub-searches; for more information the reader is directed to [2–4].

The results were positive offering a good performance increase. As mentioned earlier, the basic C implementation was approximately three to four times faster than the original LISP version. While using the dedicated hardware parallel system, a further speed increase of up to five times was observed (when using seven processors). These results were very good, although there are a few drawbacks with their design: The implementation was suited only for a dedicated hardware system using these DSP chips, which limits the user base considerably. Also, the implementation was only tested on up to seven parallel units therefore a speedup model cannot be obtained to show the benefits of parallelisation on a large scale. As mentioned earlier, QSIM has the limitations of only reasoning about one derivative and uses crisp quantities. A new parallel QR engine was designed and implemented to overcome these limitations.

4 JMorven

JMorven is a Java implementation of a Qualitative Reasoning engine based on Morven and re-written with a novel abstract parallel architecture (see fig. 3). As with Morven, JMorven uses Qualitative Differential Equations across Differential Planes to specify models. For example, the following differential equation

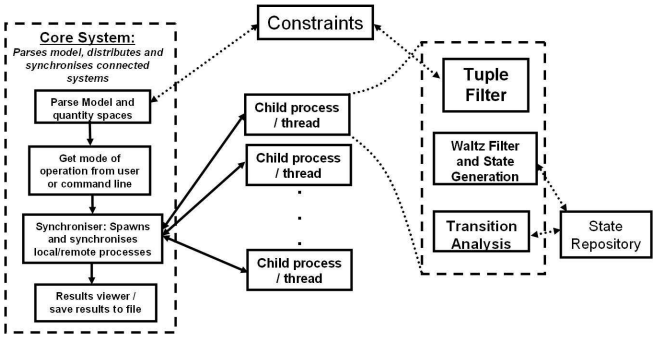

Figure 3: The JMorven Parallel Architecture.

$$V' = q_i - q_o$$

is specified as follows for use with JMorven as follows:

Constraint: sub (dt 1 V) (dt 0 qi) (dt 0 q0)

The first keyword after 'Constraint:' specifies the type of constraint. The subtraction constraint requires three variables to be specified - the result, the variable to subtract from and the variable to subtract. Variables are specified by (dt DERIV VARNAME) where DERIV specifies the order of the derivative for the variable (zero denotes the magnitude of the variable) and VARNAME specifies which variable is in the constraint. All constraints are specified per differential plane allowing the detail of higher order derivatives to be reduced for speed of execution. Since JMorven uses Fuzzy Vector Envisionment (which allows a variable number for the maximum order of derivative for a variable), the number of derivatives in the constraint are not restricted, however every derivative must be constrained by the constraints.

JMorven also uses *Fuzzy Quantity Spaces* to specify the fuzzy quantities within the system. The example quantity space shown in figure 1 would be specified in JMorven as follows:

n-max	-1	-1	0	0.1
n-large	-0.9	-0.75	0.05	0.15
n-medium	-0.6	-0.4	0.1	0.1
n-small	-0.25	-0.15	0.1	0.15
zero	0	0	0	0
p-small	0.15	0.25	0.15	0.1
p-medium	0.4	0.6	0.1	0.1
p-large	0.75	0.9	0.15	0.05
p-max	1	1	0.1	0

Quantities are specified by the following: QNAME $a\ b\ \alpha\ \beta$ (where $a\ b\ \alpha\ \beta$ form the paramaterised four-tuple as described in section 2). JMorven also uses the *Approximation Principle* discussed in section 2 to map calculated quantities from constraints back into the relevant quantity space.

4.1 Parallelisation

Taking our motivation from Platzner and Rinner's work in parallelisation, JMorven utilises several parallel stages to increase efficiency. The JMorven architecture was written from scratch and therefore allowed a novel parallel design to be easily implemented. The new architecture is abstract and therefore scalable allowing JMorven to make use of multi-threaded machines, multiprocessor systems or to run in distributed computing environments. Being written in Java, JMorven is also very portable which increases the potential number of users. The design allows the number of parallel units to be specified at run-time making optimal use of the resources available. During the design of JMorven, all three stages of *Qualitative Analysis* have been identified as parallelisable. These are detailed below. Figure 4 shows a flow diagram of how JMorven is executed for different modes of operation.

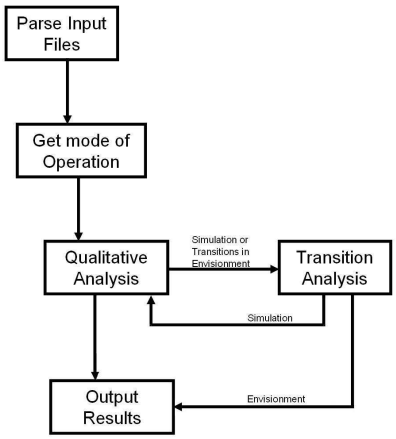

Figure 4: Flow Chart of JMorven.

4.2 Tuple-filter

The Tuple-filter was parallelised in a similar manner to that of Platzner and Rinner where each tuple can be filtered independently of the others. JMorven iterates through each constraint in turn obtaining a set of valid tuples from each. For example, from our single tank example above, one of the constraints is:

$$V' = q_i - q_o$$

which states that the rate of change of the volume of water is equal to the difference between the inflow and outflow. This constraint only reasons about the first derivative of V and the zeroth derivatives of q_i and q_o. A valid tuple from this constraint may include:

$[V'\ q_i\ q_o]$ = [*p-small p-large p-medium*]

A number of valid tuples will be produced for each constraint in the model. Since each constraint can be filtered independently, they can be executed in their own parallel unit. JMorven creates a new thread for each constraint. If there are more constraints than the maximum number of available threads, JMorven queues constraints until a thread becomes available. A diagram of how the *Tuple Filter* is parallelised is shown in figure 5.

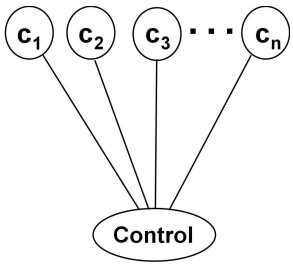

Figure 5: The Tuple Filter in parallel. C_x denotes each constraint in the system which is filtered in its own thread.

4.3 Pairwise-filter

After the *Tuple filter* there are a large set of constraint-tuples, and each constraint-tuple is a vector of valid fuzzy quantities similar to the one shown above. To ensure that the tuples are consistent over all constraints, a pairwise-filter is used. which involves pairing all possible constraint-pairs that are adjacent (two constraints are said to be adjacent if they each share a common derivative of a variable). Each pairing then iterates through all possible tuples and discards those that are not common to both. For example, if we have constraints C1 and C2 from the single tank example with tuples as shown:

C1: $[V'\ q_i\ q_o]$ = [*p-small p-large p-medium*]
C2: $[V\ q_o]$ = [*p-medium p-medium*]

where C2 is the one to one mapping representing a monotonic increasing function between V and q_o. The only common variable to both of these constraints is q_o which is consistent for the given values (*p-medium* in both constraints) therefore the filter would keep this pair, however if C2 = [*p-large p-large*] then the pair would be discarded as q_o would be inconsistent across the pair of constraints.

Each pair of constraints can be executed independently of each other, therefore can be parallelised. One way to implement the pairwise-filter is to first create an exhaustive list of all possible pairs of constraints for the model, and then create a new thread for each pair (queuing pairs if the maximum number of threads has been reached until one becomes free as before). JMorven incorporates the Waltz-filter in the State-generation stage as explained in the next section.

4.4 State-generator

The State-generator is the most computationally expensive stage of *Qualitative Analysis*. This is the process of iterating through each set of tuples and creating unique states for

every combination of variables' derivatives possible. JMorven combines the pairwise-filter described above within this stage to optimise performance. To create these unique states, JMorven uses a recursive technique to enumerate the constraints. All of the tuples within a constraint are considered in turn. If the pairwise-filter discards the tuple, the next tuple is considered. If there are no more tuples left in the current constraint the iteration stops and the end of this recursion is met allowing the previous constraint's iteration to continue. If and when the pairwise-filter finds a consistent tuple, the next constraint is considered from its first tuple (unless the current constraint is the last one). If this is the last constraint and a tuple is consistent then a unique state is created. This procedure carries on until there are no more tuples left in the first constraint. The following pseudo-code shows the process described above:

```
constraint c=0
function:  recurse(int c)
{
   iterate t tuples in constraint c
   {
      if tuple t is consistent
      {
         if c is last constraint
            create unique state
         else
            recurse(c+1) //next constraint
```

To parallelise this stage, the first recursive step is broken down into an iterative step, and each iteration is spawned in its own thread (note that all tuples in the first constraint are valid since no other constraints have been set, therefore there is no need to check the validity of these tuples). The iteration is shown below:

```
iterate i through tuples in c=0
{
   c = 1
   recurse(c)
}
```

This allows the state-generation to run in parallel as shown in figure 6. The first constraint is chosen to be the one which has the number of tuples closest to the number of available threads. It can be seen that a filter is included in the State-generation which negates the need for a separate Waltz-filter, leading to a decrease in execution time.

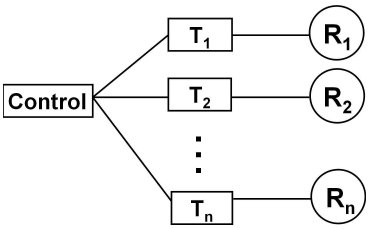

Figure 6: The State Generator in parallel. T_x shows each thread which executes the Recursive function R_x.

4.5 Transition Analysis

The Transition Analysis (TA) phase involves determining how qualitative states transit between one another. This is achieved by following *Transition Rules* which assume that all transitions are continuous. The TA phase has been implemented in JMorven however no parallelisations have been attempted yet. This will form part of future work still to be completed.

5 Results & Discussion

To test JMorven in a distributed computing environment, the GRID network is intended to be used via the Globus Toolkit [1]. This allows many machines to communicate in a virtual network and share resources allowing an amount of processing power usually only achieved by supercomputers. So far, JMorven only runs on single machines with one or more processors. JMorven was executed on a ten processor SUN server running Solaris 5.8 with Sun Java 1.4.2_03. Two test models were used. For testing the Tuple-filter a coupled

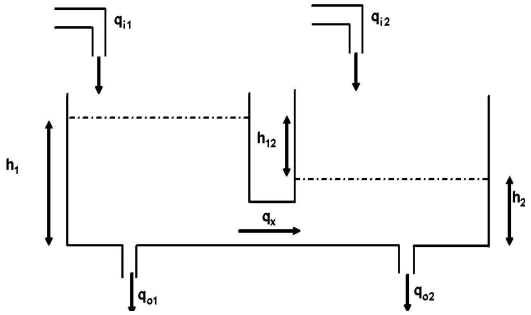

Figure 7: Coupled tanks model showing two tanks of water with heights h_1, h_2 and there difference h_{12}. Two inflow taps q_{i1}, q_{i1} and two outflow plugs q_{i1}, q_{i1} determine the flow in and out of the tanks and the cross-flow q_x describes the flow between them.

tanks model (see fig. 7) with two inputs and two outputs was used as shown below (only the first differential plane):

```
Constraint: sub (dt 0 h12) (dt 0 h1) (dt 0 h2)
Constraint: func (dt 0 qx) (dt 0 h12)
Constraint: func (dt 0 qo2) (dt 0 h2)
Constraint: func (dt 0 qo1) (dt 0 h1)
Constraint: sub (dt 0 q1flow) (dt 0 qi1) (dt 0 qx)
Constraint: add (dt 0 q2flow) (dt 0 qi2) (dt 0 qx)
Constraint: sub (dt 1 h1) (dt 0 q1flow) (dt 0 qo1)
Constraint: sub (dt 1 h2) (dt 0 q2flow) (dt 0 qo2)
```

The *sub* and *add* constraints are organised by having the result in the first variable specified. The *func* constraint is a qualitative function where values can be mapped from the left variable to the right variable which allows many types of function to be implemented. For these models, these merely define the monotonic increasing function (M^+).

For testing the State-generator a coupled tanks model was also used, but with only one input (to tank A) and one output (from tank B) as shown below:

Constraint: func (dt 0 qo) (dt 0 h2)
Constraint: func (dt 0 qx) (dt 0 h12)
Constraint: sub (dt 0 h12) (dt 0 h1) (dt 0 h2)
Constraint: sub (dt 1 h1) (dt 0 qi) (dt 0 qx)
Constraint: sub (dt 1 h2) (dt 0 qx) (dt 0 qo)

The quantity spaces used for both consisted of nine fuzzy intervals as detailed in section 1. Each model was run ten times for each number of threads, and results show the mean speedup achieved.

The results of the Tuple-filter are shown in figure 8. It is clear to see that there is a benefit from the parallelisations. There was quite a large error in times recorded for the Tuple-filter - this is due to the very small execution time. The Tuple-filter takes well under one second to complete for this model. Using six threads over one almost decreases the execution time by a factor of two, which is less than expected. This is probably due to the very small amount of time taken for the Tuple-filter. When the time taken is this small, the overhead of creating and killing threads becomes apparent. A much more complex model should show larger benefits from the parallelisations, and will form part of future work.

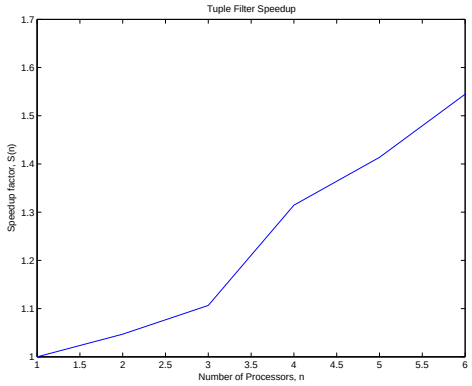

Figure 8: Execution times of the Tuple-Filter. Shows speedup factor of Tuple-filter for multiple numbers of processors using a coupled tanks qualitative model with two inputs and two outputs.

The State-generator results are shown in figure 9. The benefit of the parallelisations is apparent for a smaller number of threads - using four threads over one almost halves the running time of the State-generator. This benefit is less obvious when using a larger number of threads. This is thought to be due to the model used since the State-Generator splits the first constraint into threads. The first constraint chosen may have only a few valid tuples when recursed to the next constraint. For example, if there are ten valid tuples in the first constraint (C0) then ten threads will be spawned and each thread starts the recursion in constraint C1. One of these threads might not have any consistent tuples in C1 therefore would terminate after a very short time. However one of the other threads may have several valid tuples in the constraint C1 and would require recursion to the next constraint for each of them therefore this thread may take substantially longer to execute. Optimising these parallelisations should allow a better advantage from them which will form part of future work to be undertaken.

Another possibility for the benefits to be less apparent than expected might be due to the amount of semaphoring required to protect the data from corruption when accessing it from more than one thread simultaneously. For a distributed computing environment, semaphoring would not be required as all data would be copied for each process, which should allow more benefit from the parallelisations.

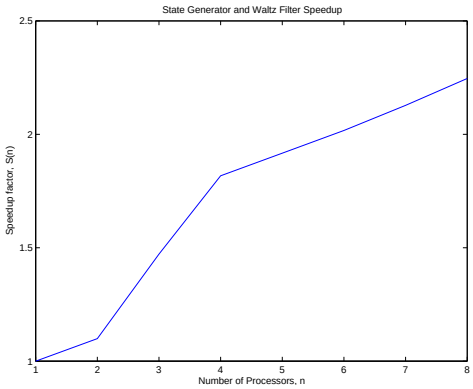

Figure 9: Execution times of the Waltz-filter and State-generator. Shows speedup factor of State Generator and combined Waltz-filter for multiple processors of threads for a coupled tanks qualitative model with one input and one output.

6 Conclusion

JMorven was written in Java to allow it to be portable and run on a wide variety of systems. JMorven has been successfully tested on WindowsXP (SP1 and SP2), MacOS 10.1, Solaris 5.8 and Fedora Core 2. Due to the abstract parallel architecture, JMorven can make use of the best available resources, be it multiple processors or machines in a distributed computing environment.

Parallelisations have been found in all three stages of *Qualitative Analysis* which offer a good speed increase. Execution time has been halved for a small number of processors in the parallel stages for the models tested and greater benefits should be apparent for more complex models and in distributed computing environments. The optimal speedup for a parallel system is known as a linear speedup [19] which states that execution time decreases linearly with the number of parallel units used, or the sequential time[1] should remain constant independent of the number of parallel units. JMorven does not experience this as not all parallel units have the same amount of processing to carry out, however it is clear from the results that parallelising is a viable technique to decrease execution time.

[1] sequential time = parallel time * no. of processors

7 Future Work

The *Transition Analysis* stage is thought to be parallelisable, this will be one area of future work of JMorven. This will be complex to implement due to the nature of the data. Transitions require an initial state to be analysed to determine the next possible states, and these states require to be present in the envisionment[2]. This means that there is a lot of shared memory being accessed at once therefore mutexes will be required to stop data corruption which makes the benefits of parallelisation less apparent.

JMorven will incorporate an interval simulation engine. Trying to parallelise this process will form the basis of some future work.

JMorven was originally intended to be used in a distributed computing environment therefore implementing JMorven with the GRID as discussed above will form another area of future work to be completed. Testing with a larger number of parallel units will provide a better speedup model.

Finally, optimising the parallelisations to achieve closer to linear speedup will be carried out as well as testing more complex models.

References

[1] Globus. http://www.globus.org/, 2005.

[2] M. Platzner and B. Rinner. Parallel qualitative simulation. *Simulation Practice and Theory - International Journal of the Federation of European Simulation Societies*, 5(7-8):623–638, 1997.

[3] M. Platzner and B. Rinner. Toward embedded qualitative simulation. *IEEE Intelligent Systems*, 15(2):62–68, March-April 2000.

[4] M. Platzner and B. Rinner. Design and implementation of a parallel constraint satisfaction algorithm. *International Journal in Computers and Their Applications*, 5(2):106–116, June 1998.

[5] B. Kuipers. Qualitative simulation. *Artificial Intelligence*, 29(3):289–338, September 1986.

[6] G. M. Coghill. *Mycroft: A Framework for Constraint-based Fuzzy Qualitative Reasoning*. PhD thesis, Heriot-Watt University, September 1996.

[7] K. Forbus. Qualitative process theory. *Artificial Intelligence*, 24:85–168, December 1984.

[8] Q. Shen and R. Leitch. Fuzzy qualitative simulation. *IEEE Transactions on Systems, Man and Cybernetics*, 23(4):1038–1061, July-August 1993.

[9] D. S. Weld and J. de Kleer. *Readings in Qualitative Reasoning about Physical Systems*, volume 1. Morgan Kaufmann Publishers, Inc., 1990.

[10] J. de Kleer and B. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):91–130, April 1987.

[2] An envisionment is the exhaustive list of states that a model may exist in as calculated during the *Qualitative Analysis* phase

[11] G. M. Coghill, S. M. Garrett, and R. D. King. Learning qualitative metabolic models. In R Lopez de Mantaras and L. Saitta, editors, *Proceedings of the 16th European Conference on Artificial Intelligence*, pages 445–449.

[12] B. Drabble. Excalibur: A program for planning and reasoning with processes. *Artificial Intelligence*, 62(1):1–40, July 1993.

[13] U. E. Keller. *Qualitative Model Reference Adaptive Control*. PhD thesis, Heriott-Watt University, September 1999.

[14] K. de Koning, B. Bredeweg, J. Breuker, and B. Wielinga. Model-based reasoning about learner behaviour. *Artificial Intelligence*, 117:173–229, March 2000.

[15] D. Waltz. *Understanding Line Drawings of Scenes with Shadows*. McGraw-Hill, New York, 1975.

[16] A. Morgan. *Qualitative Behaviour of Dynamic Physical Systems.* PhD thesis, University of Cambridge, 1988.

[17] G. M. Coghill. Vector envisionment of compartmental systems. Master's thesis, University of Glasgow, April 1992.

[18] M. Wiegand. *Constructive Qualitative Simulation of Continuous Dynamic Systems*. PhD thesis, Heriot-Watt university, May 1991.

[19] R. Greenlaw, H. J. Hoover, and W. L. Ruzzo. *Limits to Parallel Computation*. Oxford University Press, Oxford, UK, 1995.