Autonomous Mobile Programs

Xiao Yan Deng, Phil Trinder and Greg Michaelson
School of Mathematical and Computer Sciences
Heriot-Watt University, Edinburgh, EH14 4AS,Scotland,
{xyd3,trinder,greg}@macs.hw.ac.uk

Abstract

To manage load on large and dynamic networks we propose Autonomous Mobile Programs (AMPs) that periodically use a cost model to decide where to execute in the network. Unusually this form of autonomous mobility affects only where the program executes and not what it does. We present a generic AMP cost model, together with a validated instantiation and comparative performance results for two AMPs. Experiments on a homogeneous network show that collections of AMPs quickly obtain and maintain optimal or near-optimal balance. The advantages of our decentralised approach are scalability to very large and dynamic networks, improved balance, and guaranteed maximum overhead. The disadvantages are higher overheads and the necessity of both a cost model and explicit mobility control.

1 Introduction

Most distributed environments are shared by multiple users. In particular, distributed agent-based systems must also contend with external competition for resources, not least for the processing elements they share. However, agent mobility in such distributed systems tends to be driven by concerns relating to the collective goal of the agent system, independent of the actual environment in which the system is running. Thus, such systems tend not to be aware of, or respond to, environmental changes which impact on the effectiveness with which they may contribute to the collective goal.

For example, if the external load on a shared processing element increases, and the agent's activity does not require its presence on that specific processing element, then it may advantageously move to a more lightly loaded processing element without otherwise affecting its behaviour. In the absence of self- and environmental awareness, however, such systems may suffer widely varying processing and response times as the local environment changes, and accurate prediction of their behaviours becomes problematic.

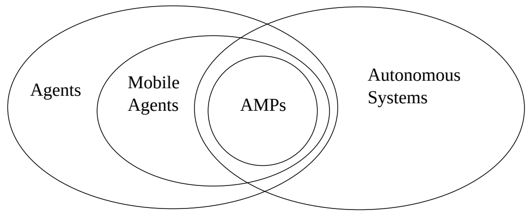

Figure 1: Agents and Autonomous Systems

We have been exploring what we term *autonomous mobile programs(AMPs)* which are aware of their processing resource needs and sensitive to the environment in which they execute. Our experiments suggest that AMPs are able to dynamically relocate themselves to minimise processing time in the presence of varying external loads on shared processing elements.

Our work is novel in that:

- mobility is truly autonomous as the AMPs themselves use local and external load information to determine when and where to move rather than relying on a central scheduler;
- AMPs combine analytic cost models with empirical observation of their own behaviours to determine their current progress;
- The cost of movement may be kept to a very small proportion of the overall execution time.

2 Context

While AMPs have strong connections with both Agents and Autonomous Systems, as shown in Figure 1, they also have important differences.

Firstly, unlike previous mobile agents approaches [11, 7, 4], AMPs have cost models and are autonomous, making decision themselves when and where to move according to the cost model. AMPs also differ from tradition autonomous systems [6, 1, 9], which use schedulers to decide whether to move. For AMPs there is no scheduler at all: AMPs themselves can make the decision when and where to move according to the cost model.

Some autonomous systems are based on mobile agents [11] which can migrate from one location to another. AMPs are similar to ethological models like Ant Algorithms[8] which are distributed computations using mobile agent technology. Both systems are searching for resources; in Ant algorithms for ”food”. But where an Ant algorithm is going to find the fastest path to get to the food, AMPs are going to find resources and decide which one is better.

Next, where most autonomous mobile agent systems adapt the computation, AMPs adapt their coordination. According to conditions given by the programmer, AMPs decide when and where to move by checking the environment where they are executing.

Finally, a collection of AMPs performs decentralised[13] dynamic load balancing. This differs from traditional static and dynamic load balancing systems such as LSF [12] and Sun Grid Engine [5]. Firstly, each AMP collects load information and decides when and where to move rather than the decision being taken by a special load balancer process or processes. Secondly, AMP autonomous load balancing may operate on a dynamic network. Finally, autonomous load balancing aims to *minimise execution time of the applications*. But the goal of dynamic load balancing is *maximising utilisation of the processing power*.

3 A Cost Model for AMPs

For AMPs a cost model is used to inform the decision whether to move to a new location. Cost model are typically parameterised on: *system architecture* including *processor* speed and *interconnect speed*; *cost of processing data*; *cost of communicating data*; *data size*; and *number of processors*. In the AMP cost model the total execution time of a process, T_{total}, has three components:
T_{total} = T_{comp} [+ T_{comm}] [+ T_{coord}]

- T_{comp} is the computation time for finishing the task.
- T_{comm} is the communication time for migrating to another location.
- T_{coord} is the coordination time for getting or exchanging status informations with other processes or system.

Figure 2 shows the generic cost model for auto-mobile programs. Formula (2) gives the condition under which the program will move, i.e. if the time to complete in the current location is more than the time to complete in the remote location.

Formula (3) states that if there are m communications in a program’s lifetime and it will take T_{comm} time for each

$$T_{total} = T_{Comp} + T_{Comm} + T_{Coord} \tag{1}$$
$$T_h > T_{comm} + T_n \tag{2}$$
$$T_{Comm} = mT_{comm} \tag{3}$$
$$T_{Coord} = npT_{coord} \tag{4}$$
$$T_{Coord} < OT_{static} \tag{5}$$
$$n < \frac{OT_{static}}{pT_{coord}} \tag{6}$$
$$T_e = W_d/S_h \tag{7}$$
$$T_h = W_l/S_h \tag{8}$$
$$T_n = W_l/S_n \tag{9}$$
$$W_a = \sum W_d \tag{10}$$
$$W_d = W_{a(this)} - W_{a(last)} \tag{11}$$
$$W_l = W_{all} - W_a \tag{12}$$

O	:	Overhead e.g. 5%
T_{total}	:	total time
T_{static}	:	time for static program running on the current location
T_{Comm}	:	total time for communication
T_{comm}	:	time for a single communication
T_{Coord}	:	total time for coordination
T_{coord}	:	time for coordination with a single processor(location)
T_{Comp}	:	time for computation
T_e	:	time has elapsed at current location
T_h	:	time will take here
T_n	:	time will take in the next location
W_{all}	:	all work
W_a	:	the total work which has been done
W_d	:	the work has been done at current location
W_l	:	the work left
S_h	:	the current CPU speed
S_n	:	the next location CPU speed
m	:	number of communication
n	:	number of coordination
p	:	number of processor

Figure 2: Generic Cost Model for Auto Mobile Programs

communication then the total time for communication is T_{comm} by m.

Formula (4) states that if there are p processors and n checks on the status of the processors in a program's lifetime and it will take T_{coord} time for checking one processor once then the total time for coordination is T_{coord} by p by n.

Formula (5) gives the condition under which the program will do the coordination work. This condition guarantees that the autonomous mobile program performance will never be worse than $100 + O$ percent of the static version. This guarantee is only valid providing that the loads on the current and target location don't change dramatically immediately after the move. For example it is easy to construct a pernicious example where each time an AMP moves to a location the load on that location becomes very high.

Substituting (4) in (5) we get Formula (6), where n specifies how many times we can consider moving.

Formulas (7), (8) and (9) relate time, work and CPU speed. The time equals the work measured by CPU speed.

In formulas (10), W_d is the work that has been done at one location, so the total work is the sum of all the W_d. In other words, formula (11) shows that the work done at the current location equals the total work done before the program moved to the current location($W_{d(last)}$) minus all the work that has been done ($W_{d(this)}$ or W_d).

Formula (12) gives the remaining work, that is the total work minus all the work that has been done.

4 Individual AMP Evaluation

4.1 Matrix Multiplication Cost Model & Validation

Following initial experiments[2] with building AMPs in the functional mobile language Jocaml[4], we are now using the more mainstream Voyager[10], a version of mobile Java which supports weak mobility. We have developed two AMPs in Voyager, matrix multiplication and ray tracing. Here we focus on matrix multiplication. An auto-mobile matrix multiplication program has been developed using three for loops:

```
for(int i=0;i<work;i++){     //first level
  for(int j=0;j<work;j++){   //second level
    for(int k=0;k<work;k++){ //third level
      m3[i][j]=mult(m1[i][k],m2[k][j]);
} } }
```

Cost Model We use the cubic cost model for naive matrix multiplication to instantiate the generic auto-mobile cost model from section 3 as in Figure 3. In equations (17) and (18) Sec is a constant which converts time to seconds. Formula (13) shows that the total work in the matrix multiplication is n^3. Formula (14) shows that the work that has been

$$W_{all} = n^3 \tag{13}$$

$$W_a = f(i,j,k) = (i-1)n^2 + (j-1)n + k \tag{14}$$

$$W_l = W_{all} - W_a = n^3 - f(i,j,k) = n^3 - (i-1)n^2 + (j-1)n + k \tag{15}$$

$$W_d = W_{a(this)} - W_{a(last)} = f(i,j,k)_{this} - f(i,j,k)_{last} \tag{16}$$

$$T_e = \frac{W_d}{S_h} = \frac{[f(i,j,k)_{this} - f(i,j,k)_{last}]\, Sec}{S_h} \tag{17}$$

$$T_h = \frac{W_l}{S_h} = \frac{\left[n^3 - f(i,j,k)\right] Sec}{S_h} \tag{18}$$

$$T_h = \frac{\left[n^3 - f(i,j,k)\right] T_e}{f(i,j,k)_{this} - f(i,j,k)_{last}} \tag{19}$$

$$T_n = \frac{W_l}{S_n} = \frac{\left[n^3 - f(i,j,k)\right] Sec}{S_n} = \frac{S_h T_h}{S_n} \tag{20}$$

Figure 3: Cost Model for Auto Mobile Matrix Multiplication

done is a function of *i, j, k*. Substituting (13) and (14) into (12) we get formula (15). The remaining time for finishing the program is a function of *i, j, k*.

The work that has been done at the current location is the total work done at every location minus the total work done at the last location, giving formula (16). Substituting (16) in (7) we get the time that has elapsed at current location giving formula (17).

Substituting (15) in (8) we get the time it will take at the current location in formula (18). Substituting (17) in (18) we get formula (19). So the time it will take at the current location is a function of *i, j, k* and T_e. Substituting (18) in (8) the time that will be taken in the next location can be predicted as formula (20).

When this cost model is put into a program, the program can determine how much time has elapsed(T_e), and the CPUs speed can be found. So it can predict how much time,T_h, the program will take if it stays in the local location, and how much time, T_n, it will take if it moves to a remote location. According to this information the program can make a decision about whether to move or not.

Execution Time Validation To show that the cost model of elapsed time and remaining time are accurate we evaluated the AMP matrix multiplication against a static version. At every first level loop we use formula (19) to predict the remaining time and the total time for the program. At the end of the program we can get the real execution time and compare the predicted time and real time. Table 1 shows that we achieve accurate predictions of processing time.

Coordination Time Validation In the simple cost model in section 3 the total coordination time is one coordination

Size	Predict	Actual	Std Dev(%)
600*600	9.75	9.86	1.2
700*700	15.75	15.57	1.2
800*800	23.00	23.30	1.3
900*900	32.40	32.97	1.7
1000*1000	45.25	45.72	1.0

Table 1: Execution time validation

Locations	Predict	Actual	Std Dev(%)
3	0.76	0.75	1.0
4	1.01	0.89	11.6
5	1.26	1.15	9.0
15	3.78	3.92	3.9
25	6.30	6.55	4.0

Table 2: AMP coordination time validation

time multiplied by p by n; where p is the number of processors and n is how many times to check the status of the processors($T_{Coord} = npT_{coord}$). For the current model we consider that T_{coord} is a constant, and that a program should not spend much time on this work, which means the smaller the coordination time the better the efficiency.

From experiments on our local Linux network we estimate the coordination time for checking a single location once as: $T_{coord} = 0.25 seconds$. So the total coordination time is: $T_{Coord} = 0.25 * p * n seconds$.

Table 2 shows validation of the AMP coordination time model for matrix multiplication. The predicted time is very close to the actual time: the worst is 11.6% of the actual time and the best is 1.0%.

Communication Time Validation We suppose the time for communication is a function of the size of the matrix(n). So we suppose the time for communication should be:

$$T_{comm} = T_{comm1} + T_{comm2} * n^2 \qquad (21)$$

T_{comm1} is the time for building a connection from the local location to a remote location; we call this the lookup time. T_{comm2} is the time for sending one unit of data to the remote location.

From experiments, we find that the time for look up is a constant, and is not related to the size of the matrix. The time for sending the program to the remote location changes according to the size of matrix, but if the size of the matrix is smaller than 50, the time for sending it is almost constant. If the matrix is bigger than 50, the time for sending is variable; the bigger the size the more time it takes. If the sending time($size > 50$) is divided by n^2(size of matrix) we get a constant. So we get a formula for communication time

Data Size	Predict	Actual	Std Dev(%)
50*50	0.042	0.047	12.5
100*100	0.081	0.079	2.3
200*200	0.236	0.259	9.9
300*300	0.495	0.510	3.0
400*400	0.857	0.840	2.0
500*500	1.323	1.276	3.5

Table 3: AMP communication time validation

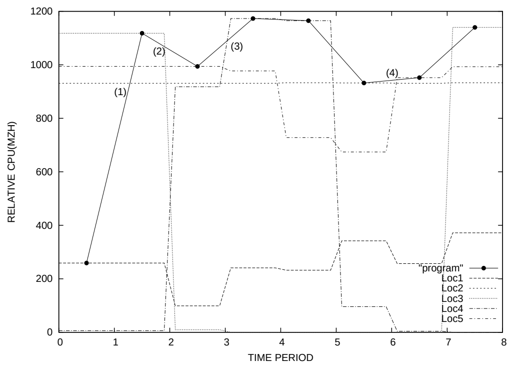

Figure 4: Auto-mobile matrix movement

which is :
T_{comm}=0.029 + (if n $<$ 50 then 0 else 5.07*10^{-6}*n^2).

Table 3 shows validation of our AMP communication time model. The predicted time is very close to the actual time: the worst is 12.5% of the actual time and the best is 2.0%. The bigger the data size, the better the prediction. We have built a similar cost model for a ray tracer AMP. Validation shows comparable results to matrix multiplication.

4.2 AMP Movement

We conducted experiments to test if the program moves as we expect. Figure 4 show the movement of the AMP matrix multiplication during successive execution time periods with CPU speeds normalised by the local loads. Our test environment is based on five locations with CPU speeds (Loc1 534MHZ, Loc2 933MHZ, Loc3 1894MHZ, Loc4 2000MHZ, Loc5 1000MHZ).

We started the mobile programs in time period 0 on $Loc1$. In time period 1 it moved to the fastest processor $Loc3$. When $Loc3$ became more heavily loaded the program moved to $Loc5$, the fastest processor in period 2. In time period 3, $Loc4$ became less loaded, and was the fastest location at that moment, so the program moved to it. In time period 7 $Loc5$ was a little faster than $Loc2$. So the program moved to $Loc5$ rather than staying on $Loc2$. Many AMPs have reproducible behaviour. We get similar results for the AMP ray tracer. Note that while ray tracing is generally

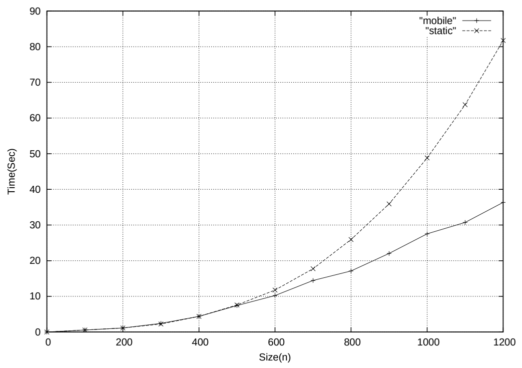

Figure 5: Mobile and Static Matrix Multiplication Execution Time

irregular we evaluated a regular instance.

We draw the following conclusions from Figure 4:

- The program may move repeatedly to adapt to changing loads and always find the fastest location in one step.
- Move (1) shows that if there is a faster location then the AMP moves to it.
- Move (2) shows that AMPs can respond to changes in current location.
- Move (3) shows that AMPs can respond to changes in other locations.
- Move (4) shows that even if the speed differential is small, the AMP moves.

4.3 AMP Performance

Figure 5 compares the execution times of static and mobile matrix multiplication programs. Our test environment is based on three locations with CPU speed 534MHZ, 933MHZ and 1894MHZ. The loads on these three computers are almost zero. We started both the static and the mobile programs on the first CPU. We can see that the bigger the size of the matrix the faster the mobile version is compared with the static version. We get similar results for the AMP ray tracer. From all the experiments, we get comparable results to those for the AMPs in Jocaml[2].

5 Collections of AMPs

In this section, we discuss experimental results for collections of AMPs on both homogeneous and heterogeneous processor networks.

In initial experiments, each AMP obtained load information from all other PEs, but we quickly introduced the

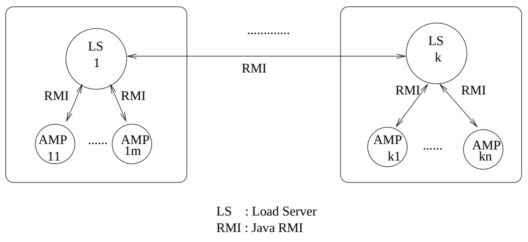

Figure 6: System with load server structure

architecture depicted in Figure 6 where each location has a load server that maintains information about location loads. Specifically, the load server records CPU speed, the number of AMPs and the load of each location. The advantages of the load server architecture are reduced time for AMPs to discover load information, and also reduced network traffic.

For homogeneous systems, we measured AMP behaviour on a dedicated network of four locations all with the same CPU speed (3193MHz) and communications latency.

We initially hypothesised that every location would have an equal number of AMPs, but experiments showed that the *initiating location*, where the AMPs are started is more heavily loaded than the others, and hence has fewer AMPs. We define *optimal balance* as each location having the same number of AMPs, except the initiating location which may have fewer. For small numbers of AMPs the initiating location has just one AMP and other locations have $\frac{a-1}{p-1}$ AMPs, where a is the total number of AMPs, p is the total number of locations.

Our experiments show that collections of AMPs achieve optimal balance as predicted in Table 4, where the first row is the number of AMPs started, the first column is the number of locations used, and the remaining columns summarise the distribution of AMPs on the locations, with the initiating location listed first. For example, if we run seven AMPs on three locations and we start all seven AMPs on $Loc1$, then we expect that there will be three AMPs on both $Loc2$ and $Loc3$, but just one AMPs on $Loc1$.

For illustration the movement of 7 AMPs between 3 locations is shown in Figure 7. The AMPs are started on $Loc1$ in time period 0. Four AMPs moved to $Loc2$ and two moved to $Loc3$ in time period 1, which is not an optimal balance, so one AMP in $Loc2$ moved to $Loc3$ in time period 2. After this move the system achieved an optimal balance and the AMPs did not move again. We have also done experiments of five AMPs on three location, nine AMPs on three locations, seven AMPs on four locations, ten AMPs on four locations and get similar results.

Near-optimal balance is when each location except the initiating location may not have the same number of AMPs,

AMPs	5 AMPs	7 AMPs	9 AMPs	10 AMPs
3 Locs	1/2/2	1/3/3	1/4/4	-
4 Locs	-	1/2/2/2	-	1/3/3/3

Table 4: Verified Optimal Balance

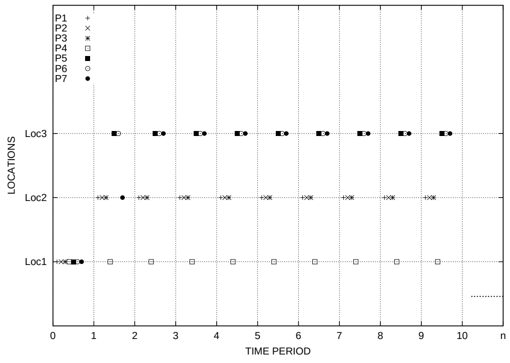

Figure 7: 7 AMPs on 3 Locations

but the number of AMPs on each location differs by no more than one. In Figure 8, we started six AMPs on $Loc1$. Three of them moved to $Loc3$, two of them moved to $Loc2$, one stayed on $Loc1$, and the AMPs do not move again after the best distribution. Figure 9 shows that if we add AMPs to or remove AMPs from a balanced AMPs system, the AMPs can rebalance themselves. Seven AMPs were initially started on $Loc1$. Once the system was balanced, we started one AMP in each time period of 4, 6 and 8, and we got balance in time period 12. The rebalancing as AMPs are removed follows a similar patten.

We have also measured the behaviour of multiple AMPs on a heterogeneous network of fifteen locations, with CPU speeds 3193MHz (Loc1-Loc5), 2168MHZ (Loc6-Loc10), and 1793MHz (Loc11-Loc15). For illustration the movement of 25 AMPs between the 15 locations is shown in Figure 10. "B" is the *balanced status*, where every AMP has similar *relative CPU speed*. In this state, AMPs will stay in the current locations and not move any more until the balance is broken. We started 25 AMPs on Loc1 in time period "0". After some movements of each AMPs, we achieved a balance in time period "k" and the AMPs maintained the balance and did not move any more until time period "k+x", when one of the AMPs is finished and the balance is broken. So the 24 AMPs moved again and reached a new balance in time period "l". Figure 11 shows that every AMP has effective resource between 150MHZ and 400MHZ with only three exceptions. Similar results were achieved when there are 24 AMPs and 23 AMPs. Loc1 is the busiest location for AMP coordination and there is only 1 AMPs on it. The results show that collections of independent AMPs rebalance quickly and with a small number of moves. In a homogeneous system, if the ratio of AMPs to locations is ideal, an optimal balance is relatively quickly obtained with every location, except the initiating location hosting the same number of AMPs. Similarly, in a homogeneous system, even if the ratio of AMPs to locations is not ideal, an near-optimal distribution can be obtained. Furthermore, the system maintains balance as AMPs are added or removed. Finally, in a heterogeneous system, AMPs can achieve balance with similar relative CPU speeds.

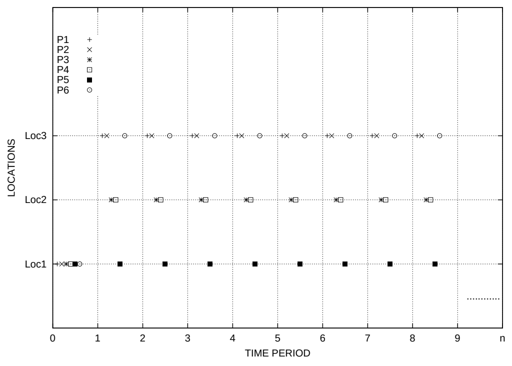

Figure 8: A Near Optimal Balance

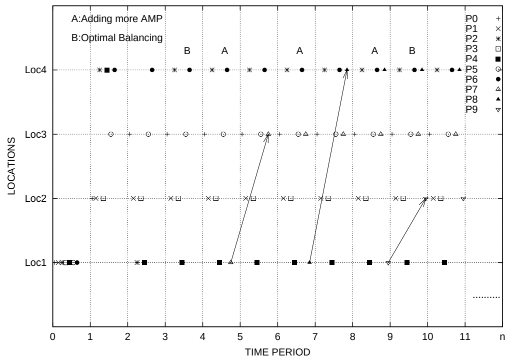

Figure 9: Rebalancing After Adding AMPs

6 Conclusion and Future Work

The advantages of an AMP model are as follows. The model scales to medium sized networks (<15 Locations), with AMPs making decentralised decisions about where to execute. Indeed on such networks only nearby locations need be considered as potential targets. The model manages dynamic networks very easily with each AMP selecting where to execute from the current set of locations, and abandoning any location that is leaving the network.

The AMP model can obtain a better balance than a classical distributed load balancer as, unlike the latter, each AMP has a cost model giving accurate information about the time

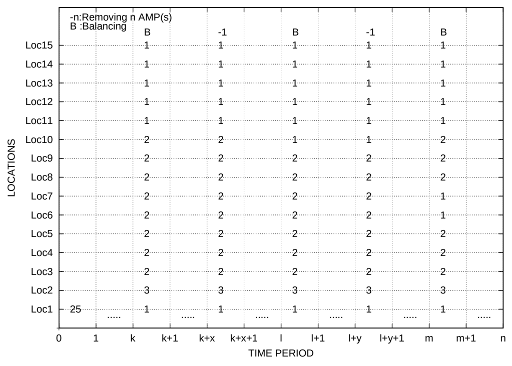

Figure 10: 25 AMPs on Heterogeneous Network (15 Locations)

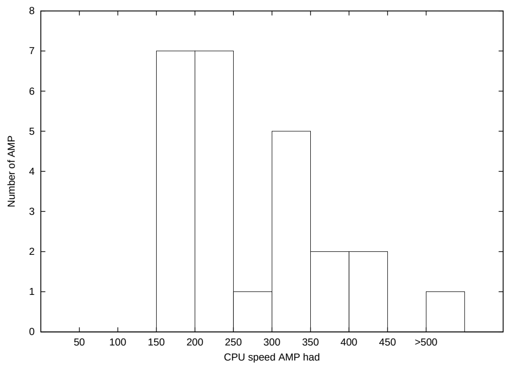

Figure 11: Relative CPU Speed for AMPs

to complete and to communicate the program. Moreover it is possible to parameterise the AMP cost model with a maximum overhead, e.g. 5%, and guarantee under a reasonable assumption that auto-mobility overheads will not exceed it.

However, the disadvantages of an AMP model are that it may introduce higher coordination costs as every location must obtain load information about other locations. AMPs also require an accurate model of computation and communication costs. Finally the program contains additional code to control the autonomous mobility.

There are a number of areas for future work. Firstly, we are developing *auto-mobile skeletons*[3] that encapsulate the mobility control for common patterns of computation. Auto mobile skeletons are polymorphic higher order functions like `automap` or `autofold` that make mobility decisions by combining generic and task specific cost models. Secondly, so far we have only considered regular problems, such as matrix multiplication, and wish to generalise AMPs to irregular problems with cost models and strategies to adapt to the behaviour of irregular programs. Thirdly, the collections of AMPs experiments have been performed on a LAN, and we aim to experiment with large-scale system e.g. computational Grids.

References

[1] J. Abawajy. Autonomic Job Scheduling Policy for Grid Computing. In *Computationl Science - ICCS 2005, part 3*, pages 213–220. Springer, May 2005.

[2] X. Y. Deng, G. Michaelson, and P. Trinder. Towards High Level Autonomous Mobility. In H.-W. Loidl, editor, *Draft proceedings of Trends in Functional Programming*, pages 97–112, Munic, Germany, November 2004.

[3] X. Y. Deng, G. Michaelson, and P. Trinder. Autonomous Mobility Skeletons. *Journal of Parallel Computing*, accepted March 2006.

[4] C. Fournet, F. L. Fessant, L. Maranget, and A. Schmitt. Jocaml: a Language for Concurrent Distributed and Mobile Programming. In *Proceedings of the Fourth Summer School on Advanced Functional Programming*, pages 19–24, St Anne's College, Oxford, August 2002. Springer-Verlag.

[5] A. Group. Sun's Grid Computing Solutions Outdistance the Competition, May 2002. http://www.sun.com/software/grid/docs/Grid_competitive.pdf.

[6] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[7] D. B. Lange and M. Oshima. Mobile agents with Java: The Aglet API. *World Wide Web*, 1(3):111–121, 1998. http://dx.doi.org/10.1023/A:1019267832048.

[8] P. E. Merloti. Optimization Algorithms Inspired by Biological Ants and Swarm Behavior. Technical report, San Diego State University, Artificial Intelligence, CS550, San Diego, June 2004.

[9] C. V. Travis Desell, Kaoutar El Maghraoui. Load Balancing of Autonomous Actors over Dynamic Networks. page 90268.1, 2004.

[10] T. Wheeler. Voyager Architecture Best Practices. Technical report, Recursion Software, March 2005. http://www.recursionsw.com/Voyager/2005-03-31-Voyager_Architecture_Best_Practices.pdf.

[11] M. Wooldridge. Agent-based software engineering. *IEE Proceedings Software Engineering*, 144(1):26–37, 1997.

[12] WWW. Platform. http://www.platform/products/LSF.

[13] A. Y. Zomaya and Y.-H. Teh. Observations on Using Genetic Algorithms for Dynamic Load-Balancing. *IEEE Trans. Parallel Distrib. Syst.*, 12(9):899–911, 2001.