Autonomous Mobility Skeletons

Xiao Yan Deng * Greg Michaelson Phil Trinder

School of Mathematical and Computer Sciences
Heriot-Watt University, Edinburgh, EH14 4AS,Scotland

Abstract

To manage load on large and dynamic networks we have developed Autonomous Mobile Programs (AMPs) that periodically use a cost model to decide where to execute. A disadvantage of directly programming AMPs is that the cost model, mobility decision function, and network interrogation are all explicit in the program. This paper proposes *autonomous mobility skeletons* that encapsulate self-aware mobile coordination for common patterns of computation over collections. Autonomous mobility skeletons are akin to algorithmic skeletons in being polymorphic higher order functions, but where algorithmic skeletons abstract over parallel coordination, autonomous mobility skeletons abstract over autonomous mobile coordination. We present the `automap`, `autofold` and `autoiter` autonomous mobility skeletons, together with performance measurements of Jocaml, Java implementations on small networks. `autoiter` is an unusual skeleton, abstracting over the `Iterator` interface commonly used with Java collections.

Key words: skeletons, mobile computation, autonomous mobile programs
1991 MSC: 68w15, 68w40

1 Introduction

Classical distributed load balancing mechanisms are centralised and control a fixed set of locations. Such mechanisms are not appropriate for dynamic or very large scale networks. We have developed Autonomous Mobile Programs (AMPs)[4,3] that periodically make a decision about where to execute in a network. The decisions are informed by cost models that measure current

* Corresponding author.
Email addresses: `xyd3@macs.hw.ac.uk` (Xiao Yan Deng), `greg@macs.hw.ac.uk` (Greg Michaelson), `trinder@macs.hw.ac.uk` (Phil Trinder).

```
for i = 0 to n-1 do                    (*first level*)
  checkmove();
  for j = 0 to n-1 do                  (*second level*)
    for k = 0 to n-1 do                (*third level*)
      m3.(i).(j) <- m3.(i).(j)+m1.(i).(k)*m2.(k).(j);
    done  done;  done ;;
```

Fig. 1. Direct Autonomous Mobile Matrix Multiplication

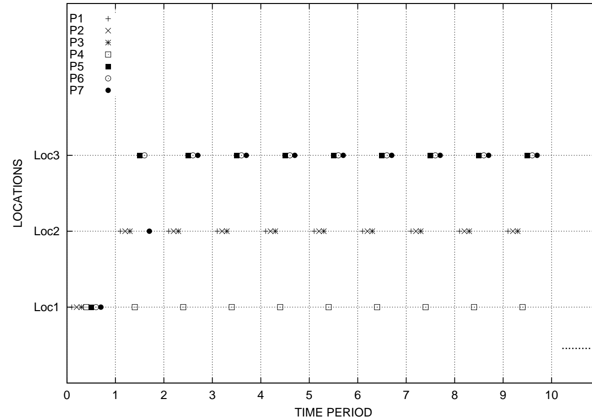

Fig. 2. AMP Load Management, 7 AMPs on 3 Locations

performance, the relative speeds of alternative network locations, and communication costs. Unlike autonomous mobile agents that move to change their function or *computation*, an AMP always performs the same computation, but moves to change *coordination*, i.e. to improve performance.

For example an autonomously mobile matrix multiplication program can be constructed by inserting a `checkmove` function into the outer for loop, as shown in Figure 1. The `checkmove` function interrogates the network to discover available locations, their processor speed and load. This information is used to parameterise cost models to determine whether to move. The program moves if the predicted time to complete at the current location (T_h) exceeds the time to move to the best available location (T_{comm}) and complete there (T_n), i.e.

$$T_h > T_{comm} + T_n \tag{1}$$

Figure 2 shows the load balancing induced by a collection of 7 matrix multiplication AMPs on an homogeneous network where all three locations have the same speed and no other load. All the AMPs are started on Location 1(Loc 1) in time period 0. In time periods 1 and 2, the processes move to to optimise load balance with little change thereafter. Locations 2 and 3 are equally loaded, but as an artefact of the Jocaml implementation Location 1, as the initiating location is less heavily loaded. A comprehensive set of results and analysis are available in [4].

A disadvantage of directly programming AMPs is that the cost model, mobility decision function, and network interrogation are all explicit in the program. This paper explores *autonomous mobility skeletons* that encapsulate mobility control for common patterns of computation over collections. Auto mobile skeletons are polymorphic higher order functions, such as `automap` or `autofold` that make mobility decisions by combining generic and task specific cost models.

This paper presents auto mobile skeletons for the classic higher order functions `map` and `fold` and for the object oriented `Iterator` interface[8]. After describing the skeleton context in section 2.2, autonomous mobility skeletons for the functional mobile language Jocaml are introduced in section 3. In section 4, we discuss the realisation of `automap` and `autofold` in Voyager [11], a mobile version of Java. We next consider `AutoIterator` in section 5. Finally, Section 6 summarises our results and considers future research.

2 Background

2.1 Mobile Computation

Network technology is pervasive and more and more software is executed on multiple locations (or machines). In a mobile language, a programmer controls the placement of code or computations in an open network, e.g. a program can migrate between locations. A typical mobile program is a data mining program that visits a series of repositories to extract interesting information from each repository.

This *software mobility* is in contrast to hardware mobility where programs move on portable devices like PDAs. A number of mobile programming languages have been developed, including Telescript [12], Jocaml [5] and a number of Java variants, e.g. Java Voyager [11] and JavaGo [9].

Fuggetta et. al. distinguish two forms of mobility supported by mobile languages [6]: *weak mobility* is the ability to move only code from one machine to another. *Strong mobility* is the ability to move both code and its current execution state.

2.2 Algorithmic and Mobile Skeletons

Abstract skeletons are higher order constructs that abstract over common patterns of coordination and must be parameterised with specific computations.

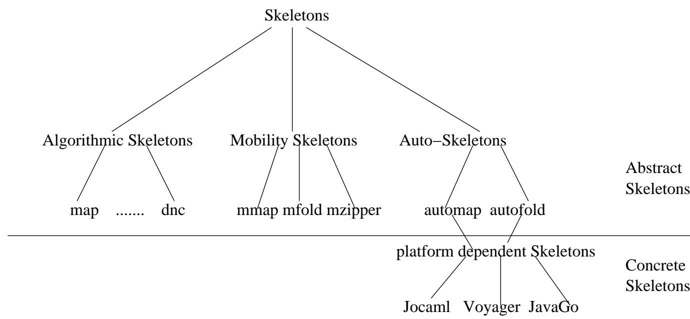

Fig. 3. Skeleton Taxonomy

Concrete skeletons are executable, and the user must link computation-specific code into the appropriate skeleton. Figure 3 shows the relationship amongst different species of skeletons. The notion of *algorithmic skeletons* was characterised by Cole[2] to capture common patterns of parallel coordination in a closed or static set of locations. *Mobility skeletons*[1] are high-level abstractions capturing common patterns of mobile coordination in an open network i.e. a dynamic set of locations. With mobility skeletons the mobile coordination is explicitly specified by the programmer, and the program makes no autonoumous decisions about where to execute. In contrast, `auto-mobile skeletons` are self-aware. Using auto-mobile skeletons the programs can make the decision about when and where to move. So auto-mobile skeletons encapsulate autonomous coordination for common computations over collections, like map, fold or iteration.

In Figure 3 we distinguish between the *abstract* conception of skeletons and their *concrete realisations*. As we shall see, auto-mobile skeletons may have different realisations in languages with different mobile constructs. Specifically the realisation in a language with weak mobility will differ from that in a language with strong mobility. In Figure 3 we distinguish between the *abstract* conception of skeletons and their *concrete realisations*.

The motivation for auto mobile skeletons is to minimise processing time by seeking the most favourable resources, without any requirement to visit specific processors. Thus different concrete realisations of a skeleton may carry out the same computation in a shortest time period with given resources, but the patterns of coordination may be very different. We will explore this further below.

```
let rec dotprod mat1 mat2 =
  match (mat1,mat2) with
    ((h1::t1),(h2::t2)) -> h1*h2+dotprod t1 t2
  | (_,_) -> 0
;;
let inner row col = (dotprod row) col;;
let rowmult row cols = List.map (dotprod row) cols;
;;
let outer cols x = rowmult x cols
;;
let rowsmult rows cols = automap current (outer cols) rows
;;
let mmultMat m1 m2 = rowsmult m1 (transpose m2);;
```

Fig. 4. Jocaml `automap` Matrix Multiplication

3 Jocaml Autonomous Mobility Skeletons

3.1 Jocaml AutoMap

The `automap` auto-mobile skeleton, performs the same computation as the map high order function, but may cause the program to migrate to a faster location. The standard Jocaml map, `map f [a1; ...; an]` applies function `f` to each list element `a1, ..., an`, building the list `[f a1; ...; f an]`. AutoMap, `automap cur f [a1;...;an]` computes the same value but takes another argument `cur`, recording current location information, e.g. CPU speed and load.

For example, Figure 4 shows how the matrix multiplication may be reformulated using automap. At first sight, this looks like a conventional program using `map`. However, as we shall see next, `automap` also includes calls to generic and problem specific cost functions to determine whether or not the program should move.

3.2 AutoMap Design and Implementation

Potentially AutoMap could investigate moving after processing every element of the list, but this induces enormous coordination overheads. Overheads are limited by specifying that the total coordination overhead of the program (T_{Coord}) must be less than some small percentage (O, say 5%) of the execution time of the static, i.e. immobile program, (T_{static}):

$$T_{Coord} < OT_{static} \tag{2}$$

```
let getGran work f h =
    let (fh,fhtime) = timedapply f h
    in let t_static = fhtime * (float (work))
       let t_coord = tcoord (numofhost)
    in let times = (ov * t_static)/t_coord
    in let gran = if times > 0
                  then (work/times)
                  else work+10
    in (fh,fhtime,gran)
```

Fig. 5. `getGran`: Calculating CheckMove Granularity

AutoMap investigates moving after processing `gran` elements, and under the assumption that the automap is the dominating computation for the program `gran` is calculated from the time to compute a single element of the map result, the length of the list, and the overhead percentage O by the `getGran` function in Figure 5.

A generic AMP cost model is used to inform the AutoMap decision about moving to a new location [3]. The cost model determines how much time has elapsed(T_e), and the *relative speed* (CPU speed divided by load) in order to predict the time to complete in the current location T_h. The network is interrogated to discover the relative speeds of available locations and the time to complete at the fastest remote location T_n is calculated. The program moves if the predicted time to complete at the current location exceeds the time to move to the best available location (T_{comm}) and complete there, i.e. $T_h > T_{comm} + T_n$. We have instantiated the generic auto mobile cost model for AutoMap and validated the cost model[3]. The movement check is encoded in the `check_move` function in Figure 6.

The definition of `automap` is given in Figure 7. It first calls `getGran` to calculate an initial granularity and before calling `automap'`. `automap'` applies standard `map` to `gran` elements before calling `getInfo` to evaluate the benefits of a move and to recalculate a `gran`.

The coordination behaviour of the Jocaml AutoMap is depicted in Figure 8. As Jocaml supports strong mobility, the program moves along with its execution state. In the figure, we started a Jocaml program with `automap`, which applies `f` to list `l` in location 1 (1). `automap` will decide where the program moves or not automatically. So the whole program moves to location 2 with its data and context (2). In location 2, the `automap` consumes the input list (3), and produces a result list (4).

```
let check_move cur work workleft fhtime=
  let t_comm = tc work
  let t_h = fhtime * (float (workleft))
  in map (check_relspeed cur) hostlist
     let host_next = check_next cur hostlist
  in let t_n =
      if  (cur <> host_next)
      then (cur.relspeed)/(host_next.relspeed) * t_h + t_c
      else t_h
  in
      if (t_h > t_n)
      then (
            go host_next
            host_next
           )
      else cur
```

Fig. 6. `getGran`: Calculating CheckMove Granularity

```
let automap cur f l =
  let work = List.length l
  in let (fh,fhtime,gran) = getGran work f (hd l)
  in fh::automap' cur work (work-1) gran fhtime f t

let rec automap' cur work workleft gran fhtime f l =
   let xs = List.map f (take (gran-1) l)
   let (h::t) = drop (gran-1) l
   in let (cur',gran', fhtime',fh') =
              getInfo cur work workleft gran fhtime f h
   in xs@(fh'::automap' cur' work (workleft-gran) gran' fhtime' f t)

let getInfo cur work workleft gran fhtime f h=
       let cur' = check_move cur work workleft fhtime
       let (fh',fhtime',gran') = getGran work f h
       in  (cur', gran', fhtime',fh')
```

Fig. 7. Jocaml AutoMap

3.3 Jocaml AutoMap Performance

Figure 9 shows the execution times of the matrix multiplication program implemented using `automap`. Using `automap`, our test environment is based on three locations with CPU speeds 534MHZ, 933MHZ and 1894MHZ. The loads on these three computers are almost zero. We started both the static and the mobile programs on the slowest CPU. Figure 9 shows the result for matrix multiplication, from which we can see the bigger the size of the matrix the

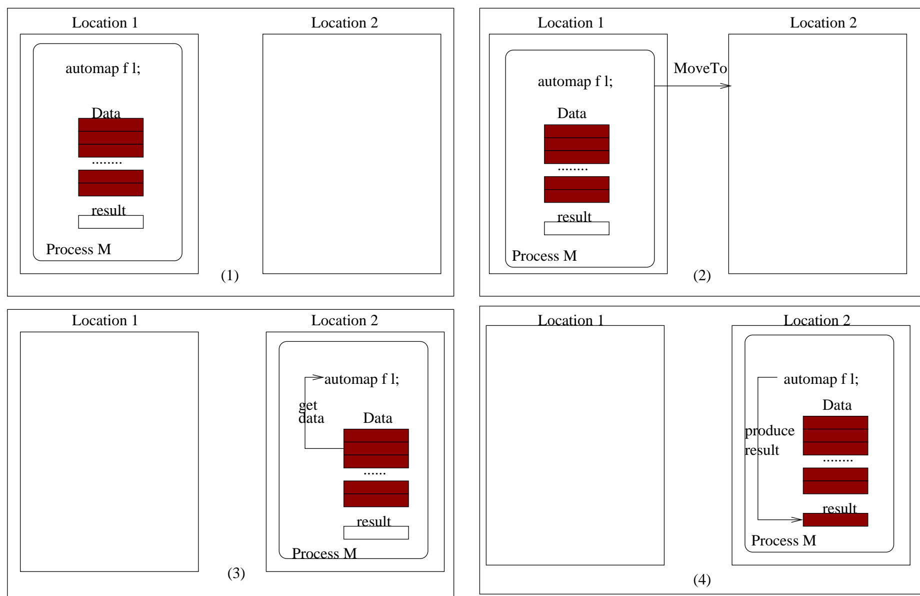

Fig. 8. Coordination Behaviour of Jocaml AutoMap

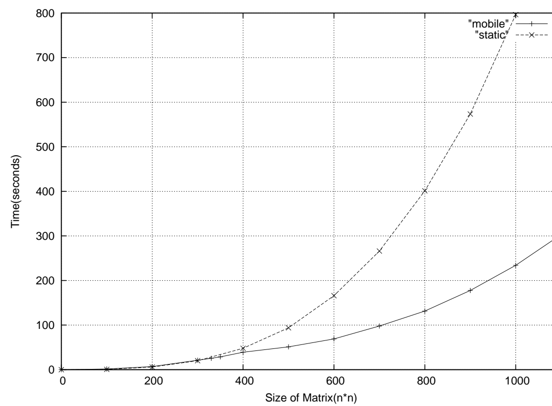

Fig. 9. Jocaml Matrix Multiplication Execution Times

faster the mobile version is compared with the static version. If the matrix is smaller than a certain size (here 330), the mobile version stays on the current location, which is because it will take more than $O\%$ (overhead) of the time for completing at the current location if the program does coordination and move. So at this size, the program does not check information and move at all, and the mobile program takes almost the same time as the static program. If the size of matrix is bigger than 330 then the mobile program moves to the fastest location, and then stays there, so the mobile program takes much less time than the static program.

```
let autofold cur f accu l =
          let work = List.length l
          in let (fh,fhtime,gran) = getGran work (f accu) h
          in autofoldl' cur work (work-1) gran fhtime f fh t

let rec autofold' cur work workleft gran fhtime f accu l =
   let xs = fold f accu (take (gran-1) l)
   let (h::t) = drop (gran-1) l
   in let (cur',gran', fhtime',fh') =
              getInfo cur work workleft gran fhtime (f xs) h
   in autofoldl' cur work (workleft-gran) gran' fhtime' f fh' t
```

Fig. 10. Jocaml AutoFold Definition

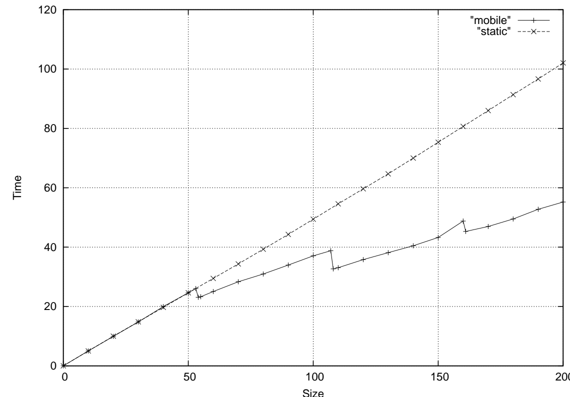

Fig. 11. Jocaml Coin Counting Execution Times

3.4 Jocaml AutoFold

The standard fold in Jocaml, `fold f a [b1; ...; bn]`, computes `f (... (f (f a b1) b2) ...) bn`. AutoFold is `autofold cur f a [b1;...;bn]` computes the same value but may migrate to a faster location. The definition of `autofold` is given in Figure 10

Figure 11 shows the execution times of static and autofold-based versions of a coin counting program using a genetic algorithm[7]. As before, once the program has a sufficiently large execution time, it benefits from moving to a faster location.

4 Java Autonomous Mobility Skeletons

It is appealing to implement Java autonomous mobility skeletons as Java is a popular language and there are numerous mobile Java variants. Voyager[11] is

```
public Object[] automap (Superclass obj, Object[] l){
  Object[] resultl = new Object[l.length];

  long timestart = 0;
  long timeend = 0;
  long fhtime = 0;
  int work = l.length;
  int gran = work;
  int checkPos = 0;

  ISuperclass proxy = (ISuperclass) Proxy.of(obj);
  IMobility mobility = Mobility.of(proxy); //bulid mobility

  for(int i=0;i<work;i++){ // map
    timestart = System.currentTimeMillis();
    resultl[i] = proxy.mapf (l[i]);
    timeend = System.currentTimeMillis();
    if( (i-checkPos) == 0 ){
      fhtime = timeend-timestart;
      gran = getGran (work,fhtime);
      checkPos = checkPos + gran;
      check_move (work,(work-i-1),fhtime,mobility);
    }
  }
  return resultl;
}
```

Fig. 12. Java Voyager AutoMap

a popular Java with weak mobility. Voyager[10] is a platform for distributed application development. It provides a set of basic and advanced services and features for distributed application development. Voyager ORB includes distributed naming service and mobile agent technology. We have developed the two Jocaml auto-mobile skeletons in Voyager, `automap` and `autofold`.

4.1 Java Voyager AutoMap

The Voyager `automap` performs the same computation as, and similar coordination to, the Jocaml `automap`. Figure 12 gives the definition of `automap` in Voyager, where the Java `check_move` and `getGran` auxiliary functions have the same functionality as in section 3.2.

As Voyager supports only weak mobility, when the program moves, it communicates only the code, and not the execution state. Figure 13 shows the coordi-

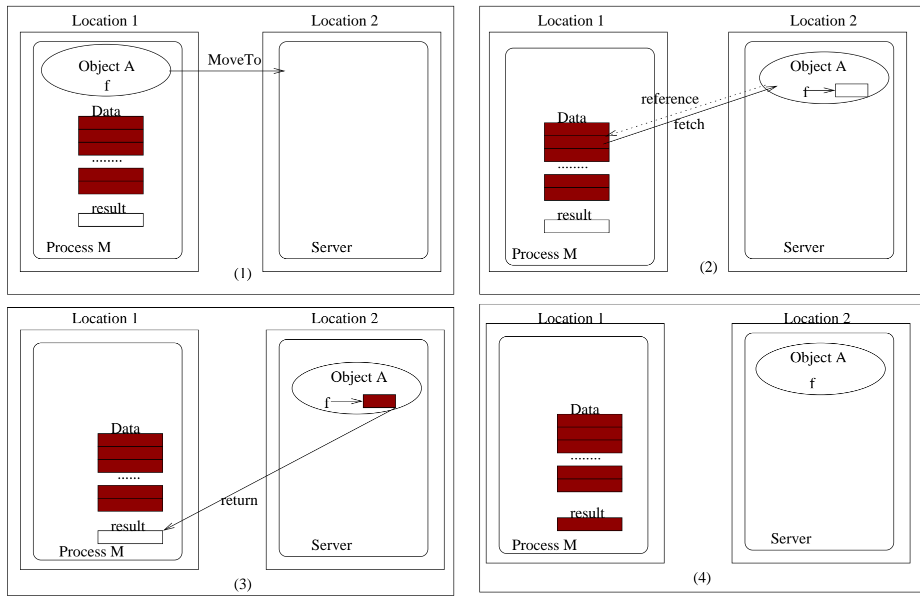

Fig. 13. Coordination Behaviour of Java Voyager AutoMap

nation behaviour of the Voyager `automap`. In the figure, we started a Voyager program with `automap`, which applies `f` in `Object A` to list `l` in location 1. `automap` will make decision where the program moves or not automatically. So the program sends the code of `Object A` to location 2 (1). The system built a reference from location 2 to the data in location 1 (2). In location 2, function `f` fetches data from location 1, produces result and returns result to location 1 (3). After finished the code of `Object A` stays in location 2 and waits for another migration but the data in location 1 will never move (4).

4.2 Voyager AutoMap Performance

An autonomously mobile matrix multiplication is readily written in Voyager Java using `automap`, as in Figure 14. The new class `Auto` has an object `auton`, which includes `automap`. Class `RowMult` has a function `mapf`, which is the function the `map` will apply to the collection. When we do `auton.automap (rowM, mat1)`, `automap` will apply `rowM.mapf` on array `mat1`, at the same time `automap` makes the decision of when and where to move.

Figure 15 shows the execution times of static and automap-based versions of Voyager matrix multiplications, using the apparatus from section 3.3.

```
public static void main (String[] args){
  int[][] mat1 = makeMatrix(size);
  int[][] mat2 = makeMatrix(size);
  int[][] matT = transpose(mat2);

  RowMult rowM = new RowMult(matT);
  Auto auton = new Auto();
  int[][] res = auton.automap (rowM, mat1);
}
```

Fig. 14. Java Voyager Autonomously Mobile Matrix Multiplication

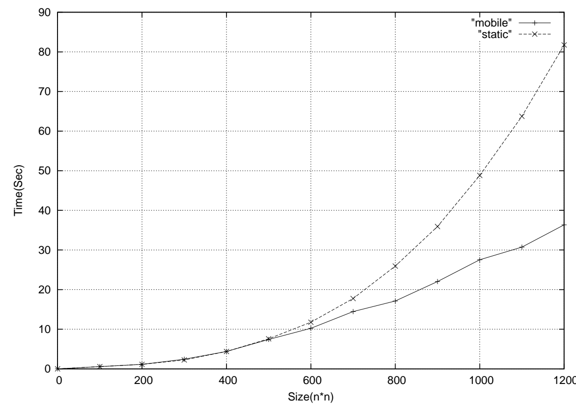

Fig. 15. Java Voyager Matrix Multiplication Execution Times

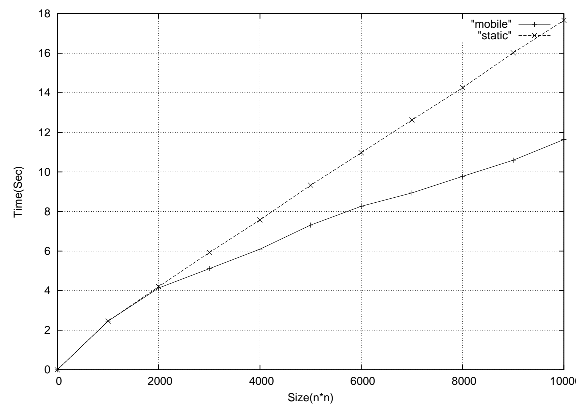

Fig. 16. Java Voyager Coin Counting Execution Times

4.3 Java Voyager AutoFold

An `autofold` is also readily constructed in Voyager Java. Figure 16 shows the execution times of static and autofold-based versions of a Java Voyager coin counting program. These results are again similar to those for the Jocaml autonomous mobility skeletons.

```
public class AutoIterator implements Iterator,Serializable{
  public AutoIterator(ArrayList theList){
    list = theList;
    nextIndex = 0;
    work = list.size();
  }

  public boolean hasNext(){
    return nextIndex < work;
  }
  public Object next() {
     if (nextIndex < work)
        return list.get(nextIndex++);
      else
        throw new NoSuchElementException("No next element");
  }
  private int checkPos = 0;
  private long timestart = 0;
  private long timeend = 0;
  private double fhtime = 0;
  private int gran = work;

  public migratory Object autoNext() {
      if (nextIndex < work){
        if(nextIndex == 0){
          timestart = System.currentTimeMillis();
          timeend = timestart;
        }
        else
          if((nextIndex-checkPos) == 0 ){
            timestart = timeend;
            timeend = System.currentTimeMillis();
            fhtime = timeend-timestart;
            check_move (size,(work-nextIndex-1),fhtime);
            gran = getGran (work,fhtime);
            checkPos = checkPos + gran;
          }
        return list.get(nextIndex++);
      }
      else
        throw new NoSuchElementException("No next element");
  }
  public void remove(){}
}
```

Fig. 17. JavaGo AutoIterator

```
public static void main(String args[]){
  undock {
    String port=null;
    int listlength = Integer.parseInt(args[0]);
    ArrayList al = new ArrayList();
    for (int i=0;i<listlength;i++){
      MatrixMul ii = new MatrixMul();
      al.add(i,ii);
    }
    long timestart = System.currentTimeMillis();
    AutoIterator ai = new AutoIterator(al);
    while (ai.hasNext()){
      MatrixMul iu = (MatrixMul)ai.autoNext();
      int[][] mat = iu.Multiplication();
    }
  }
}
```

Fig. 18. JavaGo Autonomously Mobile Matrix Multiplication

5 An Autonomous Mobile Iterator

An iterator is a class that implements the Java `Iterator` interface, which specifies a generic mechanism to enumerate the elements of a collection. The methods in the interface `Iterator` are `hasNext, next` and `remove`[8]. The `AutoIterator` class implements all three methods, and extends it with `autonext` method, which has the same functionality as `next` but can make autonomous mobility decisions.

AutoIterator requires strong mobility and hence Voyager, with only weak mobility, cannot be used. JavaGo [9] supports strong mobility and Figure 17 shows an AutoIterator implementation again using analogous `check_move` and `getGran` functions.

Figure 18 shows how `AutoIterator` can be used to implement matrix multiplication. Each element of the list is a `MatrixMul` object and includes two matrices and a function `Multiplication`, which multiplies the two matrices. `AutoIterator` enumerates each object using `autoNext` and performs the multiplication.

Figure 19 shows the execution times of static and AutoIterator-based versions of a JavaGo matrix multiplication program.

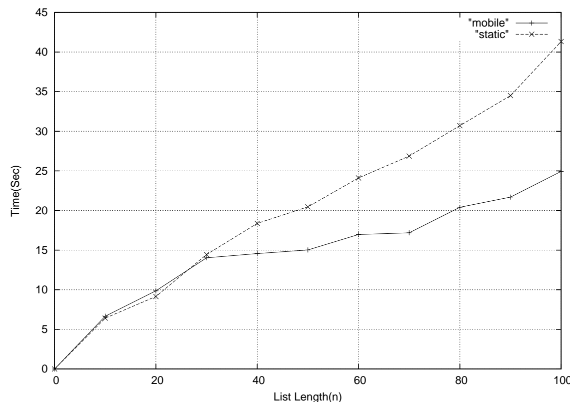

Fig. 19. AutoIterator Matrix Multiplication Execution Times

6 Conclusion

Auto-mobile skeletons encapsulate common patterns of self-aware mobile coordination that minimise execution time in networks with dynamically changing loads. In analogy with other skeleton species, they hide low level mobile coordination details from users and provide higher level loci for designing load-aware mobile systems.

We have demonstrated abstract auto-mobile skeletons with concrete realisations for the common higher-order functions `map` and `fold`, both in the functional language context shared with other skeleton species, through Jocaml, and in an object oriented context through mobile Javas. We have also demonstrated a novel `autoiter` skeleton for the widely used object oriented iterator interface. Our experiments suggest that, for our set of test programs, auto-mobile skeletons can offer considerable savings in execution times, which scale well as overall execution times increase.

With an auto-mobile skeleton the cost model is substantially implicit: predictions are based on the size of the collection and on the time to compute a single collection element. Moreover the predictions are dynamic: the time to compute an element is periodically measured during the execution.

However, our auto-mobile skeletons have a number of limitations. They are based on the assumption that programs using them expose useful loci of mobility in top-level loops that dominate the computation. If auto-mobile skeletons are not deployed at the top level then the program will not be able to take advantage of changes in neighbourhood loads during much of its life.

Furthermore, to make effective use of auto-mobile skeletons, their argument computations must be regular. For irregular computations, closed form analytic cost models tend to introduce inaccuracies.

At present, only one auto-mobile skeleton may be present in a program. We lack appropriate techniques to compose and nest auto-mobile skeletons, as we lack the ability to compose and nest their cost models.

There are two main areas for future work. Firstly, we wish to generalise auto-mobile skeletons to irregular problems with cost models and strategies to adapt to their behaviour. Secondly, we wish to be able to nest and compose auto-mobile skeletons.

To solve both problems, we are exploring a calculus to manipulate, and ultimately extract automatically, continuation cost models that can provide costs for the rest of a computation at arbitrary points during its execution. The advantage of a continuation cost model is that it is not necessary to provide a closed form solution as environmental information for a computation is always available implicitly at run-time. Thus, branches are not necessarily a source of loss of accuracy as concrete data values are available at the point where the cost is calculated. The disadvantage is that a naive cost model may have the same complexity as the computation it models, which, for programs with relatively high coordination and low processing degrees, could add considerably to the overall excution time.

We are experimenting with an early prototype of very simple coster for a tiny language, in Standard ML, but where cost functions are generated in SML rather than in the source language. We next need to look at meta-programming techniques to integrate cost functions into the source language at appropriate checking points.

References

[1] A. R. D. Bois, P. Trinder, and H. Loidl. Towards Mobility Skeletons. *Parallel Processing Letters*, 15(3):273–288, 2005.

[2] M. Cole. *Algorithmic skeletons: structured management of parallel computation*. MIT Press, 1989.

[3] 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*, 2004.

[4] X. Y. Deng, P. Trinder, and G. Michaelson. Autonomous Mobile Programs. Technical report, School of Mathematical and Computer Sciences:Heriot-Watt University, December 2005.

[5] 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.

[6] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.

[7] J.Hawkins and A.Abdallah. A Generic Functional Genetic Algorithm. In P.Trinder and G.Michaelson, editors, *Draft proceedings of the First Scottish Functional Programming Workshop*, pages 151–168, Heriot-Watt University, Edinburgh, 1999.

[8] S. Sahni. *Data Structures, Algorithms, and Applications in Java.* Mc Graw Hill, University of Florida, 2000.

[9] T. Sekiguchi. JavaGo. http://homepage.mac.com/t.sekiguchi/javago/index.html.

[10] R. Software. *Voyager User Guide.*

[11] T. Wheeler. Voyager Architecture Best Practices, March 2005. http://www.recursionsw.com/Voyager/2005-03-31-Voyager_Architecture_Best_Practices.pdf.

[12] J. E. White. Mobile Agents. In J. Bradshaw, editor, *Software Agents*, pages 437–472, Menlo Park, CA, 1997. AAAI/MIT Press.