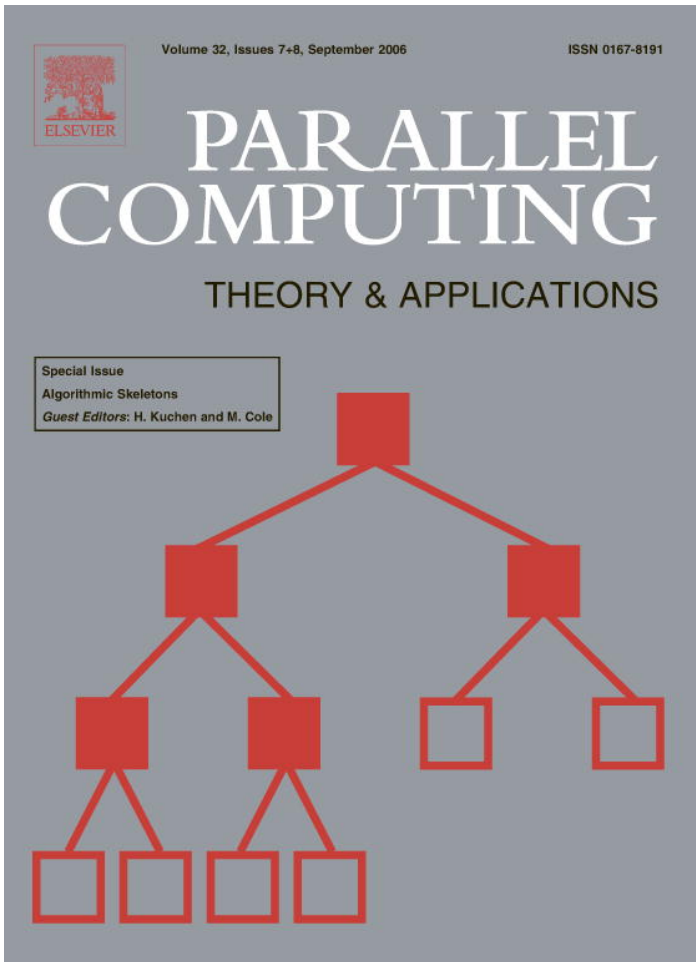
Volume 32, Issues 7+8, September 2006
ISSN 0167-8191
ELSEVIER
PARALLEL COMPUTING
THEORY & APPLICATIONS
Special Issue
Algorithmic Skeletons
Guest Editors: H. Kuchen and M. Cole

Available online at www.sciencedirect.com

ScienceDirect

Parallel Computing 32 (2006) 463–478

PARALLEL COMPUTING

www.elsevier.com/locate/parco

Autonomous mobility skeletons

Xiao Yan Deng *, Greg Michaelson, Phil Trinder

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

Received 23 December 2005; received in revised form 18 May 2006; accepted 30 May 2006
Available online 1 September 2006

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 Voyager, and JavaGo implementations on small networks. `autoiter` is an unusual skeleton, abstracting over the `Iterator` interface commonly used with Java collections.
© 2006 Elsevier B.V. All rights reserved.

Keywords: Skeletons; Mobile computation; Autonomous mobile programs; Jocaml; Java voyager; JavaGo

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 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 move 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 Fig. 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

* Corresponding author.
E-mail addresses: xyd3@macs.hw.ac.uk (X.Y. Deng), greg@macs.hw.ac.uk (G. Michaelson), trinder@macs.hw.ac.uk (P. Trinder).

0167-8191/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.parco.2006.04.002

```
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.

the current location (T_h) exceeds the time to move to the best available location (T_{comm}) and complete there (T_n), i.e.

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

Fig. 2 shows how an auto-mobile matrix multiplication moves between five locations as their relative speeds, i.e., (CPU speed * (100-load)%), change. Load is a percentage of available CPU time used e.g. 60%. The AMP starts at the relatively slow location, Loc1, and immediately moves to the fastest available location, Loc3. When the relative speed of Loc3 drops, it moves again to the new fastest available location, Loc5, and so on.

Fig. 3 shows the load balancing induced by a collection of 7 matrix multiplication AMPs (1000*1000) on an homogeneous network where all four locations have the same speed and no other load. All the AMPs are started on Location 1 in time period 0. In time periods 1 and 2, the processes move to optimise load balance with little change thereafter. Locations 2, 3, and 4 are equally loaded, but as an artefact of the Java Voyager implementation, Location 1, as the initiating location is more 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* (AMS) 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 [9]. 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 Java. In Section 5, we compare

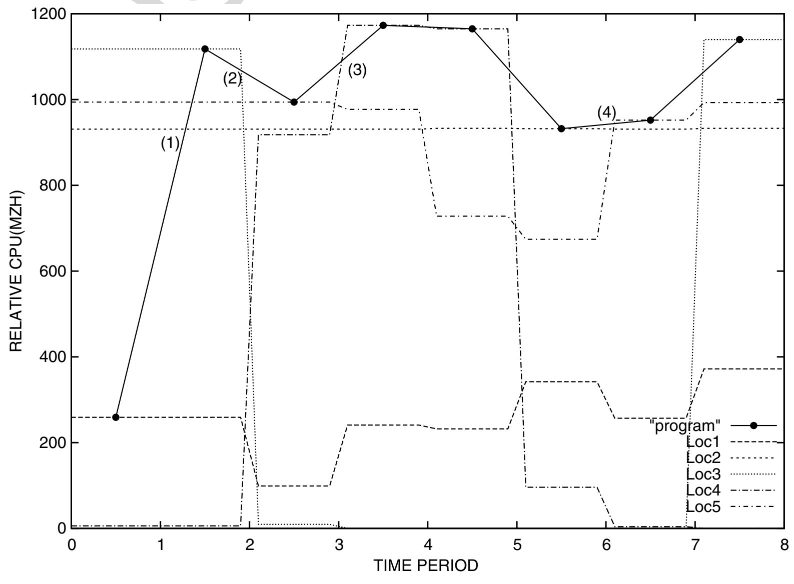

Fig. 2. Auto-mobile matrix movement.

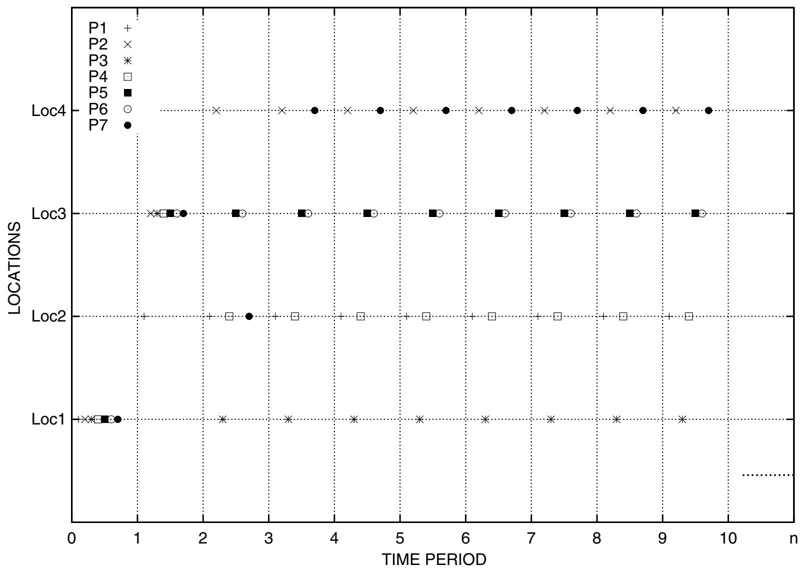

Fig. 3. AMP load management, 7 AMPs on 4 locations.

the Jocaml and Java Voyager execution times. We define and sketch the implementation of `AutoIterator` in JavaGo [10] in Section 6. Finally, Section 7 summarises our results and considers future research.

2. Background

2.1. Mobile computation

Network technology is pervasive and software is increasingly 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., program can migrate between locations. A typical mobile program is a data mining application 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 [10].

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. Concrete skeletons are executable, and the user must link computation-specific code into the appropriate skeleton. Fig. 4 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 autonomous 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.

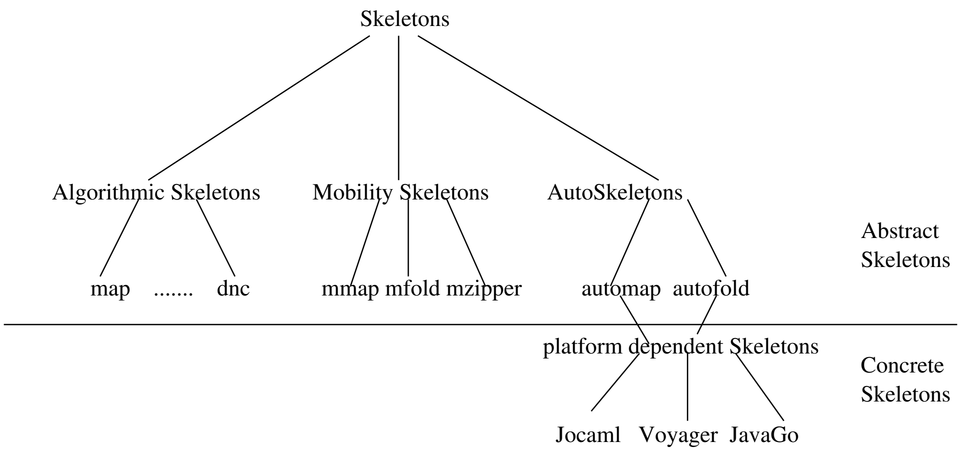

Fig. 4. Skeleton taxonomy.

In Fig. 4 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.

The motivation for auto-mobile skeletons is to minimise program execution 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.

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]`. The automap skeleton, `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, Fig. 5 shows how 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.

```
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. 5. Jocaml `automap` matrix multiplication.

3.2. `automap` Design and implementation

Potentially `automap` could investigate moving after processing every element of the list, but this induces enormous coordination overheads. Such 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}):

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

`automap` (Fig. 8) investigates moving after processing `gran` elements. 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 Fig. 6. In this function, `work` is the length of the list (list `l` in Fig. 8) of tasks, `h` represents the first task of the list, and `f` is the mapped function. So the type of `getGran` is: `int` $\rightarrow$ `(a` $\rightarrow$ `b)` $\rightarrow$ `a` $\rightarrow$ `(b*float*int)`.

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 $*$ (100 − 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., $T_h > T_{\text{comm}} + T_n$. 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 Fig. 7. Note that the sixth to last line encodes Eq. (1).

```
let getGran work f h =
    let (fh,fhtime) = timedapply f h
    in let t_static = fhtime * (float (work))
       let t_coord = tcoord (numofhost)
       let ov = 0.05                  (* 5% coordination overhead *)
    in let times = (ov * t_static)/t_coord
    in let gran = if times > 0
                  then (work/times)
                  else work
    in (fh,fhtime,gran)
```

Fig. 6. `getGran`: calculating checkmove granularity.

```
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 = cur.relspeed / host_next.relspeed * t_h
  in
      if (t_h > (t_n + t_comm))
      then (
            go host_next
            host_next
           )
      else cur
```

Fig. 7. `check_move`: deciding to move.

The definition of `automap` is given in Fig. 8. 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 Fig. 9. 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 automatically decide whether and where

```
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. 8. Jocaml `automap`.

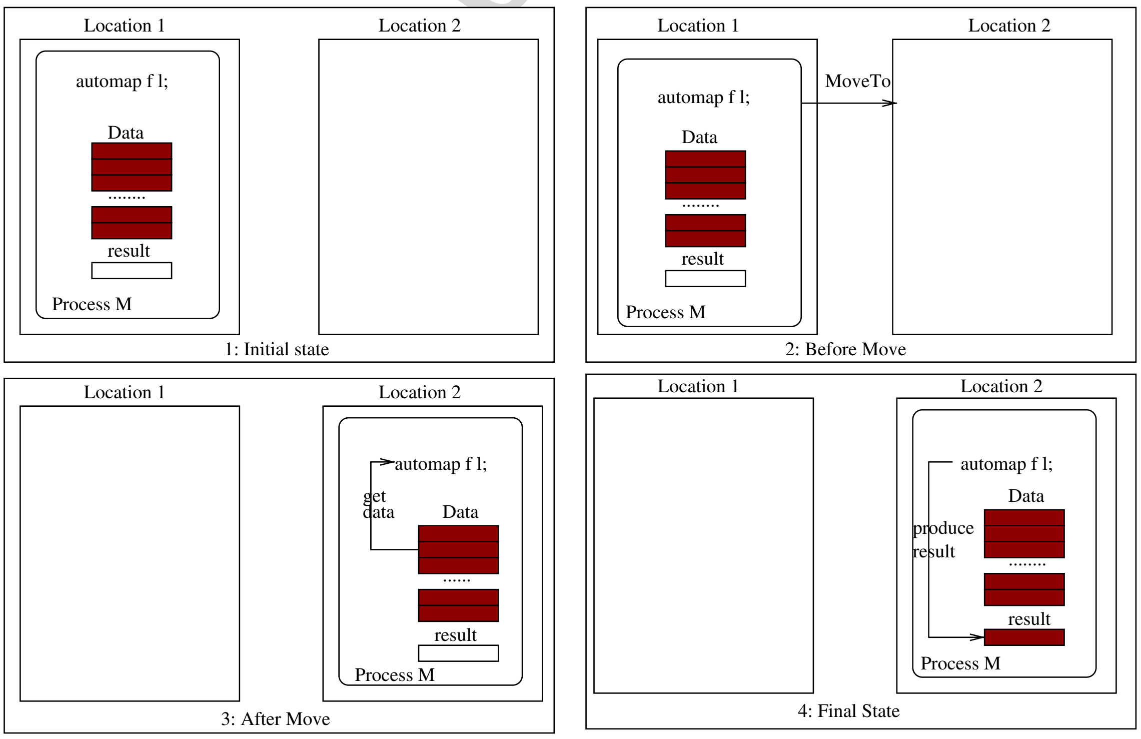

Fig. 9. Coordination behaviour of Jocaml `automap`.

the program moves 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).

3.3. Jocaml automap performance

Fig. 10 compares the execution times of a static, and automap-based matrix multiplication programs on matrixes of varying sizes. The execution environment has three locations with CPU speeds 534 MHz, 933 MHz 134 and 1894 MHz. The loads on these three computers are almost zero. We started both the static and the mobile 135 programs on the slowest CPU. Up to a certain size (here a 330∗330 element matrix) the static and AMP programs both execute at the same location and have almost identical performance. This is because a move would consume more than the specified overhead (0%, here 5%) of the execution time (T_{static}). Beyond this size, the program checks and makes a move to the fastest location, thereby reducing execution time.

3.4. Jocaml autofold

The standard fold in Jocaml, fold f a [bl; ...; bn], computes f (... (f (f a bl) b2) ...) bn. The autofold skeleton, autofold cur f a [bl;...;bn] computes the same value but may migrate to a faster location. The definition of autofold is given in Fig. 11.

Autofold has been used to construct a coin counting program that uses a genetic algorithm to find a minimal and maximal set of coins that sum to a target figure [7]. Fig. 12 shows the execution times of static and autofold-based versions of the coin counting program. As before, once the program has a sufficiently large execution time, it benefits from moving to a faster location. In this figure, there are three clear irregularities in the mobile version plot. That is because as the size of the program increases, gran (see Fig. 6) may be decrease. So at some points, even if the size of the programs is increased, it may move early to the faster location than the smaller program, so the bigger program finishing faster than the smaller program. For example, the program with size 50 matrix does not move, but the one with size 60 matrix moves. Similarly, the gran of size 100 is 51, but the gran of size 110 is 37, so the size 110 program move to faster location earlier than size 100 program. So we can see an irregularities at point 110 in the plot. These irregularities also arise in Figs. 10,16 and 17, but are too small to be noticed.

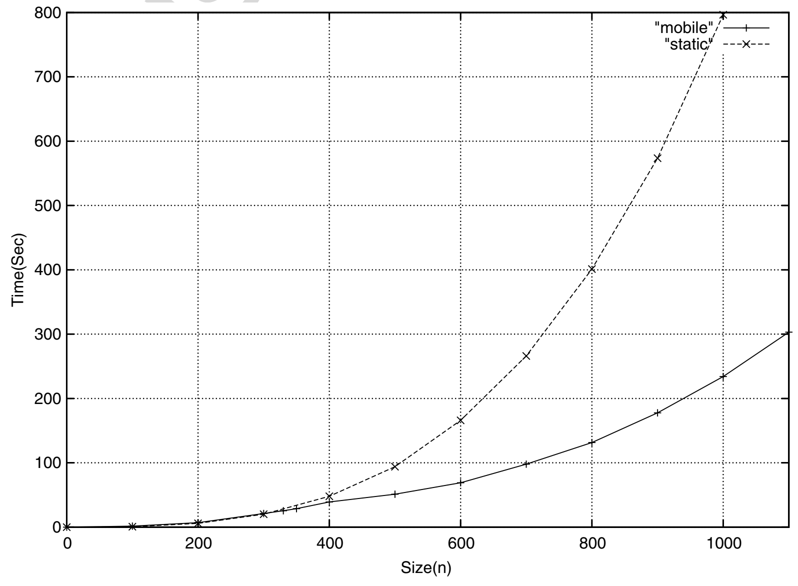

Fig. 10. Jocaml matrix multiplication execution times.

```
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. 11. Jocaml `autofold` definition.

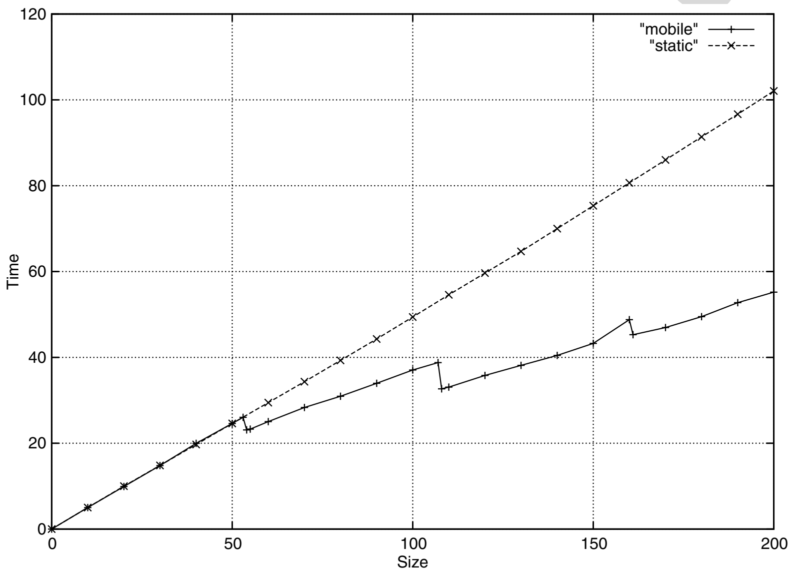

Fig. 12. Jocaml coin counting execution times.

4. Java autonomous mobility skeletons

It is appealing to implement Java autonomous mobility skeletons as Java is a very widely used language and there are numerous mobile Java variants. Voyager [11,8] is a popular Java with weak mobility, providing a wide range of services and features for distributed application development. The Voyager ORB includes a 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`. Fig. 13 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 Java 1.4 has no parametric polymorphism (i.e. Generics in Java 1.5) the Voyager `automap` operates on a list of `Object` and returns a list of `Object`.

As Voyager supports only weak mobility, when the program moves it communicates only the code, and not the execution state. Fig. 14 shows the coordination behaviour of the Voyager `automap`. Here, we started a Voyager program with `automap`, which applies `f` in `Object A` to list `l` in location 1. The program sends the code of `Object A` to location 2 (1). The system constructs a reference from location 2 to the data in loca-

```
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. 13. Java Voyager `automap`.

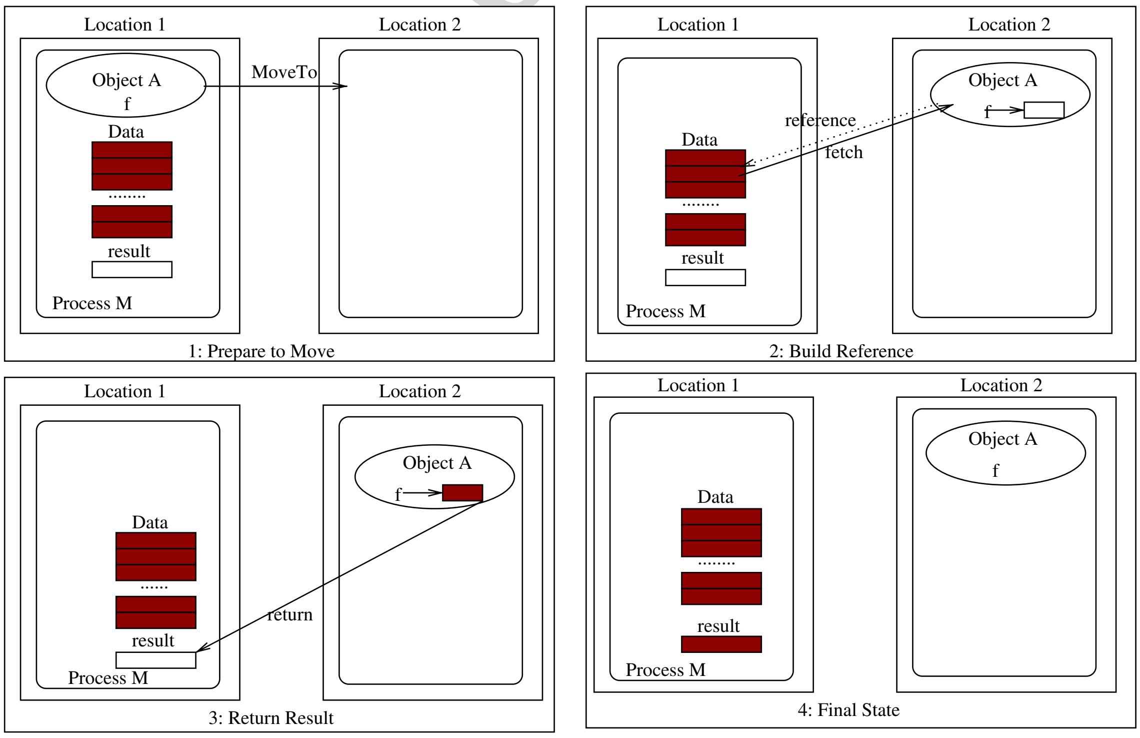

Fig. 14. Coordination behaviour of Java Voyager `automap`.

tion 1 (2). In location 2, function f fetches data from location 1, produces a result, and returns it to location 1 (3). After the program has 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 Fig. 15. 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. The method invocation auton.automap (rowM, mat1), applies rowM.mapf on array mat1, and periodically decides when and where to move.

Fig. 16 shows the execution times of static and automap-based versions of Voyager matrix multiplications, using the apparatus from Section 3.3.

4.3. Java voyager autofold

An autofold is also readily constructed in Voyager Java. Fig. 17 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 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. 15. Java Voyager autonomously mobile matrix multiplication.

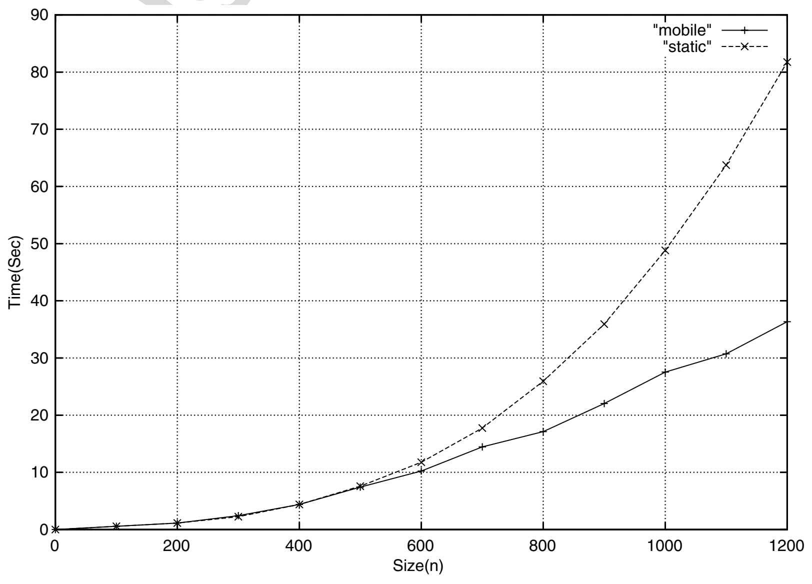

Fig. 16. Java Voyager matrix multiplication execution times.

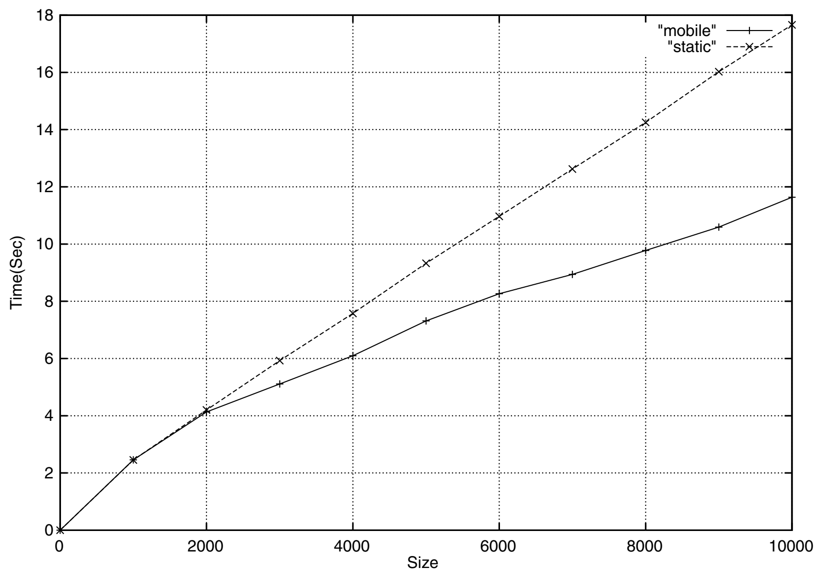

Fig. 17. Java Voyager coin counting execution times.

4.4. Multiple AMSs behaviour

We have measured the behaviour of multiple AMS programs on a heterogeneous network of ten locations. The CPU speeds are 3139MHz (Loc1–Loc5), 2167MHZ (Loc6), 1793 MHz (Loc7–Loc10). For illustration, the movement of 20 AMSs between the 10 locations is shown in Fig. 18. In Fig. 18 "B" denotes a *balanced state*, where every AMS resides at a location with a similar per-AMS relative speed, i.e. (*CPU speed*) * *load*%/(*Number of AMS*), if AMSs are the only processes on the network. In this state, the AMSs remain in the current locations and until the loads in the network change. In Fig. 18 we started 20 AMSs on Loc1 in time period "0". In consequence Loc1 was very busy and the 20 AMSs were looking for other locations which were less busy. After some movements of each AMSs, we got a balanced statue in time period "*k*". The AMSs keep the balanced statue and do not move any more until time period "*k*+*x*", when one of the

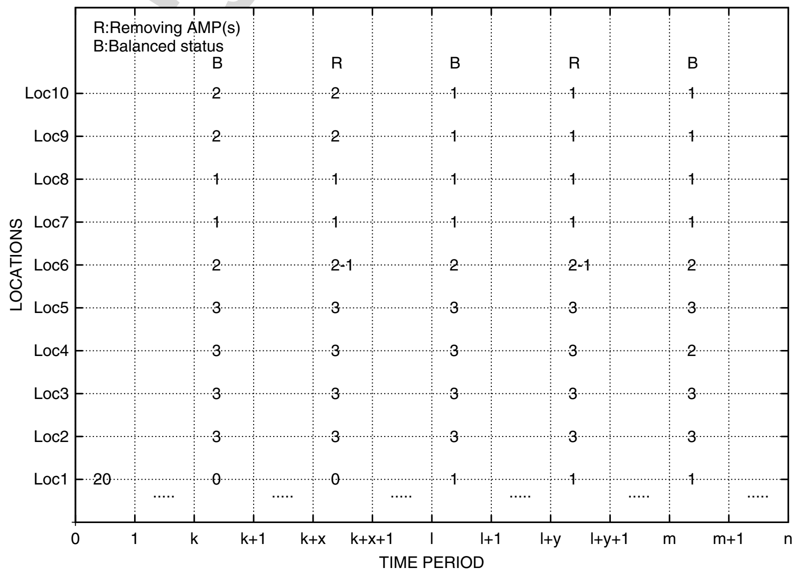

Fig. 18. 20 AMSs on heterogeneous network (10 Locations).

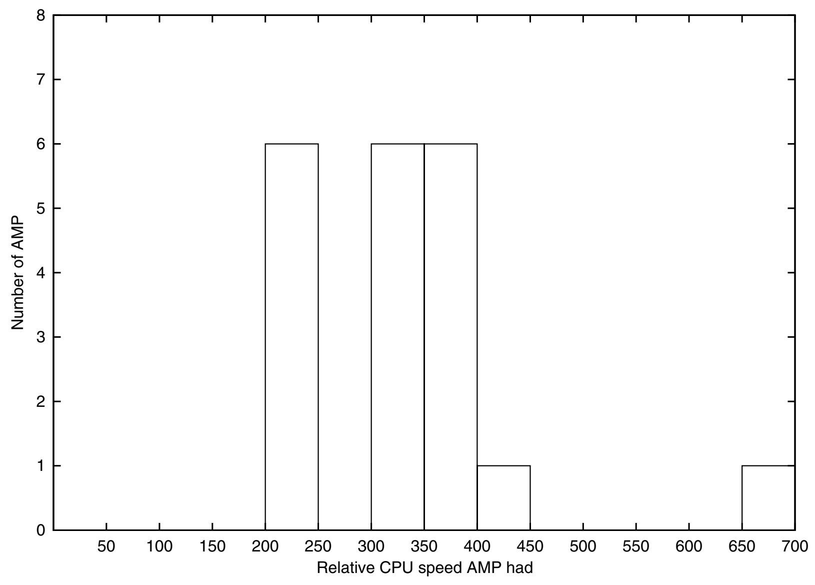

Fig. 19. Relative CPU speed for AMPs.

AMSs is finished on Loc6 and the balance is broken. So the other 19 AMSs move again and reach a new balance in time period "*l*".

Fig. 19 shows the relative CPU speed available to 20 AMSs. In the figure, most AMSs have relative CPU speed from 200 MHz to 400 MHz (18 out of 20 AMPs). There is one AMS on Loc1 with CPU speed 650–700 MHz. Similar results were obtained with 19 and 18 AMSs.

5. Jocaml and Java voyager comparison

From Figs. 10 and 16, we can see there is a huge difference in the runtimes obtained with Jocaml and with Java Voyager. Table 1 compares the runtimes of static versions of Jocaml and Java Voyager matrix multiplication programs. The time complexity of our matrix multiplication is $O(n^3)$, so in the table's third and sixth columns we use "time/size^3" as a measure of the time taken for a single matrix element multiplication. From this table, Jocaml matrix multiplications take on average 16.4 times longer time than Java Voyager.

Similar differences also can be seen in Figs. 12 and 17. For the coin counting program on average Jocaml is 272 times slower than Java Voyager and Table 2 summarises the results.

Table 1
Jocaml and Java Voyager matrix multiplication runtimes comparison

Jocaml			Voyager			Jocaml/Voyager
Size	Time	Time/size^3	Size	Time	Time/size^3	Average
300	20.1	7.4e^{-7}	300	1.3	4.7e^{-8}	–
400	47.9	7.5e^{-7}	400	3.0	4.6e^{-8}	–
500	93.9	7.5e^{-7}	500	6.0	4.8e^{-8}	–
600	166.1	7.7e^{-7}	600	10.2	4.7e^{-8}	–
700	266.2	7.7e^{-7}	700	16.0	4.7e^{-8}	–
800	401.3	7.8e^{-7}	800	23.8	4.6e^{-8}	–
900	573.6	7.9e^{-7}	900	33.4	4.6e^{-8}	–
1000	796.5	7.9e^{-7}	1000	46.2	4.6e^{-8}	–
Average		7.7e^{-7}	Average		4.7e^{-8}	16.4

Table 2
Jocaml and Java Voyager coin counting runtimes comparison

Jocaml			Voyager			Jocaml/Voyager
Size	Time	Time/size	Size	Time	Time/size	Average
30	14.8	0.49	3000	5.9	0.0019	–
40	19.7	0.49	4000	7.6	0.0019	–
50	24.6	0.49	5000	9.3	0.0019	–
60	29.5	0.49	6000	11.0	0.0018	–
70	34.3	0.49	7000	12.6	0.0018	–
80	39.3	0.49	8000	14.3	0.0018	–
90	44.3	0.49	9000	16.0	0.0018	–
100	49.4	0.49	10000	18.0	0.0018	–
Average		0.49	Average		0.0018	272

6. 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 `Iterator` interface are `hasNext`, `next` and `remove` [9]. The `AutoIterator` class implements all three methods, and extends it with `autonext`, which has the same functionality as `next` but can make autonomous mobility decisions.

Automap and autofold can use weak mobility as the computation moved is encapsulated in the function mapped or folded. In `AutoIterator`, however, there is no encapsulated computation to be weakly moved and strong mobility is required to move the entire collection. Hence Voyager, with only weak mobility, cannot be used. JavaGo [10] supports strong mobility and Fig. 20 shows an `autoNext` implementation again using

```
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");
}
```

Fig. 20. JavaGo `autoNext` method in `AutoIterator` class.

```
public class AutoIterator implements Iterator,Resalable{
  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");
  }

  public migratory Object autoNext() { ... }
}
```

Fig. 21. JavaGo `AutoIterator`.

analogous `check_move` and `getGran` functions. The whole program of `AutoIterator` is given in Fig. 21. The `AutoIterator` is very similar to `automap` and `autofold`. It counts the time of computation on the first element of the list, and calculates a checking granularity, `gran`. Then it makes autonomous decisions whether and where to move after every `gran` elements.

Fig. 22 shows how `AutoIterator` can be used to implement a sequence of matrix multiplications. 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.

Fig. 23 shows the execution times of static and AutoIterator-based versions of a JavaGo matrix multiplication program. Once again, the skeleton version is faster.

```
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. 22. JavaGo autonomously mobile matrix multiplication.

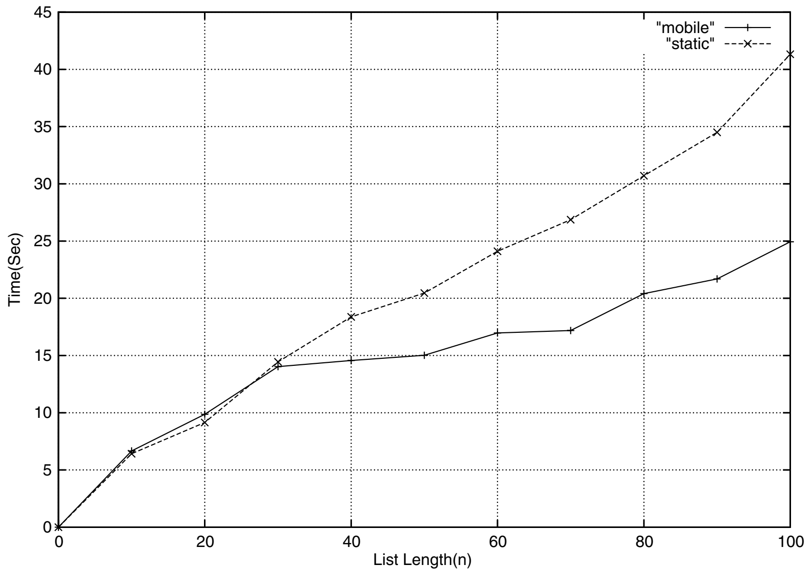

Fig. 23. `AutoIterator` matrix multiplication execution times.

7. Conclusion

We have proposed auto-mobile skeletons that encapsulate common patterns of self-aware mobile coordination aiming to 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`. The realisations are provided both in the functional language context shared with other skeleton species, using Jocaml, and in an object-oriented context using 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.

Auto-mobile skeleton cost models are dynamic and substantially implicit. During the traversal of a collection, the skeleton implementation periodically measures the time to compute a single collection element, and uses the value to parameterise an implicit cost for the remainder of the traversal.

Auto-mobile skeletons currently have a number of limitations because the skeletons dynamically parameterise the cost model with measurements of performance on the preceding collection segment. If the program is reasonably regular, i.e., computing each segment of the collection represents a similar amount of work, then the cost model will be valid, and hence the movement decisions reasonable. However, as the computations become increasingly irregular, the cost model will be less valid, and hence the movement decisions may not optimise performance.

Currently the cost models of auto-mobile skeletons do not incorporate the costs of computations following the processing of the current collection. This restricts auto-mobility skeletons to programs that expose useful loci of mobility at the top-levels that dominate the computation. In essence we lack appropriate techniques to compose and nest auto-mobile skeletons, as we are unable 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 automatically extract, 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 execution time.

We are experimenting with a simple cost analyser for a small core language, where cost functions are generated in SML rather than in the source language. We plan to investigate the use of meta-programming techniques to integrate cost functions into the source language at appropriate checking points.

References

[1] A.R.D. Bois, P. Trinder, H. Loidl, Towards mobility skeletons, Parallel Processing Letters 15 (3) (2005) 273–288.
[2] M. Cole, Algorithmic Skeletons: Structured Management of Parallel Computation, MIT Press, 1989.
[3] X.Y. Deng, G. Michaelson, P. Trinder, Towards high level autonomous mobility, in: H.-W. Loidl, (Ed.), Draft Proceedings of Trends in Functional Programming, Munic, Germany, November 2004.
[4] X.Y. Deng, P. Trinder, G. Michaelson, Autonomous Mobile Programs, 3rd International Conference on Mobility Technology, Applications and System, Bangkok, Thailand, October 2006.
[5] C. Fournet, F.L. Fessant, L. Maranget, A. Schmitt, Jocaml: a language for concurrent distributed and mobile programming, in: Proceedings of the Fourth Summer School on Advanced Functional Programming, St Anne's College, Springer-Verlag, Oxford, 2002, pp. 19–24.
[6] A. Fuggetta, G.P. Picco, G. Vigna, Understanding code mobility, IEEE Transactions on Software Engineering 24 (5) (1998) 342–361.
[7] J. Hawkins, A. Abdallah, A generic functional genetic algorithm, in: P. Trinder, G. Michaelson (Eds.), Draft Proceedings of the First Scottish Functional Programming Workshop, Heriot-Watt University, Edinburgh, 1999, pp. 151–168.
[8] Resursion Software. Voyager ORB Developer's Guide, May 2005. <http://www.recursionsw.com/Voyager/Voyager_User_Guide.pdf>.
[9] S. Sahni, Data Structures, Algorithms, and Applications in Java, Mc Graw Hill, University of Florida, 2000.
[10] T. Sekiguchi, JavaGo Home Page. <http://homepage.mac.com/t.sekiguchi/javago/index.html> (accessed October 2005.).
[11] T. Wheeler, Voyager Architecture Best Practices. Recursion Software, March 2005. <http://www.recursionsw.com/Voyager/2005-03-31-Voyager_Architecture_Best_Practices.pdf>.
[12] J.E. White, Mobile agents, in: J. Bradshaw (Ed.), Software Agents, AAAI/MIT Press, Menlo Park, CA, 1997, pp. 437–472.