MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 1614 October 6, 1997

Paradigms for Structure in an Amorphous Computer

Daniel Coore, Radhika Nagpal, Ron Weiss

Abstract

Recent developments in microfabrication and nanotechnology will enable the inexpensive manufacturing of massive numbers of tiny computing elements with sensors and actuators. New programming paradigms are required for obtaining organized and coherent behavior from the cooperation of large numbers of unreliable processing elements that are interconnected in unknown, irregular, and possibly time-varying ways. *Amorphous computing* is the study of developing and programming such ultrascale computing environments. This paper presents an approach to programming an amorphous computer by spontaneously organizing an unstructured collection of processing elements into cooperative groups and hierarchies.

This paper introduces a structure called an *AC Hierarchy*, which logically organizes processors into groups at different levels of granularity. The AC hierarchy simplifies programming of an amorphous computer through new language abstractions, facilitates the design of efficient and robust algorithms, and simplifies the analysis of their performance. Several example applications are presented that greatly benefit from the AC hierarchy. This paper introduces three algorithms for constructing multiple levels of the hierarchy from an unstructured collection of processors.

This publication can be retrieved by anonymous ftp at URL ftp://publications.ai.mit.edu/ai-publications/

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for this research was provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-96-1-1228. Radhika Nagpal was also supported by the AT&T GRPW Fellowship.

Contents

1 Introduction

Amorphous computing is the study of developing and programming ultrascale computing environments [1]. Recent developments in microfabrication and nanotechnology will enable the inexpensive manufacturing of vast numbers of tiny computing elements with integrated sensors and microactuators. These sensor-rich processing elements will be distributed and embedded in structures to create intelligent and responsive environments, such as bridges with load sensing capabilities or smart surfaces that monitor the weather. In such ultrascale environments, the elements are unreliable, interconnected in unknown, irregular, and possibly time-varying ways, and are constrained to interact locally. New programming paradigms are required for obtaining organized, fault-tolerant, and coherent behavior in such environments.

Biological systems indicate that hierarchies are a useful method for controlling vast numbers of processing elements, for example cells. Cells specialize for different functions that together perform a unified task within a tissue. Tissues themselves specialize and cooperate within an organ. Finally, organs collaborate to form complex systems, such as the digestive system. At each level the group can be viewed as single entity accomplishing a particular task.

This paper presents an approach to programming an *amorphous computer* by spontaneously organizing the unstructured collection of processing elements into a hierarchy of cooperative groups, called an AC hierarchy. The *AC Hierarchy* logically organizes the processors into groups at different levels of granularity. Each group can operate as a single entity where the group members collaborate on specific tasks. The AC hierarchy provides bounds on the communication latency within a group, for efficient member collaboration, and bounds on communication between groups at the same level. It also provides physical bounds on the size of a group and the proximity of logically close groups. These properties make the AC hierarchy a suitable programming abstraction for implementing a variety of applications, like naming and routing, factoring and mergesort, and distributed sensory control.

As demonstrated by these applications, the AC hierarchy provides a useful programming abstraction for aggregated computation and communication. It simplifies programming by providing high-level abstractions for partitioning a problem into tasks and multiple levels of tasks, while hiding the details of how a group accomplishes the task. The hierarchies can be used to design efficient resource allocation and to specialize regions within the amorphous computer for different computational or sensory tasks. Elements can be aggregated to increase computational power or increase robustness, which is particularly important in such an unreliable environment. The AC hierarchy bounds provide important timing and locality guarantees which simplify the design and analysis of algorithms using the hierarchy.

This paper also presents three algorithms for constructing AC hierarchies. The algorithms are suited to the amorphous computing environment and are self-assembling, scalable, fault-tolerant, have low message overhead and rely only on local interactions. The first two algorithms, *overlapping-clubs* and *tight-clubs*, construct the first level of the hierarchy on the unstructured processors and take advantage of the local broadcast. In addition to AC hierarchy properties, overlapping clubs provide geometric bounds and tight clubs provide groups with high fault tolerance. The third algorithm, *tree-regions*, can be applied recursively to create higher levels of the hierarchy. It uses spanning trees to generate maximal groups and uses the internal tree structures for group coordination.

The remainder of this paper proceeds as follows: Section 2 describes the important characteristics of an amorphous computer. Section 3 introduces AC hierarchies, provides three example applications and discusses the benefits of using an AC hierarchy paradigm for programming an amorphous computer. Section 4 describes the implementation of an AC hierarchy using the three algorithms mentioned above and presents an analysis of their properties. Section 5 presents related work. Finally, section 6 offers conclusions and points to future work in developing new construction algorithms and a language interface for the hierarchies.

2 Amorphous Computing Properties

The unique features of an amorphous computer, like the potential for embedded applications, inexpensive fabrication, fault-tolerance and scalability, arise from the following underlying assumptions.

- *Individual processors are identical and mass produced.* This allows us to manufacture vast quantities of processors cost effectively. Each processor has a random number generator to distinguish itself from others.
- *Processors possess no a-priori knowledge of their location, orientation, or neighbors' identities.* Given the sheer number of elements, providing processors such information is extremely expensive. Individual processors must discover their layout and position information.
- *Processors operate asynchronously although they have similar clock speeds.* An amorphous computer does not assume that the processors are synchronized because it may be difficult or inefficient to guarantee synchronization in certain physical environments.
- *Processors are distributed densely and randomly.* The random distribution is uniform.
- *Processors are unreliable.* In order to manufacture large quantities cheaply we cannot afford to build and test individual processors. Furthermore the sheer number of processors inevitably results in failures. Fault tolerance is achieved via redundancy rather than relying on hardware perfection.
- *Processors communicate only locally and do not a have precision interconnect.* Given the vast numbers of processors, it is not cost effective to connect processors using wires. Rather the processors communicate with physically nearby neighbors through a local broadcast mechanism, although the specific mechanism is dependent on the substrate. The communication radius is assumed to be much smaller than the total area occupied by the amorphous computer.

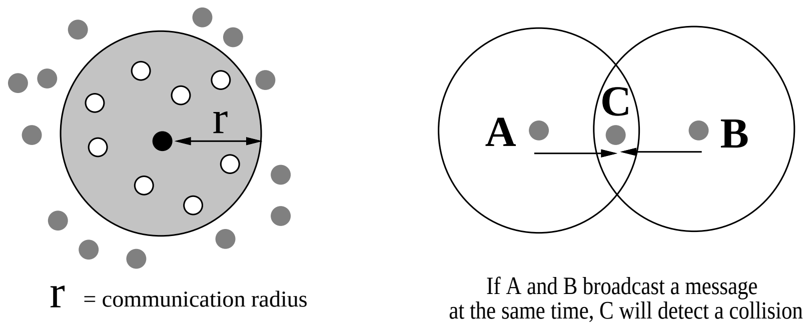

Figure 1: Communications Model

This paper assumes the following additional constraints regarding the physical characteristics of an amorphous computer.

1. *The amorphous computer exists on a planar* 2D *surface.* This assumption is made to simplify the analysis. The algorithms presented also work in a 3D space.
2. *The communications model assumes that all processors have a circular broadcast of approximately the same fixed radius and share a single channel.* As a result, collisions may occur when two processors with overlapping broadcast areas send messages simultaneously. The receiver can detect a collision, although the sender can not detect a collision. This is illustrated in figure 1.

3 Hierarchies as an Organizational Principle

This paper presents an approach to programming an amorphous computer by organizing the unstructured collection of processing elements into an AC hierarchy. This section presents the definition of an AC hierarchy and discusses several logical and physical properties that can be derived from the definition. Three motivating applications are discussed - naming and routing, mergesort and distributed vibration control. As demonstrated by these examples, the hierarchical abstraction simplifies programming an amorphous computer and facilitates the analysis of amorphous computing algorithms.

3.1 Amorphous Computing Hierarchy

An *AC Hierarchy* is an ordered sequence of *levels*. A level is a collection of *groups* and an associated means of inter-group communication. In a graph representation, G_n, of a level n, a group is a node and an edge represents communications between adjacent groups. A group at level n is a connected component of the level $n-1$ graph, G_{n-1}, with a specified constant diameter bound, D_n. Groups at level n are considered adjacent if any of their members were adjacent in G_{n-1}. Formally, let u_n denote a group at level n, then

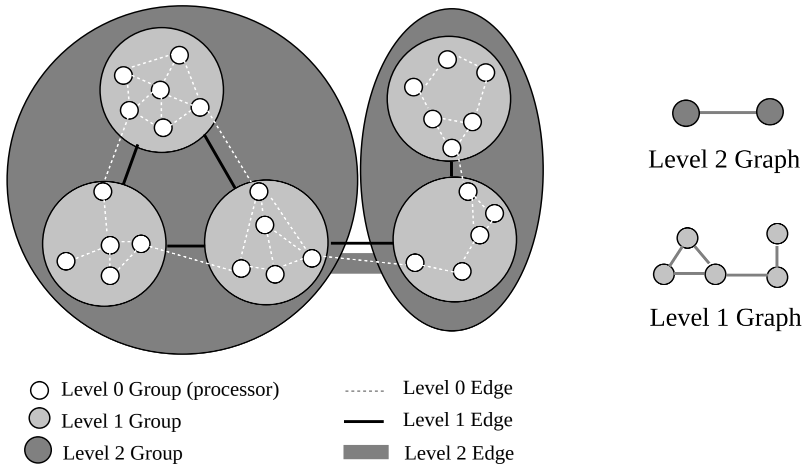

Figure 2: An AC Hierarchy

Every u_0 is a processor.

A level 0 edge (u_0,v_0) exists *iff* processors u_0 and v_0 are within each other's local broadcast region.

Group u_n is a connected set of level $n-1$ groups with a diameter bounded by D_n level $n-1$ edges, all internal to u_n. Groups need not be disjoint.

A level n edge (u_n, v_n) exists *iff* there exists an edge (x_{n-1}, y_{n-1}) such that $x_{n-1} \in u_n$ and $y_n \in v_n$. Edge (u_n, v_n) must be constructed by using only level $n-1$ nodes that are members of u_n or v_n.

A level n is said to have *coverage* if every level $n-1$ group is a member of at least one level n group. If level n has coverage and G_{n-1} is a connected, then G_n will also be connected. This connectivity property is useful for many applications, such as distributing global information. An AC hierarchy where every level has coverage is said to be *complete*. Figure 2 illustrates an example of an AC hierarchy.

An important property of an AC hierarchy is the *Bounded Edge Property* which states that the number of processor hops required to traverse an edge at any level has an upper bound. This follows directly from the diameter bound on groups and the constraint on edges to use only nodes that are members of the groups. At level 0 the maximum number of processor hops on an edge is clearly 1. The bound at level n is denoted by B_n, where $B_n = \prod_{i=0}^{n}(2D_i + 1)$ (see proof in Box 3).

The physical nature of the amorphous computer combines with the Bounded Edge property to produce some interesting results:

- **The physical distance associated with any level n edge is bounded by a constant P_n:** The physical distance associated with an edge between two groups is the maximum distance from any processor member of one group to any processor belonging to the other group. A processor's broadcast range is fixed, so $P_n = B_n \cdot r$.

Bounded Edge Property

Let $E(G_n)$ be the set of all level $n-1$ edges in the graph G_n.

Let $|(u_n, v_n)|$ represent the maximum number of processor hops that may be required to traverse the edge (u_n, v_n). More precisely, $|(u_n, v_n)|$ denotes the maximum over the lengths of the shortest paths between all pairs of processors in u_n and v_n, measured in G_0 restricted to the processors in $u_n \cup v_n$. Clearly $|(u_0, v_0)|$ is 1.

For any level n, e_n denotes the longest edge (in terms of processor hops) between groups at that level.

$$e_n = \max_{e \in E(G_n)} |e|. \tag{1}$$

Let $\delta(u_n)$ denote the diameter of a group u_n measured in processor level hops. In other words $\delta(u_n)$ is the maximum number of processor level hops required for any two processors within u_n to communicate.

Then, for an arbitrary edge (u_n, v_n) in G_n,

$$\begin{aligned}
|(u_n, v_n)| &\leq \delta(u_n) + \delta(v_n) + e_{n-1} \\
&\leq D_n e_{n-1} + D_n e_{n-1} + e_{n-1} \\
&= (2D_n + 1)e_{n-1}
\end{aligned}$$

So,

$$\begin{aligned}
e_n &\leq (2D_n + 1)e_{n-1} \\
&= (2D_n + 1)(2D_{n-1} + 1)\ldots(2D_1 + 1)e_0 \\
&= (2D_n + 1)(2D_{n-1} + 1)\ldots(2D_1 + 1)
\end{aligned}$$

since at level 0 the longest edge is 1 processor hop.

Therefore, an edge at level n is at most B_n processor hops long, where

$$B_n = \prod_{i=0}^{n}(2D_i + 1)$$

Figure 3: Bounded Edge Property

- **The area occupied by any level n group is bounded by a constant A_n:** As a result of the bounded diameter D_n of a group and the bound P_n on edge distances, the group cannot cover an area larger than a circle with a radius $D_n \cdot P_{n-1}$.
- **The number of processors in any level n group is bounded:** Since processors occupy space and are assumed not to overlap on a plane, this property follows immediately from the previous one.

3.2 Example Applications

Ultimately, an AC hierarchy is really an abstraction of an amorphous computer that simplifies high level programming. It provides logical relationships between the processors that facilitate implementing algorithms already designed for other architectures. The physical properties emphasize the connection between the geometry of the distribution of the processors and programming abstractions induced by the AC hierarchy.

These properties make it possible to easily and efficiently implement several applications on an Amorphous Computer. In addition, hiererachies form a natural way of decomposing many problems. This section presents three motivating examples of applications using an AC hierarchy.

Naming and Routing:

The hierarchical organization provides a natural addressing scheme for both groups and processors. This approach is analogous to the *post office naming paradigm*, where the region in which the entities exist is partitioned into a hierarchy of localities - homes, towns, states, countries. These entities are addressed by a list of containing region names in the post office hierarchy. Similarly, a globally unique identifier for a group can be constructed from the sequence of its nesting groups' identifiers starting from the top level. This organization can be used to generate both a global naming scheme as well as a relative one. In this context, edges reflect physical distances between the named entities. When comparing two names, the length of the matching prefixes between them indicates the physical distance between the processors they represent.

Certain routing schemes, such as area routing [11], use organizations with properties similar to the AC Hierarchy properties. Area routing uses a hierarchy of areas, where the name of a processor is the sequence $\{A_n, \ldots, A_1\}$ of areas at different granularities to which it belongs. The groups in the hierarchy correspond to these areas. Edges between groups are similar to edges between areas. Routing between two processors consists of two phases. In the first phase of routing, a group routes messages to a designated sibling, and this sibling delivers the message at the next higher level. This continues until the message reaches a group common to both the source and destination. In the second phase, the message is routed "down" the hierarchy to the destination in a manner analogous to the first phase. The benefit provided by the AC hierarchy is the simplicity of the implementation of area routing on an amorphous computer. The fixed communication bounds at each level guarantee that the routing tables that are stored at each group are not too large.

Divide and Conquer:

For a divide and conquer algorithm such as *mergesort*, the hiererachy provides a simple and efficient framework for distributing the problem (i.e. allocating processing resources) and combining answers. Groups in the hiererachy can be treated as computational units, so that mergesort can be invoked on a group. A group can partition the sorting task amongst its members and merged the results. Below is pseudo-code for *mergesort* expressed in terms of a method on a group. In this implementation, the array of integers to be sorted is stored in the group's state.

```
// inherited state:
//   Members = list of members

group MergeSortGroup {
  // state
  IntegerArray my_ints

  // methods
  void sort () {
    if (expected_compute_time(my_ints) < time_to_distribute_problem_to_members)
       quicksort()
    else
       for all members in this.Members, in parallel
       {
         member.set_my_ints(sublist)
         member.sort()
       }
       merge_member_lists()      // set my_ints to the merge of members my_ints
  }
}
```

The programmer can use the abstraction without worrying about the intimate details of implementation of coordinated group behavior. A language abstraction built on top of the hiererachy enables one to readily define and invoke operations on groups of processors. The implementation of coherent group behavior can be accomplished via a group leader that coordinates and distributes the subtasks to member groups. Through this leader, the group can maintain state. The bounds on communications and groups sizes allow efficient implementations of coordinated group behavior.

The number of subgroups and the resources of each one determines how the group partitions its task. In order to program an efficient algorithm, the programmer needs an estimate of the cost of operations and intra-group communications. In this example, a group compares the cost of distributing the subtasks based on the cost of the edges to the cost of computing it in-place in order to determine whether additional recursion is required. A more sophisticated sorting routine may vary the branching factor and the sublist sizes based on detailed knowledge of its members.

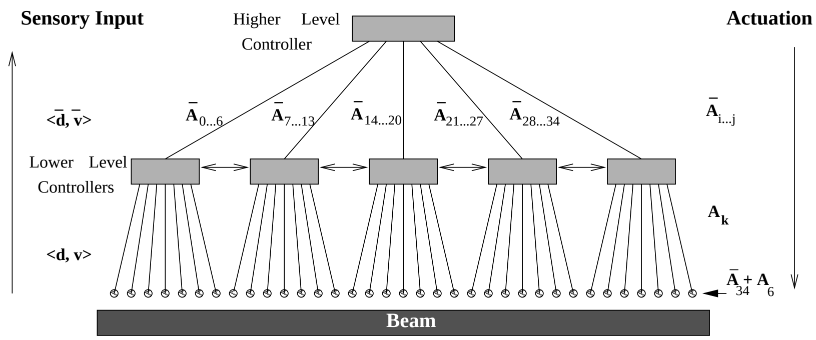

Figure 4: Information Flow for Controlling a Beam

Factorization and primality testing are two examples of divide and conquer algorithms that can be mapped to the hierarchy in a similar way. The cost of building the hierarchy can be amortized over several executions of such algorithms. In general, divide and conquer algorithms with balanced trees can be implemented efficiently in a straightforward manner.

Sensory Input Collection and Distributed Control:

Distributed control of physical structures is a prime motivation for the development of an amorphous computer. When equipped with sensors and actuators, the myriad amorphous computing elements have an immense potential for distributed sensory control. Processors accumulate sensory input and use actuators to control the property of interest. The distributed control problem, where control decisions are based on only a subset of all the sensory information, is not well understood.

Hall, Crawley, How and Ward show in [6, 8] that an effective approach to controlling a stiff material (e.g. maintaining a particular shape) in a distributed manner is to build a hierarchical controller. They describe a two level hierarchical control system. The lower level controllers control localised regions of the material and report summarised "sensor readings" to the higher level (there are also variants where the lower level controllers will collaborate with immediate neighbours before reporting a summarised reading to the higher level). The higher level controller in turn uses a set of interpolation functions to estimate the shape of the entire beam from the summarised readings, and then makes actuation decisions based solely on that shape estimate. The key insight in their approach is that higher spatial (ie. modal) frequencies in stiff materials have localised extent, and so can be controlled effectively using only information from localised sensors. Low spatial frequencies have much larger (non-localised) extent and so controlling them effectively requires sensor information from the entire surface. Roughly speaking, the lower level controllers are responsible for controlling the higher frequency disturbances while the upper level controller responds to the lower frequencies.

Figure 4 illustrates the flow of sensory input up the hierarchy and actuation commands down the hierarchy for controlling a 1-dimensional beam. The sensors report their displacement and velocity $\langle d, v\rangle$ to the lower level of the hierarchy. Each lower level node computes a new value $\langle \bar{d}, \bar{v}\rangle$, based on its sensors' readings, the interpolation functions, the beam properties (such as mass, stiffness) and the control law. The lower level controllers then send the summarised readings, $\langle \bar{d}, \bar{v} \rangle$, to the higher level of the hierarchy. The higher level, in turn, computes a global actuation vector, $\bar{A}$, and sends to each lower level node, its corresponding subvector $\bar{A}_{i \ldots j}$. Each lower level controller computes a separate response A for each of its actuators based on the lower level control law. The signal sent to the kth actuator is $A_k + \bar{A}_{i+k}$, which is the addition of the global and local actuation commands. Below is pseudo-code for controlling lower and higher level groups.

```
group Lower_Level { // state ProcessorArray sensors // an array of
  level 0 members that have sensors ProcessorArray actuators // an
  array of level 0 members that have actuators Vector Dbar, Vbar, A
  IntegerArray D, V

  // methods
  void compute_Dbar_and_Vbar() {
    for all sensors i, in parallel
      D[i] = sensors[i].displacement()
      V[i] = sensors[i].velocity()
    Dbar,Vbar,A = control_law(D, V, beam_properties, interpolation_function)
    for all actuators i, in parallel
      actuator[i].apply(A[i])        // apply low level actuation command
  }

  void affect_actuator(Vector Abar) {
    for all actuators i, in parallel
      actuator[i].apply(Abar[i])     // apply high level actuation command
  }
}


group Higher_Level {
  // state
  Vector Abar

  // methods
  void start() {                     // code entry point
    for all Lower_Level members i, in parallel
      member.compute_Dbar_and_Vbar()
    // read Dbar,Vbar from members and compute response
    compute_Abar(beam_properties, interpolation_function)
    for all Lower_Level members i, in parallel
      member.affect_actuator(Abar[i])
    pause(constant)                  // compute at regular intervals
    start()
  }
}
```

An AC Hierarchy can be used to implement such a hierarchical controller, although there are additional considerations. Ordinarily, the first step in setting up a hierarchical controller is to choose n_g, the number of summarised sensor readings, $\langle \bar{d}, \bar{v} \rangle$, that will be passed to the higher level controller. Small values of n_g will increase the (undesired) coupling between the high and low level controllers, while values that are too large will make the actuation computation prohibitive. (The number of required arithmetic operations is proportional to ${n_g}^2$.) In an AC Hierarchy, the worst-case computation speed of a group is inversely proportional to its spatial extent[1]. Since it is critical to keep the computation time of each controller low, the choice of n_g also places an upper bound on the spatial extent over which the amorphous hierarchical controller has authority. As a result multiple high level controllers may be required to control the entire area of the processors.

Once an AC Hierarchy has been constructed, two levels, h and l, are chosen for the groups that will simulate the higher and lower level controllers. The constraints governing those choices are that the number of level l groups in each level h group must be at least n_g and that each group must be able to compute its actuation responses faster than the critical response time for the controller it is simulating. The critical response time for a controller is inversely proportional to the highest spatial frequency over which it has authority and also depends on the physical properties of the beam material. The highest spatial frequency that a level l group can effectively control depends on the number of its sensors and their placement, while that of a level h group is determined by n_g. Each level h group will independently try to control its region with a hierarchical controller.

At this stage the controller is setup and ready to be exercised. Each level l group collects its sensor readings, and computes both the local response and the aggregated reading. The aggregate is routed to its containing level h group, while the local responses are routed to its actuators. When the level h groups receive all the aggregates from their level l subgroups, they compute the global response, and route to each level l subgroup the corresponding segment of the response. Upon arrival of a segment, a level l group routes its components to the corresponding actuator. The success of the two-phased actuation (as opposed to computing the sum of responses and applying it once) depends heavily on the light coupling of the two levels' controllers. Some approaches to reducing the coupling between the lower and higher level controllers are discussed in [6]. Increasing the number of levels in the hierarchical controller without sacrificing its effectiveness is a non-trivial task. One of the crucial obstacles would be the decoupling of the controllers at the different levels. Other multilevel approaches to distributed control are either not as effective as that of Hall et al. or tend to require non-local sensor readings, thereby making them unsuitable for an amorphous computer.

3.3 General Benefits

The examples mentioned above illustrate several important benefits provided by the AC Hierarchy. Specifically, the AC Hierarchy

[1] Computation on a group, say at level n, may rely on communication among its level $n-1$ member groups. So if message latencies within the group are large because of widely separated members, then the computation latency of the group is also potentially large.

Simplifies programming of an amorphous computer

Groups as computational units: In the sorting example, although the actual computations (e.g. comparisons of list elements) are performed by the individual processors, the sorting program never has to address them directly. The group metaphor provides a useful programming abstraction for decomposing tasks without concern for the actual details of how the group coordinates to act as a unit.

In order to support this abstraction, groups must have a means of coordinating behavior and for passing messages between the groups. The coordinated group behavior can be implemented by choosing a group leader or by consensus. Inter-group communication can be easily implemented recursively in terms of lower level group communications.

Partitioning tasks across levels: In the sensory and actuation example, the first level implements a localized algorithm for estimating new actuation values, while the second level runs a global algorithm that attempts to stabilize the overall actuation levels. The abstraction allows the programmer to easily express different algorithms for different levels of the hierarchy.

Facilitates analysis of algorithms

Because the logical organization reflects physical proximities of the processors, the performance of algorithms expressed in terms of the AC hierarchy can be readily analyzed. The bounds on group diameters provide estimates on the time taken to coordinate a group activity. The maximum distance associated with edges at a specific level provides an estimates on communications costs between groups. These two factors help determine the expected performance of algorithms.

Increases Efficiency and Robustness

Hierarchies are a commonly used structure to increase the efficiency of a particular task. For example, routing schemes often use hierarchies to minimize storage requirements of routing tables. In the sensory example, control is based on detailed local information and more coarse knowledge of distant data in order to improve response times.

Aggregating processors to act as a unit can increase the reliability of the unit above that of a single processor. Robustness is often achieved through replication of tasks (e.g. mergesort) or data (e.g. routing tables) amongst groups.

4 Construction of an AC Hierarchy

This section presents two algorithms for constructing the first level of an amorphous computing hierarchy, and a third algorithm that constructs higher levels of the hierarchy. These algorithms satisfy several requirements that ensure their applicability to an amorphous computing environment. First, they are both efficient and scalable, e.g. they can quickly

DO

Leader Election: Processors compete for leadership. A symmetry-breaking mechanism is used to determine the winner.

Recruiting: New leaders recruit their neighbors. Processors that are recruited by a leader become members of the leader's group and stop competing for leadership.

UNTIL

Termination: All processors are either leaders or group members. Individual processors must be able to detect termination.

DO [optional]

Redistribution: Processors transfer from larger groups to smaller groups. This phase attempts to reduce the disparity in sizes between groups at the same level.

Figure 5: Parallel Group Formation

organize trillions of processors. Second, the construction of the hierarchy is spontaneous, and uses identical processors with no a-priori knowledge about the environment, e.g. no global ids. Third, the communication primitives required rely only on broadcasts between neighboring processors. Finally, these algorithms are tolerant of message loss. The algorithms presented in this section also include settable parameters (e.g. for diameter bound), and guarantee coverage. In addition, the algorithms attempt to minimize communication interference and produce robust groups that can tolerate member failures.

The algorithms introduced in this section follow a basic structure for parallel formation, illustrated in Figure 5. All processors vie for leadership, and once elected, they recruit members. The algorithm terminates when all processors are either leaders or members of groups. The remainder of this section first introduces the *overlapping clubs* and *tight clubs* algorithms for constructing the first level of the hierarchy (Section 4.1), and then introduces the *tree regions* algorithm for establishing higher levels (Section 4.2).

The code presented in this section executes on an individual processor. A system level thread listens for incoming messages and places them on a bounded queue. A single user thread executes the program, sends messages, and retrieves incoming messages from the queue. Messages are either broadcast or peer-to-peer. In both cases, the source identifier is included in the message. With peer-to-peer communications, the message is prefixed with a destination identifier.

4.1 Establishing the First Level

The combination of a dense population of processing elements and a broadcast mechanism for communications creates an inefficient environment for a point-to-point messaging mechanism. Unless processor behavior is coordinated, the potential for communications interference between neighboring processors is large. An important goal of the first level of the hierarchy is to reduce the interference problems between neighboring processors.

The algorithms for constructing the first level groups presented in this section do not depend on point-to-point communications between the processors. Rather, the algorithms only use the substrate's local broadcast mechanism for message passing. Once the algorithms form the first level processor groups, an efficient point-to-point message protocol between the groups can be established, where group behavior is coordinated in order to reduce communications interference between processors.

In addition, the algorithms do not require globally unique identifiers. Rather, each processor chooses an id that is locally unique within a two hop radius. This can easily be accomplished using a random number generator and a simple correction scheme that relies only on local communication.

A *club* is a group of processors at the first level of the group hierarchy. The rest of the section introduces two algorithms, *overlapping-clubs* (Section 4.1.1) and *tight-clubs* (Section 4.1.2) for constructing cooperative groups at the first level.

4.1.1 Overlapping Clubs

The first algorithm produces clubs that are allowed to share processors, i.e. a processor may belong to more than one club, and there is an intra-group communication latency bound of 2 processor hops. Leader election uses random numbers selected from some statically determined range R. All processors start out as potential leaders. Each processor chooses a random number once and uses this as a "delay" after which it can become a leader. When a processor becomes a leader, it broadcasts that fact, which causes all processors within "earshot" to stop vying for leadership. These processors become members of the leader's club. Figure 6 describes the details of the algorithm, and Figures 7 and 8 illustrate the progression of the algorithm. The idle period is a fixed number less than R, hence termination is guaranteed after time R. Each processor chooses R based on the number of its neighbors. Termination implies that all processors belong to a group and hence coverage is guaranteed.

Clearly, the extent of each club formed is determined by the area covered by the intrinsic broadcast mechanism of each processor. Since the followers cannot become leaders themselves, each club should contain exactly one leader. This guarantees a minimum separation of a broadcast radius between leaders and therefore reduces the overlap between neighboring clubs. In Figure 8, no leader is within the broadcast radius of another leader.

```
integer R                    ;   range for choosing random numbers
integer T                    ;   number of trials (variant_2)
boolean leader, follower     =   false

procedure MAIN()
   OVERLAPPING_CLUBS

procedure OVERLAPPING_CLUBS ()
   r := random(1,R)
   while (not follower and not leader)
      if (r > 0)
         decrement r by 1
         if (not_empty(msg_queue))
            if (pop(msg_queue) = "recruit")
            follower := true
      else
         leader := true
         broadcast("recruit")
   if (follower)
      listen for other leaders until timeout
```

Figure 6: Algorithm for Overlapping Clubs

A leadership conflict occurs when two or more processors less than a broadcast radius apart declare leadership at the same time. This violates the minimum separation between clubs. The conflict results from either the inherent asynchronicity of the processors or because two processors choose the same random number. If occasional violations of the minimum separation can be tolerated, then R can be chosen to minimize the probability of conflicts. In that case, the time required for the algorithm is simply R. If minimum separation between leaders is required, it can be achieved by running multiple rounds of the algorithm. After time R, leaders can detect conflicts by conferring with their neighbors. If there is a conflict, the conflicting leaders and their members run another round of overlapping clubs. The algorithm is similar to the maximal independent set algorithm described in [9] and has similar expected time complexity of $O(\log n)$, where n is the total number of processors. However, overlapping clubs is well suited for a broadcast environment because of its low message overhead and low synchronization requirements.

Overlapping clubs produces valid groups even in if message loss occurs. If a processor hears a collision, then it can assume that at least two of its neighbors declared leadership, and therefore it stops competing for leadership. During leadership conflict resolution, it determines who the leaders are. If a processor is not aware of the a message loss, then it may continue to vie for leadership, and possibly conflict with one of its neighbors. Again, this situation will get resolved.

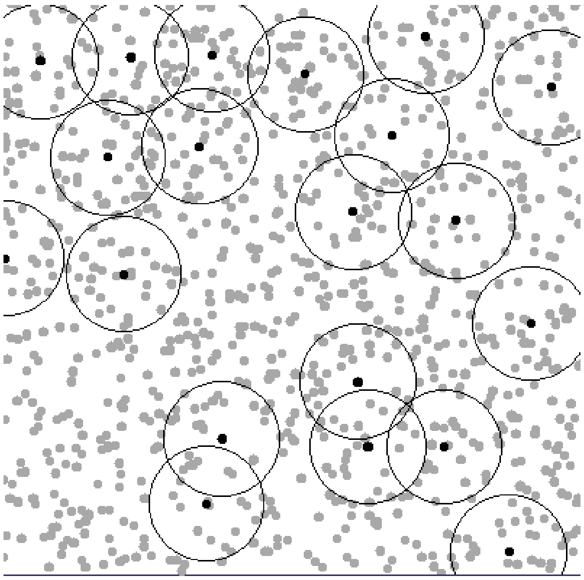

Figure 7: This figure shows leaders forming clubs. The circles circles indicate the area within "earshot" of the leader (at the center of the circle). All processors within this area are recruited as members of the leader's club.

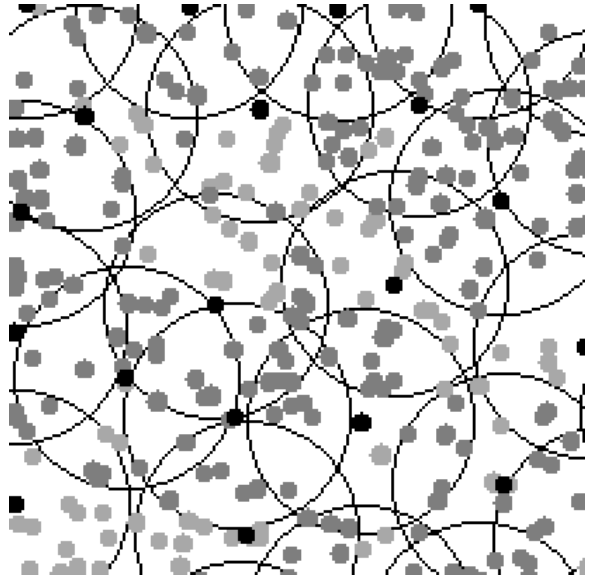

Figure 8: This shows a close-up of the final clubs formed. Processors with a darker shade of gray belong to more than one club because they are in the overlapping region of several leaders broadcast range.

Features and Properties: The clubs produced satisfy the level one requirements of an AC hierarchy. The intra-group communication bound is at most two hops. Each club leader can communicate with all its members in one hop. An edge requires at most three hops to deliver messages between the leaders. An efficient implementation of an edge is to use only the processors in the overlap region to relay messages between the clubs. Such an edge requires at only two message hops between the leaders. Leaders chosen during the group formation phase can act as coordinators for the group activities. They are a good candidate for coordinating behavior because they can communicate directly with all members of the club. However, if the leader fails, the club may become disfunctional because members may no longer be able to communicate with each other. If a leader fails but the club remains connected, a new leader can be elected. If the club is disconnected, then members elect leaders to form new clubs. The likelihood of a club disconnecting decreases as the number of club members increases.

In addition to the AC hierarchy properties, the overlapping clubs have several unique characteristics:

- *Bounded Degree:* There is a constant upper bound of 37 on the degree (i.e. number of neighbors) of each club. The bound is derived from the densest packing of circular broadcast regions in a plane, with a minimum separation of a radius between their centers.

- *Total Number of Clubs:* There is a statically determined upper bound on the total number of clubs. This number can be derived from the maximal packing of circles in a plane and the area of the plane.

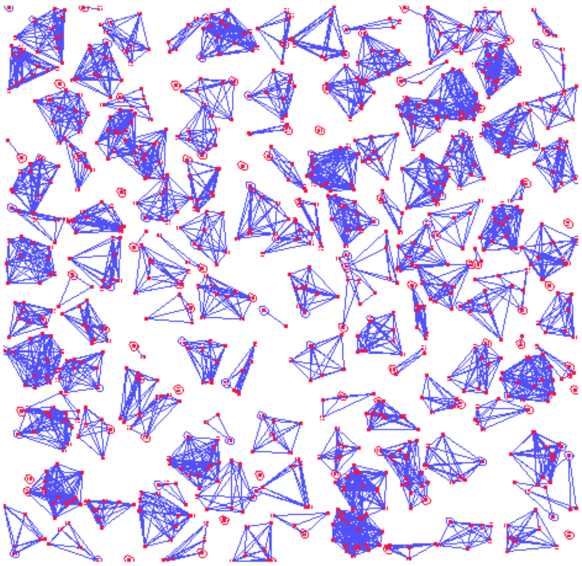

Figure 9: Result of the first phase of the tight club algorithm.

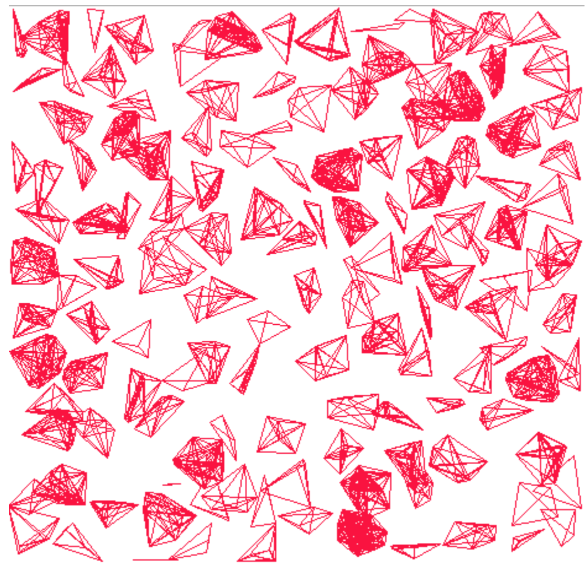

Figure 10: Result of the second stage of the tight club algorithm.

The proofs for these properties are given in the appendix. As a result of these unique properties, there is a means for measuring the interference between clubs and determining a robust message protocol for the edges between clubs. In addition, there is an estimate of the size and complexity of the level one graph. This aids in the analysis of algorithms built on top of overlapping clubs.

4.1.2 Tight Clubs

A second algorithm, *tight-clubs*, addresses issues of fault-tolerance by constructing clubs that are more tightly coupled. The clubs formed by overlapping-clubs have a single point of failure, namely the leader. The leaders act as the loci of communications because they can communicate directly with all member of the group. If the leader fails, the members of the club could potentially become disconnected internally, and the club would no longer be able to function as a unit. Also, even if the members are still connected, there may not be a single member that can communicate directly to all other members. Thus, the leader is not easily replaceable in the event of faults.

The *tight-clubs* algorithm produces groups of processors where all members of the group can communicate with each other directly. In addition, each element is required to be a member of exactly one group. The tighter coupling between the group members eliminates the dependency and associated bottlenecks of having one group leader for intra-group communications. Any member can be the leader and the failure of individual members does not affect the connectivity within the club.

Phase I - Construction of Tight Clubs: The first phase uses a greedy algorithm to construct tight-clubs. Processors use their locally unique ids to determine leadership.

```
procedure MAIN()
  Check-Ids

procedure Check-Ids()
  if (my_id < lowest id in active)                  ; active Initially set to all neighbors
     leader := true
     Recruit()
  else
     Listen-For-Msgs()

procedure Recruit()
  while (potential not-empty)
     low_id := lowest id in potential
     send("recruit") to low_id
     wait for msg from low_id
     recruit_nbrs = pop(msg_queue)                  ; msg is false if low_id is already recruited
                                                    ; otherwise it is the list of low_id's neighbors
     if (recruit_nbrs ≠ false)      ; ensure that all club members can communicate directly
        potential := potential ∩ recruit_nbrs       ; potential initially set to all neighbors
  broadcast("inactive", my_id)
  Done()

procedure Listen-For-Msgs()
  wait for msg
  if (msg = "recruit")
     broadcast("inactive")
     send(neighbor_list) to msg_source_id
     Done()
  else if (msg = "inactive")
     active := active - msg_source_id
     Check-Ids()

procedure Done()
  while (active not-empty)
     wait for msg
     if (msg = "recruit")
        send(false) to msg_source_id
     else if (msg = "inactive")
        active := active - msg_source_id
```

Figure 11: Tight Club Algorithm

Processors whose ids are local minima declare leadership and greedily recruit neighboring processors. Figure 11 illustrates the steps involved in the first phase of the tight club algorithm. Initially, all processors are active. Each processor then checks whether it has the lowest processor id among its active neighbors. If so, the processor becomes a leader and attempts to recruit its active neighbors in the order of increasing id values. The leader recruits a processor only if it can directly communicate with all current members of the tight club. When a processor is recruited to a club, it becomes inactive and notifies all its neighbors of this fact. Active non-leader processors listen to messages and wait either to be recruited or until they are the smallest id among their active neighbors. In the latter case, they begin a new club themselves. Figures 9 shows the tight clubs formed as a result of the first phase.

Termination is guaranteed because at each step a local minimum exists, which implies that at least one processor is removed at every step. Hence coverage is also guaranteed. The worst case running time is $O(d)$, where d is the diameter of the amorphous computer in terms of processor-to-processor message broadcast hops.

Unlike overlapping clubs, the tight-club algorithm is deterministic because it does not rely on choosing random values. On the other hand, the algorithm is scalable because it does not require global identifiers. The algorithm depends only on the local uniqueness of randomly chosen processor id. The message overhead is small because processors keep track of their neighbors' state in order to determine when they have become local minima. The algorithm relies on acknowledgments to account for message loss of point-to-point recruiting messages. In the case where a processor is waiting for a single neighbor to become inactive, it can occasionally poll the neighbor to verify that progress is taking place.

Phase II - Redistribution of Processors: As can be seen from Figure 9, the first phase of the tight-club algorithm produces many clubs that are small and may even contain only a single member. The goal of the second phase of the algorithm is to increase robustness by increasing the number of members in small clubs whenever possible. The second phase redistributes processors from larger clubs to smaller clubs to reduce the disparity in club sizes. At the end of the second phase, a locally optimal distribution of members is achieved. Locally optimal implies that the size difference between any pair of neighboring clubs is no more than one if there are members that can be transferred.

Features and Properties: Tight clubs provide all the properties required for hierarchies. Each processor can communicate directly with each other processor in its club. Because of the tight coupling, there are many ways to efficiently coordinate the group behavior. For example, within a group, different members can be coordinators for different activities, or the group can operate by consensus. Any processor can assume the role of a leader, and consensus protocols are simple to implement. In the case of failures, small clubs can attempt to recruit members in a manner similar to the re-distribution phase to increase their robustness. The edge bound between any two members of different groups is at most three hops.

4.2 Establishing Higher Levels of the Hierarchy

This section introduces *tree regions*, an algorithm for constructing groups of processors at different levels of granularity. The algorithm takes advantage of the coordinated group behavior and inter-group communication capabilities that are provided by the immediately lower level to construct processor groupings. The groups formed by this algorithm also conform to the requirements of an AC Hierarchy, and therefore can be applied recursively to create multiple levels.

Tree regions generalize techniques from overlapping clubs and tight clubs to form higher levels of the hierarchy. The leader election is implemented using the countdown mechanism. The main difference between tree regions and the previous algorithms is in the recruiting phase. The algorithm includes a parameter h that determines the diameter bound of the groups formed. When constructing level n, a newly elected leader not only recruits its immediate level $n-1$ neighbor groups but also recruits neighboring groups up to a distance h away. The distance is measured by level $n-1$ group hops. The mechanism used for recruiting neighbors resembles growing a spanning tree.

The construction of level n groups proceeds as follows. Each level $n-1$ group chooses a random number from the range 1 to R and begins to count down. If it reaches zero without being interrupted, the group becomes a tree root. It then seeds a tree of fixed height h by recruiting its neighbors as children. If a group receives a recruiting message before counting down to zero, it becomes a child of the sender. The child group then tries to recruit its immediate neighbors as its children, unless it is at a depth h from the root. This guarantees that the trees are of bounded height h. Eventually all groups are either recruited to a tree or have seeded a tree. Figure 12 presents the procedures used by this algorithm for seeding the tree and recruiting neighbors.

Figures 13 shows the algorithm running on top of overlapping clubs. In this case, the leaders of the clubs coordinate the club's decision to seed or join a tree. Club members route messages between leaders of neighboring clubs. The lines connecting club leaders represent the spanning trees, of bounded height 2, formed by the algorithm. Figure 14 shows the final set of regions and their corresponding trees.

The algorithm can be modified to allow either overlapping or non-overlapping groups. Thus, the *overlapping clubs* algorithm is a special case of tree based region growing where $h = 1$ and groups are allowed to overlap.

Due to asynchronicity, a several hop message may reach a destination before a single hop message. Therefore branches do not necessarily grow at the same pace. If a group chooses the first recruiter as its parent, it may not be at the lowest possible depth from the root. The algorithm allows a member to change both its depth and tree affiliation if it hears a recruiting message that will reduce its tree depth. The member propagates its new deoth and new tree affiliation to its children. This improves the distribution of group sizes. The algorithm uses acknowledgements and exponential backoff to deal with message loss and collisions.

```
integer R                     ;   range for choosing random numbers
integer T                     ;   expected time to create a region/tree
integer MAX_H                 ;   maximum height a tree can grow to
integer current_h = inf
boolean competing = true

procedure Main()
r = random(R) * T
Election_Loop()

procedure Election_Loop()
if (not_empty(msg_queue))
   msg = pop(msg_queue)
   if (msg = "recruit")
      Recruited(msg)
else if (competing)
   decrement r
   if (r = 0)
       Seed_tree()
else Election_Loop()

procedure Recruited(msg)
if ((msg.h ¡ current_h) && (msg.h < MAX_ H))
   broadcast ("recruit", (h+1))    ; recruit neighbors for depth h+1
   current_h = msg.h
   competing = false
Done()

Seed_Tree()
root := true
   broadcast ("recruit", 1)    ; recruit neighbors for depth 1
Done()

Done()
wait for time out
```

Figure 12: Algorithm for a Tree based Hierarchy

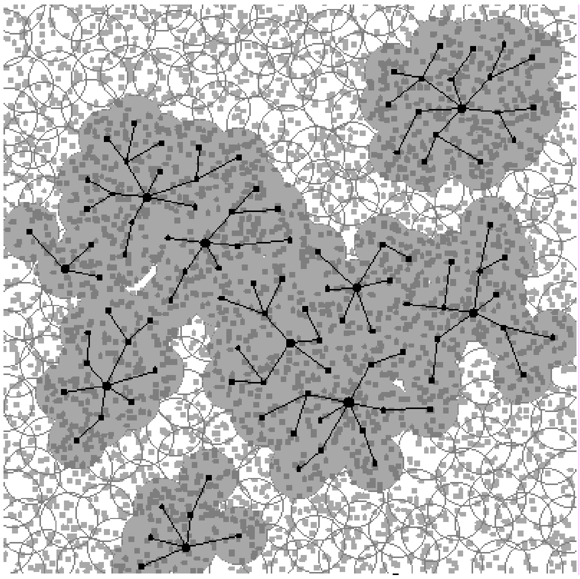

Figure 13: Early stage of trees sprouting on top of overlapping clubs. The tree depth is at most two.

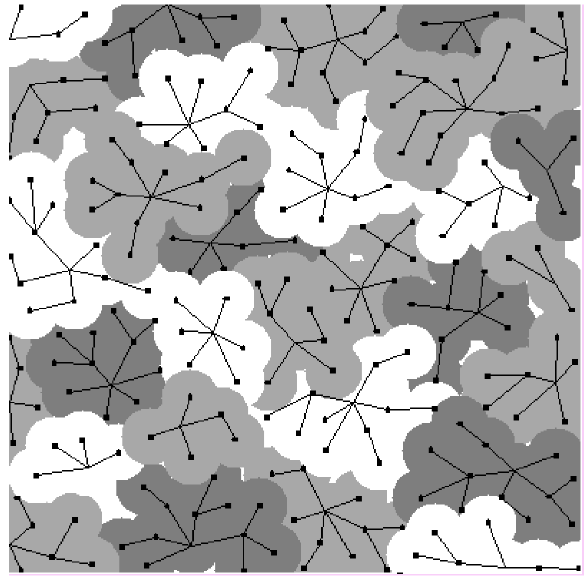

Figure 14: Final level 2 groups formed

In order to ensure proper spacing between trees it is important to prevent trees from sprouting too close in time and allow trees sufficient time to grow. Let T be an estimate of the time taken to grow a tree of height h, i.e. an estimate of the time rquired for a message to travel distance h away. Each group chooses a random number from the range R and then multiplies it by T. This causes trees to sprout roughly at intervals of time T and therefore reduce competition. The range R is chosen in a manner similar to overlapping clubs so as to reduce the probability that a node within distance h of a tree root is also a root. Hence R is based on the neighborhood size of h hops away. The algorithm is scalable because it only depends on density and not the total number of processors. Termination is guaranteed after time $R \cdot T$, because all nodes will have either been recruited to a tree or declared themselves tree leaders. Hence coverage is also guaranteed.

Features and Properties: The tree provides an important structure for control, synchronization and gathering/scattering of information within the group. It can be used for coordinating group behavior. The tree height is restricted to h, therefore the diameter of the group is bounded by $2h$ lower level edges. An edge between two groups has a communication latency from any member of one to any member of the other of at most $4h+1$ lower level hops. Both bounds are subject to node failures. In the case of a fault the tree can be reorganised to use other lower level edges between members. The likelihood of a group being completely disconnected by faults is very unlikely.

One unique feature of the groups generated by the tree regions is that there is a bound of $O(P_{n-1} \cdot h^2)$ on the number of processors in the group, where P_{n-1} is the upper bound on the physical distance associated with a lower level edge. This is because the tree grows in a physical plane and the maximum area it can occupy is the circle of radius $P_{n-1} \cdot h$ centered at the root. Hence, the number of processors in a group grows quadratically in the height of the tree, as opposed to exponentially.

5 Related Work

The leader election mechanisms used by the clubs and tree-regions algorithms, are similar to other parallel algorithms for Maximal Independent Set (MIS) problem described by Luby[9], and applied to asynchronous networks in [10, 2]. In this case however, the properties of coverage and low message overhead are more important than obtaining a maximal independent set. Many synchronous and asynchronous algorithms for generating spanning trees are described in [10]. [3] presents an algorithm that produces tree based clusters by using global ids.

Many different papers have suggested aggregation and hierarchies of aggregates as a possible mechanism for programming and hiding complexity. Swarm [7] presents a hierarchy based language for simulation environments. Concurrent Aggregates [5, 4] also presents a language based on aggregates being treated as objects which is to be used to program a parallel machine. The main difference is the spatial nature of the amorphous computer and the direct mapping between the hierarchy language and the distribution of tasks. The AC hierarchy preserves locality and allows applications that depend on spatial locality.

6 Conclusion and Future Work

Amorphous computing is a new field of study that attempts to identify the principles and languages for obtaining coherent behavior from the cooperation of massive numbers of unreliable processor elements connected in unknown and irregular ways. AC hierarchies provide an important mechanism for structuring an amorphous computer and enabling aggregated computation and communication. The AC hierarchy provides a good abstraction for hiding complexity and communication bounds and locality properties for designing and analyzing efficient and robust algorithms.

Work is underway to develop a prototype of an amorphous computer. This will allow us to demonstrate the feasability of the algorithms on actual hardware. The prototype will incorporate appropriate sensors and actuators to demonstrate sensory applications.

This paper presents only the first steps towards programming an amorphous computer. Additional algorithms will be investigated for generating different hierarchies that achieve the same goals. We are working on developing the language on top of the AC hierarchy and supporting group coordination and edge communication primitives, independant of the algorithm used to construct the groups. The language will be used to map other sensory applications to the AC hierarchy. Finally, we are simultaneously exploring other mechanisms used by biological systems to achieve coherent behavior from vast numbers of cooperating components.

References

[1] Hal Abelson, Tom Knight, and Gerry Sussman. Amorphous computing. *White paper*, October 1995.

[2] Baruch Awerbuch, Lenore Cohen, and Mark Smith. Efficient asynchronous distributed symmetry breaking. In *Proceedings of the 26th Annual Symposium on the Theory of Computation*, Montreal, Canada, May 1994.

[3] Baruch Awerbuch, Andrew Goldberg, Michael Luby, and Serge Plotkin. Network decomposition and locality in distributed computation. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 364–369, Research Triangle Park, North Carolina, October 1989.

[4] Andrew A. Chien. *Concurrent Aggregates (CA): Supporting Modularity in Massively-Parallel Programs.* MIT Press, Cambridge, Massachusetts, 1993.

[5] Andrew A. Chien and William J. Dally. Concurrent aggregates. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* ACM, March 1990.

[6] Steven R. Hall, Edward F. Crawley, Jonathan P. How, and Benjamin Ward. Hierarchic control architecture for intelligent structures. *J. Guidance, Control, and Dynamics*, 14(3), May-June 1991.

[7] David Hiebeler. The swarm simulation system and individual-based modeling. In *Decision Support 2001: Advanced Technology for Natural Resource Management*, Toronto, Canada, September 1994.

[8] Jonathan P. How and Steven R. Hall. Local control design methodologies for a hierarchic control architecture. *J. Guidance, Control, and Dynamics*, 15(3), May-June 1992.

[9] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal of Computing*, 15(4), November 1986.

[10] Nancy Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, Wonderland, 1996.

[11] Paul F. Tsuchiya. The landmark hierarchy: A new hierarchy for routing in very large networks. In *Proc. of the SIGCOMM '88 Symposium on Communications Architectures and Protocols*, Stanford, CA, August 1988.

Appendix A: Physical Properties of Overlapping Clubs

The physical properties of the overlapping clubs presented here depend on the model of communication - a circular broadcast of fixed radius r in a plane. Many of these geometric properties can be derived for broadcasts of other shapes as well.

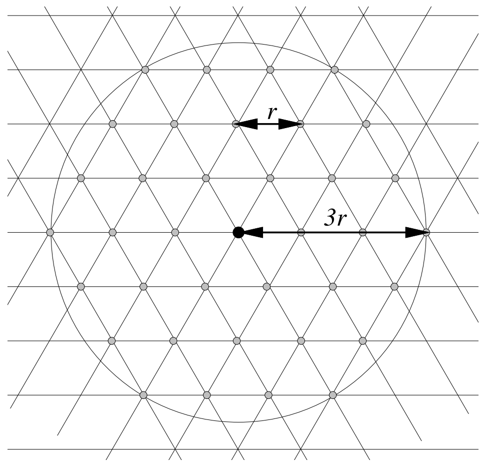

Figure 15: Hexagonal Packing of Neighboring Leaders

Constant Upper Bound on Degree of Clubs: The overlapping club algorithm guarantees that no two leaders are closer than the communication radius r to each other. According to the definition of neighbors in the AC Hierarchy, two clubs are neighbors if any of their members can communicate directly. If two leaders A and B are further than $3r$ apart, then all members of A are further than r from all members of B. Hence, A and B cannot be neighbors. All neighboring leaders of a club A must lie within the circle of radius $3r$ centered at the leader of A.

The maximum number of neighbors that a club can have is the maximum number of leaders that can fit within the circle of radius $3r$. Restated as a packing problem, the maximum degree is the maximum number of points that can fit within a circle of radius $3r$ such that no two points are closer than r to each other. We know that hexagonal packing of unit distance r provides the densest packing of these points in a 2D plane. Figure 15 illustrates that at most 37 such points fit within the circle. Therefore, the upper bound on the degree of any overlapping club is 36.

Statically Determined Upper Bound on Total Number of Clubs: The total number of clubs in an amorphous computer on a 2D plane can also be expressed as a packing problem. The upper bound on the total number of clubs is the maximum number of leaders that can be packed into the plane of a given area such that no two leaders are closer than r. We know that hexagonal packing with unit distance r is the densest packing of these points. Then, one can compute the total number of clubs by overlaying a hexagonal grid on the 2D surface area and counting the number of points on the hexagonal grid.