Sequential Program Decomposition for Distributed Systems

Tamim Sookoor
Department of Computer Science
University of Virginia
sookoor@cs.virginia.edu

1. PROBLEM DEFINITION

With computing resources getting cheaper and more powerful and wireless networking technology improving, distributed systems is emerging as a very promising application domain. Distributed systems, composed of wirelessly connected, distributed, heterogeneous components, have led to the implementation of many applications. Some examples of such applications are habitat monitoring [**?**], disaster relief, and elderly monitoring. Despite these applications, and possibility of many more applications, development of software for distributed systems remains a challenge.

The most commonly used application development method for distributed systems is device level programming. This entails writing programs to be run on each device that composes the distributed system and placing this code on each device individually. Since distributed systems are usually composed of many different types of devices, such as motes, PDAs and PCs, separate programming languages might have to be used for each type of device. Also application developers will have to translate their high level vision of the application into tasks that should occur in each device in order for the distributed system to implement the application. This method of programming is error-prone, difficult to debug, and complicates application development.

AlarmNet [16], an elderly monitoring distributed system is a prime example of a distributed system hampered by device level application development. AlarmNet utilizes tiny sensors of to monitor elderly people and the data collected by these sensors can be accessed via PCs as well as PDAs used by medical staff. The sensors used in this system range all the way from accelerometers to dust sensors. In addition to these myriad sensors, PDAs and PCs, AlarmNet uses data collected through medical equipment such as blood pressure monitors as well as through devices such as weighing scales. As display devices, PDAs and PCs are supplemented by LCD equipped motes called SeeMotes. As can be seen from this description, AlarmNet is composed of a large number of different devices and using the current programming methods each of these devices have to be independently programmed in order to form the distributed system that is AlarmNet. This can be very tedious.

Device level programming, using nesC [6] and TinyOS, is currently used for distributed systems in order to efficiently utilize their constrained resources. Sensor nodes, which compose the largest part of most distributed systems in terms of quantity, are small form factor devices. Due to their small size, sensor nodes, such as the Mica mote, are usually powered by batteries. These batteries have a limited lifetime which has to be maximized in order to enable the distributed system to have a sufficiently long lifetime. In order to Minimize power consumption and extend system lifetime other resources, such as processing power, memory, and bandwidth have to be constrained. All of these factors lead to sensor nodes, and thus distributed systems being resource constrained. Programming at the device level enables application developers to optimize the utilization of these scarce resources. The dynamic nature of distributed systems are due to their physical characteristics as well as the nature of some of their applications. Most distributed systems use wireless communication to link the devices forming the system. Wireless links change in quality over time and do not have the reliability of wired links. Thus, the systems that depend on these links must be able to adapt to changing link quality at the device level and this is achievable through device level programming.

Recently, system level programming models for distributed systems, such as TinyDB and Regions Streams [11], have been implemented. These models enable programming for distributed systems without having to consider machine boundaries and architectures but their expressiveness is restricted to querying distributed systems. This restriction makes these methods limits the suitability of these methods for programming distributed systems applications.

We propose a programming method that enables distributed system applications to be developed at the system level while maintaining the expressiveness of device level programming. Our method would enable application developers to program in a high level language such as Java or Python. The program created by the developer would be automatically decomposed into scripts to be run on the devices composing the distributed system. We anticipate this method to produce object code that runs on devices less efficiently than if the code was developed at the device level, but are willing to make this tradeoff in order to enable distributed systems application development to be easier than it is today.

2. RELATED WORK

There are many algorithms proposed to decompose sequential code into modules and identifying dependencies in loops in order to parallelize them. There is also a considerable amount of research done in parallelizing sequential programs for multiprocessor systems. There are also a few projects aimed at parallelizing sequential programs for distributed systems. This section will describe some of these research.

Researchers at Drexel University developed Bunch [5], a tool that automatically clusters software systems using a genetic algorithm. Bunch applies a genetic algorithm on module dependency graphs (MDGs) of software. The algorithm optimizes for low inter-cluster relationships and high intra-cluster relationships. A genetic algorithm similar to this will be evaluated for sequential code parallelization. Yet changes will have to be made to the algorithm, particularly the objective function, since, in distributed systems, there will be a tighter dependence between some lines of code and the device on which they can run and also since the algorithm described in this paper has problems generating output MDGs that are as expected.

A paper by Wiggerts [15]provides an introductory survey of various clustering techniques that have been successfully applied to system modularization. Some of these techniques could be incorporated in this project.

A paper by researchers at Washington State University [12] describe a knowledge-based algorithm for parallelizing sequential programs. The system uses a pattern matcher to match input programs against a library of commonly occurring parallelizable constructs in order to identify code segments that can be parallelized. It also enables users to modify code based on suggestions from the system in order to obtain the most efficient parallel code. The pattern matching approach could be used to identify parallelization in our system, but since we are aiming for maximum automation, we would not incorporate the interactive nature of this project.

A paper by Banerjee et al. [2] provides an excellent survey of various methods to identify dependencies within programs, particularly within loops, and transform the program in order to make it parallelizable. Some of these methods will be used in order to identify code dependencies.

Blume et al. have developed Polaris [4], a system that restructures Fortran code for high-performance multiprocessor computers.They use dependency analysis techniques similar to those described by Banerjee et al. in addition to variable privatization and substitution. Since Polaris is aimed at restructuring code to be run on parallel on homogeneous components, namely processors of a multiprocessor computer, it does not directly solve the problem of parallelizing code to be run on the heterogeneous components that make up a distributes system. Yet some of the ideas described in the paper can be utilized in our system.

The Coign system [7] developed by researchers from Microsoft and the University of Rochester is the first system to automatically partition and distribute a binary application. It partitions COM applications between two tightly interconnected hosts within a local area network. Coign constructs a graph model of the inter-component communication of the application to be partitioned and applies a graph-cutting algorithm to partition the application. Since this system is targeted at wired distributed system on PC class devices it does not consider any of the constraints, such as power, imposed by wireless distributed systems.

The ABACUS system [1], developed at Carnegie Mellon University, extends Coign by showing that incorporating dynamic information into component placement decisions can improve application performance. This system monitors application running on clusters of client and server PCs and dynamically changes function placement. This application has similar shortcomings as the Coign system in that it does not consider resource constraints as its target is PC class devices. Also this system does not perform any code partitioning which is an important component of the programming environment that we home to implement. Researchers at the University of Washington developed the Emerald system [8] which allow objects written in the Emerald language to move freely within a distributed system. The code migration is directed by source-level programmer annotations. This is relevant to our project since we hope to allow programmers to annotate code and provide hints to the programming environment on how to partition the code yet it is not a solution to the application since it requires explicit programmer control to trigger code migration, does not support modern distributed systems, and is targeted at traditional applications.

Legion[10], developed at the University of Virginia, is a language independent, scalable, object-oriented operating system for wide-area infrastructure networks. It runs on wide-area assemblies of workstations, supercomputers, and parallel supercomputers providing system services that create the illusion of a single virtual machine to users. This system provides the abstraction over different networked devices that we hope to provide users yet it does not deal with code partition. Also, it is aimed as wired wide-area networks of server class devices and would not meet the requirements of distributed systems that are composed of heterogeneous wirelessly connected components.

University of California, Berkeley, developed Mate[9] a virtual machine for sensor networks. This virtual machine allows high level scripts to be written and run on sensor networks. While this makes programming easier than the low level that programming that is common using systems such as nesC on TinyOS, it does not provide the abstraction over the various classes of devices that we desire. Nor does it automatically partition code for various classes of devices.

Whitehouse, et al., also working at Berkeley, implemented Marionette [14], a tool suite that provides the ability to call function and read or write variables on pre-compiled programs on sensor nodes during runtime. This tool suite runs code on various classes of devices yet it works with code written in nesC. Also, it does not group code that can be run on a particular device instead going through the code in program order and executing each line of code at the appropriate location.

Researchers at Cornell University developed MagnetOS[13, 3]. MagnetOS is a distributed operating system for ad-hoc and sensor networks that enables power-aware, adaptive, and easy-to-develop ad-hoc networking applications. The system provides a single system image of a unified Java virtual machine over an ad-hoc collection of sensor nodes. It automatically partitions applications into components and places the components on nodes within the network. This system goes a long way in satisfying the requirements of the application that we are targeting, yet it could be improved. For one it provides the unified image over only the

sensor nodes. Many modern applications incorporate other devices, such as PDAs, that MagnetOS does not abstract over. Also it partitions the code at class granularity. This is not be the most efficient way to partition code since code may be distributed unfairly. For instance, a class that is constantly executing would be run on a node, draining its power, while another class which is rarely run would reside on another node ensuring it longevity. We hope to partition code at line level granularity leading to more efficient and fairer code distribution.

As described above, various components necessary for the programming environment we envision have already been implemented. Programming environments that automatically partition code have been implemented in the distributed systems domain. Yet all of these systems are either incomplete solutions to the problem we are addressing or could be improved upon. Therefore, we claim that we are engaging in original research.

3. SYSTEM PROPOSAL

In this project we aim to develop and evaluate algorithms that can be used to decompose a sequential program into scripts to be run on multiple devices. These algorithms would be developed and applied to a number of sample programs with varying qualities, such as number of lines or number of loops, in order to decompose and distribute the code among devices.

The following techniques for code distribution would be evaluated:

1. Greedy algorithm
2. Graph search optimization
3. Random assignment

Some of these algorithms require an objective function in order to quantify the quality of the modules generated and know when a sufficient solution has been reached. Ideally we would like the objective function to take into consideration user input quality of service constraints as well as constraints of the distributed system resources. For this project we make a simplification by assuming a high quality partition to be one where there is the fewest messages passed between devices.

A sample of the input code, which is for illustrative purposes of the various techniques, is displayed below:

3.1 Greedy Algorithm

The first step in applying the greedy algorithm for code distribution would be the generation of a line dependency graph (LDG). This graph would capture the dependencies between lines of the sequential program as well as the choice of devices each line has to execute. The LDG for Fig. 1 is shown in Fig. **??**.

The first step for all parallelization algorithms listed above would be assigning lines of code that are dependent on a particular device to that device. For instance

```
lightVal = nodes[i].lightReading
```

has to be executed in a node that has a light sensor and thus would be assigned to such a node. The code with lines assigned to dependent devices would be as follows.

```
temp = 0;                                (1)
for (i = 0; i < NUM_NODES; i++)          (2)
{
 val = nodes[i].lightReading;            (3)
 if (val < 50)                           (4)
 {
  printf("Error");                       (5)
 }
 else                                    (6)
 {
  time = node[i].localTime;              (7)
  hashTable.store(temp, time, val);      (8)
 }
 pda1.display(val);                      (9)
 temp++;                                (10)
}
```

Figure 1: Sample input code

```
for (i = 0; i < NUM_NODES; i++)
{
 val = nodes[i].lightReading; // mote
 if (val < 50)
 {
  pda1.display(val);          // PDA
 }
 else
 {
  time = node[i].localTime;   // mote
  hashTable.store(time, val); // PC
 }
}
```

Figure 2: Code after lines dependent on devices are assigned

The various parallelized module generation techniques would be applied to this code in order to produce the modules to be run on the various devices comprising the distributed system.

3.2 Greedy algorithm

The system would go through each line of code and assign it to a device that optimizes the objective function. It would aim to make decisions based on the current state of the code with the hope of ending up with a globally optimized code. For instance the line:

```
if (lightVal < 50)
```

Figure 3: Line of code to be assigned using greedy algorithm

would be assigned to mote, since in Fig. 2 two lines of code are assigned to mote while only one each are assigned to PC and PDA. Therefore, in hopes of minimizing the number of packets passed, this line of code would be assigned to mote resulting in the following assignment:

Now since three lines of code have been assigned to mote,

```
for (i = 0; i < NUM_NODES; i++)
{
 val = nodes[i].lightReading; // mote
 if (val < 50)                // mote
 {
  pda1.display(val);          // PDA
 }
 else
 {
  time = node[i].localTime;   // mote
  hashTable.store(time, val); // PC
 }
}
```

Figure 4: Code after first assignment using greedy algorithm

the remaining line of code:

```
else
```

Figure 5: Line of code to be assigned using greedy algorithm

will also be assigned to mote resulting in the assignment shown in Fig. 6. Now the code can be decomposed into three scripts to be run on the mote, PC, and PDA as shown in Fig. 7, 8, 9.

```
for (i = 0; i < NUM_NODES; i++)
{
 val = nodes[i].lightReading; // mote
 if (val < 50)                // mote
 {
  pda1.display(val);          // PDA
 }
 else                         // mote
 {
  time = node[i].localTime;   // mote
  hashTable.store(time, val); // PC
 }
}
```

Figure 6: Code after second assignment using greedy algorithm

```
 val = nodes[i].lightReading;
 if (val < 50)
 @RMI pda1(val)
 else
 {
  time = node[i].localTime;
  @RMI PC(val, time)
 }
```

Figure 7: Script generated by greedy algorithm to be run on the motes

```
if @RMI
{
 hashTable.store(time, val);
}
```

Figure 8: Script generated by greedy algorithm to be run on PC

```
if @RMI
{
 pda1.display(val);
}
```

Figure 9: Script generated by greedy algorithm to be run on PDA

4. EXPERIMENTAL SETUP

The algorithms described in the previous section would be applied to code segments ranging in size from 10 lines to 100 lines and the outputs would be evaluated in terms of the number of messages that have to be passed between the modules generated. There would be three classes of devices, motes, PCs, and PDA, over which the code would have to be distributed. The application of the various algorithms will be carried out by hand.

5. EXPERIMENTAL RESULTS

The experimental results show that the quality of the modules generated by the various algorithms are almost identical for code with few lines but show great variation when the code segment is large. The genetic algorithm and the graph search algorithms produce the best results constantly while the random assignment produces the worst results with the greedy algorithm producing slightly better quality modules as can be seen in the following graph.

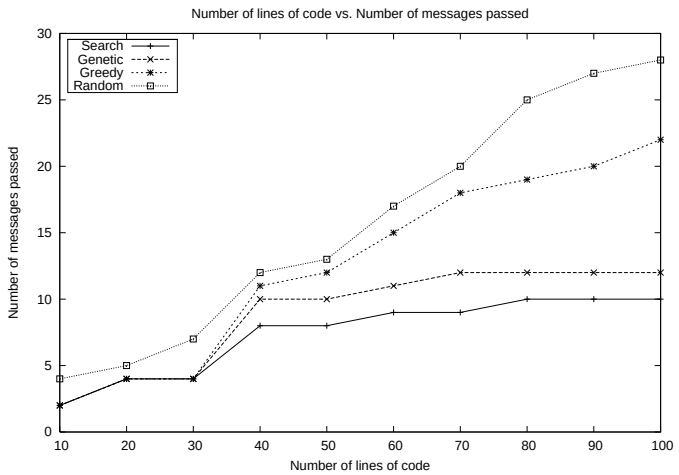

Figure 10: Graph of number of lines of code vs. the number of messages passed

6. REFERENCES

[1] K. Amiri, D. Petrou, G. Ganager, and G. Gibson. Dynamic function placement in active storage clusters. In *Proceedings of the USENIX Annual Technical Conference*, 2000.

[2] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, February 1993.

[3] R. Barr, J. C. Bicket, D. S. Dantas, B. Du, T. W. D. Kim, B. Zhou, and E. G. Sirer. On the need for system-level support for ad hoc and sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(2):1–5, April 2002.

[4] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Advanced program restructuring for high-performance computers with polaris. Technical report, University of Illiois at Urbana-Champaign, 1996.

[5] D. Doval, S. Mancoridis, and B. S. Mitchell. Automatic clustering of software systems. In *IEEE Proceedings of the 1999 International Conference on Software Tools and Engineering Practice (STEP'99)*, 1999.

[6] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, 2003.

[7] G. C. Hunt and M. L. Scott. The coign automatic distributed partitioning system. In *Proceedings of the Third Symposium on Operating System Design and I mplementation*, 1999.

[8] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the emerald system. *ACM Trans. Comp. Syst.*, 6(1):109–133, February 1988.

[9] P. Levis and D. Culler. Mate: a virtual machine for tiny networked sensors. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.

[10] M. Lewis and A. Grimshaw. The core legion object model. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, 1995.

[11] R. Newton and M. Welsh. Region streams: functional macroprogramming for sensor networks. In *DMSN '04: Proceeedings of the 1st international workshop on Data management for sensor networks*, 2004.

[12] C. Raghavendra and S. Bhansali. On porting sequential programs to parallel machines. URL: `http://citeseer.ist.psu.edu/145740.html`.

[13] E. G. Sirer, R. Barr, T. W. D. Kim, and I. Y. Y. Fung. Automatic code placement alternatives for ad hoc and sensor networks. Technical report, Cornell University, 2001.

[14] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler. Marionette: using rpc for interactive development and debugging of wireless embedded networks. In *IPSN '06: Proceedings of the fifth international conference on Information processing in sensor networks*, 2006.

[15] T. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *Proceedings of the Fourth Working Conference on Reverse Engineering, 1997*, 1997.

[16] A. Wood, G. Virone, T. Doan, Q. Cao, L. Selavo, Y. Wu, L. F. amd Z. He, S. Lin, and J. Stankovic. Alarm-net: Wireless sensor networks for assisted-living and residential monitoring. URL: `http://www.cs.virginia.edu/wsn/medical/pubs/tr06-alarmnet.pdf`.