An Architecture for Distributing the Computation of Software Clustering Algorithms

Brian Mitchell, Martin Traverso, Spiros Mancoridis
Department of Mathematics & Computer Science
Drexel University, Philadelphia, PA, USA
{bmitchel, umtraver, smancori}@mcs.drexel.edu

Abstract

Collections of general purpose networked workstations offer processing capability that often rivals or exceeds supercomputers. Since networked workstations are readily available in most organizations, they provide an economic and scalable alternative to parallel machines. In this paper we discuss how individual nodes in a computer network can be used as a collection of connected processing elements to improve the performance of a software engineering tool that we developed.

Our tool, called Bunch, automatically clusters the structure of software systems into a hierarchy of subsystems. Clustering helps developers understand complex systems by providing them with high-level abstract (clustered) views of the software structure. The algorithms used by Bunch are computationally intensive and, hence, we would like to improve our tool's performance in order to cluster very large systems. This paper describes how we designed and implemented a distributed version of Bunch, which is useful for clustering large systems.

1. Introduction

Distributed computing environments of heterogeneous networked workstations are becoming a cost effective vehicle for delivering highly scalable parallel applications. This trend is being supported by the availability of low-cost computer hardware, high-bandwidth interconnection networks, and programming environments (*e.g.,* CORBA, Java RMI) that alleviate the developer from many of the complex tasks associated with building distributed applications. As distributed systems are inherently loosely coupled, they will not, at least in the near future, replace specialized multiprocessor architectures which are better suited for handling applications that have significant interprocess synchronization and communication requirements. However, computationally intensive applications that can partition their workload into small, relatively independent subproblems, are good candidates to be implemented as a distributed system. One such application is a tool that we developed to recover the high-level subsystem structure of a software system directly from its source code.

More than ever, software maintainers are looking for tools to help them understand the structure of large and complex systems. One of the reasons that the structure of these systems is difficult to understand is the overwhelming number of modules, classes and interrelationships that exist between them.

Ideally, system maintainers have access to accurate requirements and design documentation that describes the software structure, the architecture and the design rationale. Unfortunately, developers often find that no accurate design documentation exists, and that the original developers of the system are not available for consultation. Without automated assistance, developers often modify the source code without a thorough understanding of how their modifications affect the overall system structure. Over time, this ad hoc approach to maintenance results in the deterioration of the system structure.

To address some of the above mentioned problems we developed a tool, called Bunch, which automatically decomposes a software system into meaningful subsystems. Subsystems facilitate program understanding by grouping together (clustering) related source level components. Furthermore, the subsystems themselves can be clustered into even higher level structures, resulting in a hierarchy of subsystems. Thus, developers can study the organization of a system at various levels of detail by navigating through the subsystem hierarchy.

Figure 1 shows the architecture of our automatic software modularization environment. The first step

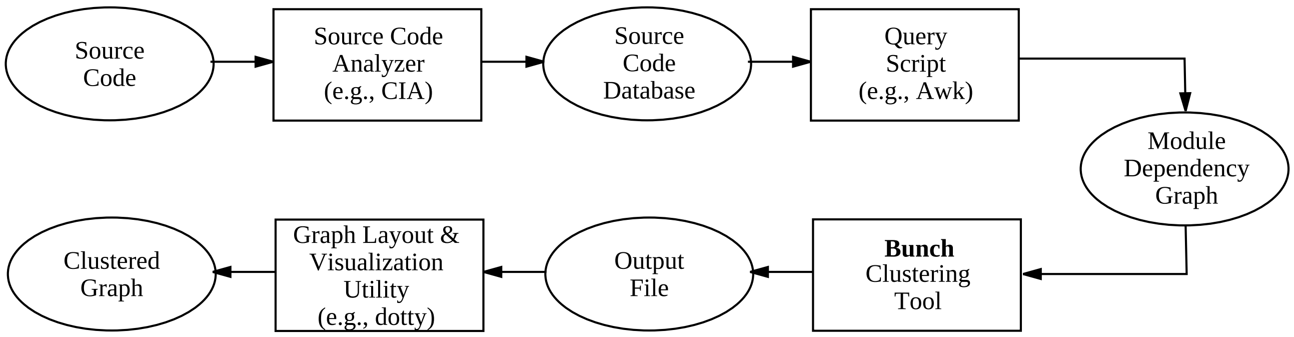


Figure 1. **Automatic Software Modularization Environment**

in our process involves the use of source code analysis tools to determine the *Module Dependency Graph* (*MDG*) of a software system. The *MDG* is a generic, language independent representation of the structure of the system's source code components. This representation includes all of the modules (classes) in the system and the set of dependencies that exist between the modules (classes). The second step in our process uses the Bunch tool to partition the *MDG* into a set of non-overlapping clusters (*e.g.,* subsystems). The third step involves the use of a graph visualization tool to show the partitioned *MDG*.

The algorithms used by the original versions of Bunch are computationally intensive. We found that a drastic improvement in Bunch's performance would be necessary in order to cluster very large systems. To achieve this goal, we first optimized our tool's internal algorithms. We then reengineered the Bunch design to use distributed techniques. Bunch still allows users to cluster systems on a single machine, which is recommended for small and intermediate-sized systems (up to 300 modules or roughly 150,000 lines of code). The remainder of this paper focuses on the distributed aspects of Bunch, as the original version of the tool has been documented elsewhere [10, 17, 18].

The structure of the remainder of this paper is as follows: Section 2 provides a brief overview of the Bunch tool. Section 3 describes the distributed implementation of Bunch. Section 4 presents a case study where we ran clustering experiments on several systems to show how performance increases when using the distributed version of Bunch on large systems. Section 5 presents an overview of related research in the area of software clustering and distributed computing. We conclude with a discussion on the research contributions of our work, and an outline of our future plans to extend Bunch.

2. Automatic Software Modularization with Bunch

In other papers [10, 17, 18] we describe our automatic software modularization technique which is based on quantifying the quality of a partition of a module dependency graph (*MDG*) formally. The $MDG = (M, R)$ is a graph where M is the set of named modules in the software system, and $R \subseteq M \times M$ is a set of ordered pairs of the form $\langle u, v\rangle$ which represents the source-level relationships that exist between module u and module v. The *MDG* for a system can be constructed automatically by using source code analysis tools such as CIA [6] for C, Acacia [7] for C++, and Chava [14] for Java. An example *MDG* consisting of 8 modules is shown on the left side of Figure 2.

Once the *MDG* of a software system is determined, we search for a partition of the *MDG* graph where coupling (*i.e.,* connections between the components of two distinct clusters) is low and cohesion (*i.e.,* connections between the components of the same cluster) is high. We accomplish this task by treating clustering as an optimization problem where the goal is to maximize the value of an objective function that is based on a formal characterization of the trade-off between coupling and cohesion. We refer to our objective function as the *Modularization Quality* (*MQ*) of a partition of an *MDG*. The *MQ* measurement is formally described next.

2.1. Quantifying the Quality of a Modularization

Below we illustrate our modularization quality (*MQ*) measurement. This measurement is used as the objective function of our clustering algorithm and represents the "quality" of a system decomposition. The *MQ* measurement adheres to our fundamental assumption that well-designed software systems are organized into cohesive clusters that are loosely interconnected.

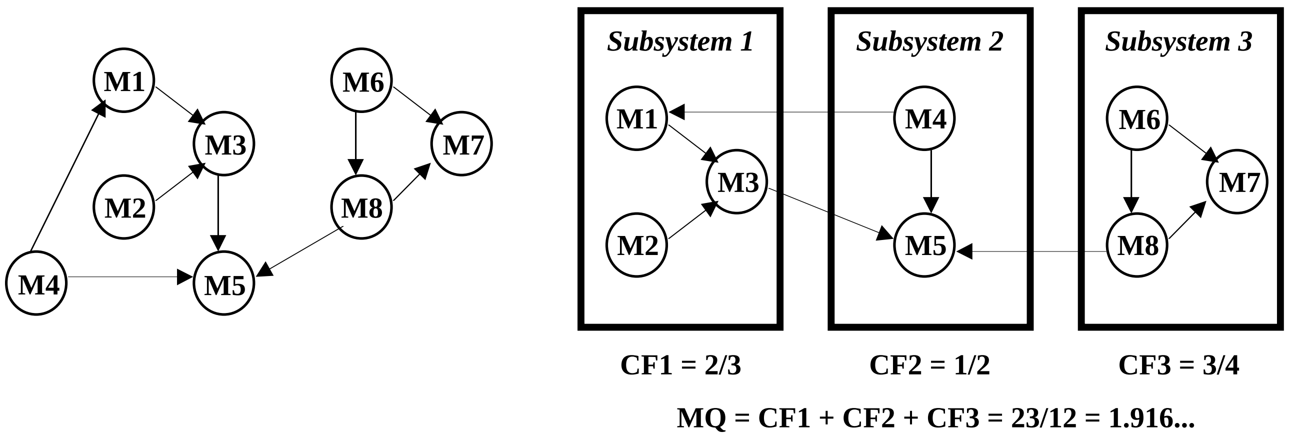


Figure 2. Modularization Quality Example

The MQ expression is designed to reward the creation of highly cohesive clusters while penalizing excessive inter-cluster coupling.

$$MQ = \sum_{i=1}^{k} CF_i \qquad CF_i = \begin{cases} 0 & \mu_i = 0 \\ \frac{\mu_i}{\mu_i + \varepsilon_i} & otherwise \end{cases}$$

The MQ for an MDG partitioned into k clusters is calculated by summing the *Cluster Factor* (CF) for each cluster in the partitioned MDG. The Cluster Factor, CF_i, for cluster i is defined as a normalized ratio between the total number of internal edges and the total number of external edges that originate in cluster i and terminate in other clusters. We refer to the internal edges of a cluster as intra-edges (μ_i) and the edges between two distinct clusters as inter-edges (ε_i).

Figure 2 illustrates an example MQ calculation for an MDG consisting of 8 components that are partitioned into 3 subsystems. The value of MQ is approximately 1.9167, which is the result of summing the Cluster Factor for each of the three subsystems. Each CF measurement is between 0 (no internal edges in the subsystem) and 1 (no edges exiting the subsystem). The larger the value of MQ, the better the partition. For example, the CF for Subsystem 3 is 0.75 because there are 3 intra-edges, and 1 inter-edge. Applying these values to the expression for CF results in $CF_3 = 3/4$.

2.2. Bunch's Clustering Algorithms

One way to find the best partition of an MDG would be to perform an exhaustive search through all of the valid partitions, and select the one with the largest MQ value. However, this is often impossible because the number of ways the MDG can be partitioned grows exponentially with respect to the number of its nodes(modules) [18]. Because discovering the optimal partition of a MDG is only feasible for small software systems (*e.g.,* fewer then 15 modules), we direct our attention, instead, to using search algorithms that are capable of discovering acceptable suboptimal results quickly. The suboptimal search strategies that we have investigated and implemented are based on hill-climbing [18] and genetic [10] algorithms.

We next describe the search algorithms that are supported by Bunch in order to find a partition of the MDG that maximizes MQ.

1. **SAHC:** The Steepest Ascent Hill Climbing algorithm is based on traditional hill climbing techniques. Given an initial partition of the MDG, the goal of this algorithm is to create a new partition with a higher MQ value from the current partition. Each iteration of the algorithm attempts to improve MQ by finding a *maximum neighboring partition* (MNP) of the current partition. We define NP to be a neighbor of partition P if NP is exactly the same as P except that a single node in P is in a different cluster in NP. The MNP is determined by examining all of the NPs of P and selecting the one that has the largest MQ. The SAHC algorithm converges when it is unable to find a MNP partition with a larger MQ than the current partition.

2. **NAHC:** The Next Ascent Hill Climbing algorithm is another hill climbing algorithm that is similar, but often faster, then its SAHC counterpart. The NAHC algorithm differs from SAHC in how it systematically improves the partitions of the MDG. NAHC is based on finding a *better neighboring partition* (BNP) of the current partition. A BNP of the current partition P is found by enumerating through the NPs of P randomly

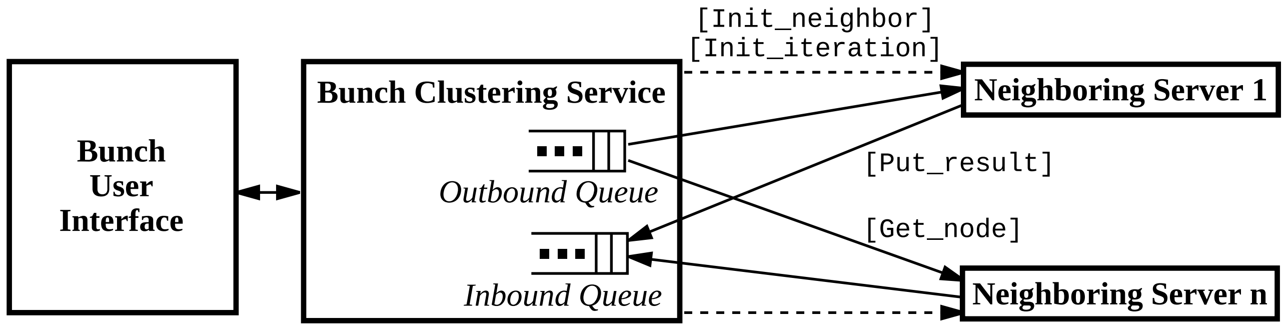


Figure 3. Distributed Bunch Architecture

until an *NP* is found with a larger *MQ* than *P*. The NAHC algorithm converges when no *BNP* of *P* can be found with a larger *MQ*.

3. **GA:** This algorithm uses operators such as selection, crossover, and mutation to find a good partition of the *MDG*. This technique is especially good at finding solutions quickly, but the quality of the results in NAHC and SAHC is typically better. Additional presentation of our Genetic Algorithm approach is described elsewhere [10].

4. **Exhaustive Search:** This algorithm is only useful for very small *MDGs* because it enumerates through every possible partition of the *MDG* and chooses the best one. We have found this technique useful for testing purposes.

We have applied our algorithms to many systems successfully. However, for very big systems, the time needed before the algorithms converge to a good solution tends to be large because the computation requirements grow non-linearly with respect to the number of nodes in the *MDG*. To improve the performance of our tool for such systems, we extended two of our search algorithms (NAHC, SAHC) to utilize distributed techniques.

3. Distributed Bunch

The architecture of the distributed implementation of Bunch is shown in Figure 3. The Bunch User Interface (BUI) is a thin-client application that enables a user to specify clustering and distributed environment parameters, initiate the distributed clustering process, and present results to the user. The Bunch Clustering Service (BCS) is responsible for managing the distributed clustering process by providing facilities to maximize the utilization of the various Neighboring Servers (NS) in the environment. Each neighboring server is initialized with a copy of the *MDG* and the user specified search algorithm (*e.g.,* NAHC, SAHC). Once initialized, the NS program requests work from the BCS and applies one of the neighboring partition strategies (described in Section 2.2) in an attempt to find an "improved" partition of the *MDG*.

3.1. Neighboring Partitions

Recall that the SAHC and NAHC algorithms use hill-climbing techniques to drive Bunch's automatic clustering engine. Both of these algorithms are based on a technique that systematically rearranges nodes in a partitioned *MDG* to improve the *MQ*. This task is accomplished by generating a set of *neighboring partitions* (*NP*) for a given partition (*P*) of the *MDG*. For each *NP* we measure the *MQ*. Our goal is to find *NPs* such that $MQ(NP) > MQ(P)$.

Given a partitioned *MDG*, partition *NP* is defined to be a neighbor of partition *P*, if and only if exactly one node in a cluster in *P* is in a different cluster in partition *NP*. All other nodes in *NP* are in exactly the same cluster in partition *P*. Figure 4 shows all of the neighboring partitions of an *MDG* partitioned into 3 clusters.

An interesting observation can be made about our neighboring partition strategy. Consider an *MDG* partitioned into k clusters, and a particular node from the *MDG*. The identified node can be relocated exactly k times to produce k distinct *NPs* of the partitioned *MDG*. Because each node in the *MDG* can be considered independently, this activity can be conducted in parallel. So, theoretically we can improve our algorithm by a factor of N if we have N processors. In the next section we describe how we used this neighboring partition property in the design of the distributed version of Bunch.

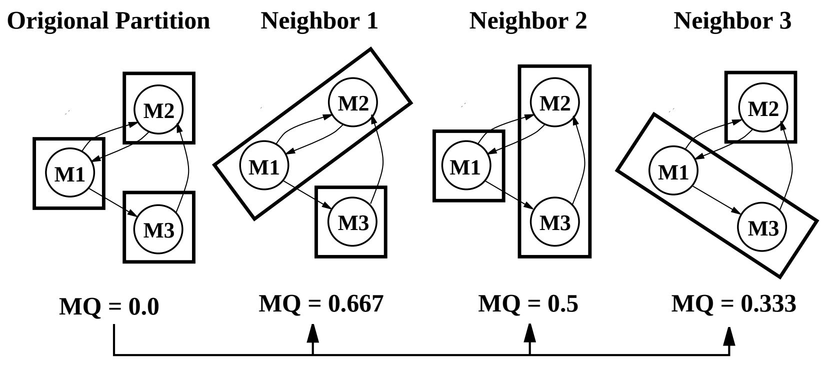


Figure 4. Neighboring Partitions

3.2. Distributed Clustering

The distributed version of Bunch was designed so that the BUI, BCS, and NS processes can run on the same or different computers. All three programs communicate with each other using 4 standard messages, and common data structures to represent the *MDG* and a partition of an *MDG*. In Table 1 we show our four standard messages, along with the associated parameters for each message type. In Figure 5, we illustrate the *MDG* and cluster vector data structures that correspond to the example presented in Figure 2. The cluster vector, which represents a partitioned *MDG*, encodes the cluster number for each node in the *MDG*. For example, $CV[4] = 2$ means that node $M4$ is in cluster 2.

Message	Parameters
`Init_neighbor`	MDG, SearchAlgorithm
`Init_iteration`	ClusterVector
`Get_node`	Node
`Put_result`	Node, MQ, ClusterVector

Table 1. Distributed Bunch Message Types

In the first step of the distributed clustering process, the BUI program creates an instance of the BCS and sends all user-supplied parameters to the BCS. The BCS then initializes all of the NS processes by sending the `Init_neighbor` message to each NS process. The `Init_neighbor` message contains an adjacency list representation of the *MDG* and the user-selected search algorithm (*e.g.,* NAHC, SAHC). These values are cached by each NS for the duration of the clustering process. In the final initialization step, the BCS places each node of the *MDG* into the outbound queue.

Once the BCS and all NS processes are initialized, the BCS generates a random partition of the *MDG*. The random partition, which represents the current

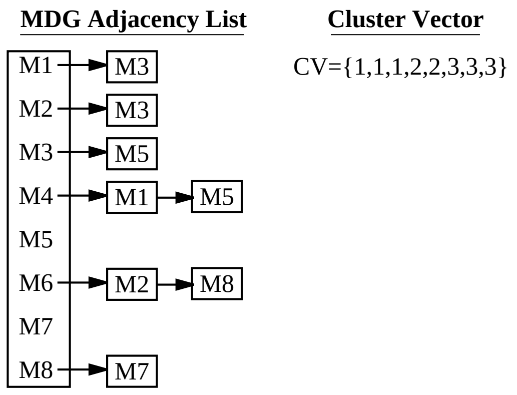


Figure 5. Bunch Data Structures

partition being managed by the BCS, is encoded into a cluster vector and broadcast to each NS using the `Init_iteration` message.

Upon receipt of the `Init_iteration` message, each NS process sends the `Get_node` message to the BCS. In response to this message, the BCS removes the next node from the outbound queue, and returns the node to the requesting NS. If the outbound queue is empty, the BCS generates an exception that is handled appropriately by the NS.

In Section 3.1 we described how our neighboring partition strategy can be used to generate a set of distinct neighbors for a particular node in a partitioned *MDG*. The NS process uses this strategy, on its assigned node, in an attempt to find a better neighboring partition of the clustered *MDG*. The NS process generates the set of neighboring partitions by moving its assigned node to each cluster in the current partition of the *MDG* (stored in the cluster vector). For each move the *MQ* is measured. If the user-selected algorithm is NAHC, the NS stops after it finds the first neighboring partition with a higher *MQ*. If the user-selected algorithm is

SAHC, the NS examines all neighboring partitions to obtain the partition with the largest *MQ*. If the NS cannot find a neighbor with a larger *MQ*, the cluster vector that it received as input represents the best known partition of the MDG.

After the NS has determined the best neighbor for its assigned node of the current partition, it generates and sends a `Put_result` message to the BCS. The `Put_result` message contains the node that the NS received as input, the best discovered neighboring partition (encoded into a cluster vector data structure), and the *MQ* value of the partition. When the BCS receives the `Put_result` message it generates a `(node, ClusterVector, MQ)` tuple from the arguments of the message, and places this tuple into the inbound queue. The NS repeats this process, by sending a `Get_node` message to the BCS, until the outbound queue in the BCS is empty.

Eventually, the outbound queue will be empty, and a tuple for each node in the *MDG* will be in the inbound queue. When the BCS discovers this condition it locates the tuple with the largest *MQ* value in the inbound queue. The cluster vector in this tuple represents the best neighboring partition of the current partition. The current partition managed by the BCS is replaced by the best neighboring partition if it has a higher *MQ* value.

The BCS then reinitializes the outbound queue with all of the nodes from the *MDG* and broadcasts the new current partition (cluster vector) to the NS processes using the `Init_interation` message. This activity is repeated, using the best partition found at each iteration as input to the NS processes, until no neighboring partition can be found with a higher *MQ*.

The NS program supports our NAHC and SAHC hill climbing algorithms. These algorithms are initialized with a random partition and converge to a local maximum. Unfortunately, not all randomly generated initial partitions of the *MDG* produce an acceptable sub-optimal result. We address this problem by creating an initial *population* (*i.e.,* collection) of random partitions. The population size is a configurable parameter that is set by the user in the BUI. The BCS runs one experiment, for each of the random partitions in the population, and picks the experiment that results in the largest *MQ* as the sub-optimal solution. As the size of the population increases, the probability of finding a good sub-optimal solution also increases.

The BCS maintains the best known partition of the *MDG* by comparing the results returned from the neighboring servers for each member of the population. Once the clustering activity finishes, the overall best partition is returned from the BCS to the BUI program. This partition is converted to a representation that can be visualized by a Bunch supported viewer. Bunch currently supports the dotty [21] and Tom Sawyer [23] visualization tools. Optionally, Bunch can direct the dotty tool to create a GIF or postscript file representation of the partitioned *MDG*.

4. Case Study

In this section we present a case study to demonstrate the performance of Distributed Bunch. We used a test environment consisting of 9 computers (with 12 processors) connected to a 100 Mbit Ethernet network. Our desire was to test the distributed implementation of Bunch using a collection of networked workstations. The processor descriptions for each of these computers are shown in Table 2.

System Name	**Processor Speed**	**Processor Description**
Coordinator	450 Mhz.	Pentium II (Windows2000)
Neighbor 1	Dual 366 Mhz.	Ultra Sparc (Solaris)
Neighbor 2	Dual 366 Mhz.	Ultra Sparc (Solaris)
Neighbor 3	Dual 366 Mhz.	Ultra Sparc (Solaris)
Neighbor 4	450 Mhz.	Pentium II (Linux)
Neighbor 5	400 Mhz.	Pentium II (WindowsNT 4.0)
Neighbor 6	233 Mhz.	Pentium II (Windows98)
Neighbor 7	266 Mhz.	Pentium II (WindowsNT 4.0)
Neighbor 8	166 Mhz.	Pentium (WindowsNT 4.0)

Table 2. Testing Environment

The applications selected for the case study were chosen to demonstrate the scalability of Bunch by using many common systems of different sizes. A brief description of the applications we used in this case study is presented in Table 3. Furthermore, the workstations used in our tests varied in processing capacity. Thus, our results are not intended to show the speedup due to adding equivalent-sized processors to the distributed environment, but instead, to show the speedup that can be achieved by using a collection of diverse workstations. Such diversity in the processing capability of computers is typical of office and university computer environments.

Application Name	Modules in MDG	Application Description
Compiler	13	Turing language compiler
Ispell	24	Unix Spell Checker
Bison	37	Parser Generator
Grappa	86	Graph Visualization and Drawing Tool
Incl	174	Subsystem from a Source Code Analysis System
Perspectives	392	Office Application. Includes drawing, spreadsheet, text editor and e-mail components.
Proprietary Compiler	939	A proprietary industrial strength compiler.

Table 3. Application Descriptions

We conducted 2 tests for each of the sample systems described in Table 3. The first test shows the non-distributed performance of Bunch. For this test we ran the BUI, BCS, and NS programs on the Coordinator computer. The second test was executed using a distributed environment with 12 processors. The distributed testing configuration consisted of the BUI and BCS programs running on the Coordinator computer and one instance of the NS program for each processor on the neighboring computers. Thus, for our dual processor systems we ran two copies of the NS program.

4.1. Case Study Results

In Table 4 we present the results of the case study. We show the amount of time needed to cluster each application, along with the speedup associated with using multiple distributed processors. Speedup [12] is defined as t_1/t_n where t_1 is the time needed to cluster the system with 1 processor and t_n is the time needed to cluster the system using n processors. Figures 6 and 7 illustrate the results of the case study graphically. We held the clustering engine parameters constant for all tests.

The results indicate that the distributed environment improves the speedup for clustering the incl, Perspectives, and the Proprietary Compiler systems. For example, with a single processor, Bunch took 153.5 minutes to cluster the Perspectives system. With 12 processors, the Perspectives system was clustered in 26 minutes, resulting in a speedup of 5.9. Performance actually decreased when using the distributed environment to cluster small systems because of network and marshalling overhead.

During the testing process we monitored the CPU utilization of the computers in the distributed environment. For the non-distributed test, the CPU utilization was 100%. For all of the distributed tests, the CPU utilization was approximately 90% for the Coordinator computer, and almost 100% for each computer running the NS program. These results indicate that the distributed version of Bunch maximizes the utilization of the distributed processors. We were not, however, able to saturate the network with a testing environment containing 12 processors. We expect that the utilization of the Neighboring servers will decline as network traffic increases.

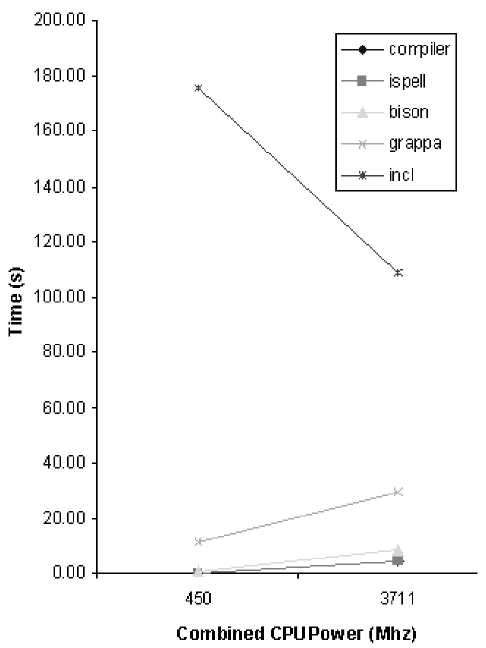
5. Related Work

Using networks of heterogeneous workstations to implement parallel applications has been widely investigated by researchers. Most of this work falls into two broad categories: distributed frameworks (*e.g.,* NOW, CORBA) and applications (*e.g.,* Bunch). From a distributed frameworks perspective, the Berkeley NOW project [2] demonstrates how the power of general-purpose workstations can be harnessed to create supercomputer-scale computing environments. In addition to developing the NOW architecture, the Berkeley team created numerous applications, such as a fast web search engine, to demonstrate the capabilities of their environment. Other popular distributed frameworks include TreadMarks, PVM, and Mozart. The TreadMarks [1] approach is based on a distributed shared memory computing model. PVM [4] is a message passing API for constructing distributed applications on standard Unix workstations. Comparisons of the TreadMarks and PVM environments was conducted by Lu et al. [16]. Motzart [11] is a special-purpose programming language that simplifies the task of constructing distributed parallel applications.

Research on the use of standard networked workstations to create high-performance distributed applications has resulted in several widely used commercial products. Four popular distributed computing middleware frameworks are Microsoft's DCOM [9], OMG's CORBA [8], Java RMI [13], and IBM's MQ Series [19]. The DCOM, Java RMI and CORBA approaches are based on remote procedure calls. The MQ Series product is based on asynchronous message queues.

System	1 Processor	12 Processors	Speedup
compiler	0.05	4.06	0.01
ispell	0.09	4.08	0.02
bison	0.59	8.64	0.07
grappa	11.35	29.26	0.39
incl	175.46	108.63	1.62
Perspectives	9212	1561	5.90
Proprietary	541586	100471	5.39
all times are shown in seconds (wall time)			

Table 4. Case Study Results



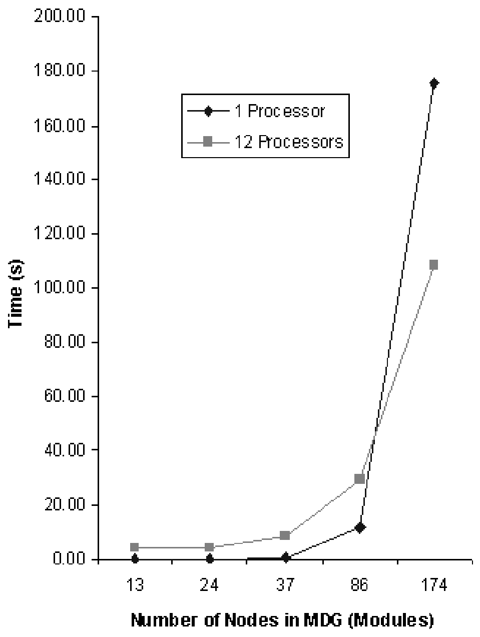
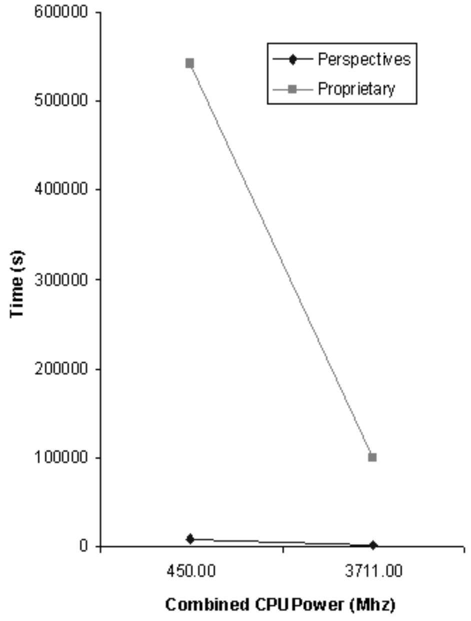


Figure 6. Case Study Results for Small- and Intermediate-Sized Systems

The popularity of using networks of workstations to implement parallel applications is apparent in the diversity of case studies that have been published in this area. Carceroni et al. [5] implemented a probabilistic search engine to support a real-time computer vision problem. Lee et al. [15] describe a case study where they harnessed the power of a distributed computing environment to manage large amounts of medical image data. In this paper we presented the distributed version of our Bunch software clustering tool, which was created using the CORBA environment. This paper also describes a new *MQ* measurement that improves the performance of our original objective function without sacrificing quality.

The original version of Bunch [18] automated the software clustering process by treating clustering as an optimization problem. The next version of Bunch [17] added features to support user-supplied information about the system structure to direct Bunch's automatic clustering process. Other well known software modularization tools are Rigi and ARCH. The ARCH tool [22] applies the concepts of high cohesion and low coupling to the software clustering problem. The Rigi [20] tool implements several semi-automatic clustering heuristics that assist software practitioners in understanding the structure of systems. Some clustering techniques do not make direct use of the dependencies in the system's source code. For example, Anquetil and Lethbridge [3] describe a clustering technique based on using common patterns in file names to group related software components. For an overview of clustering techniques, we recommend a survey paper by Wiggerts [24].



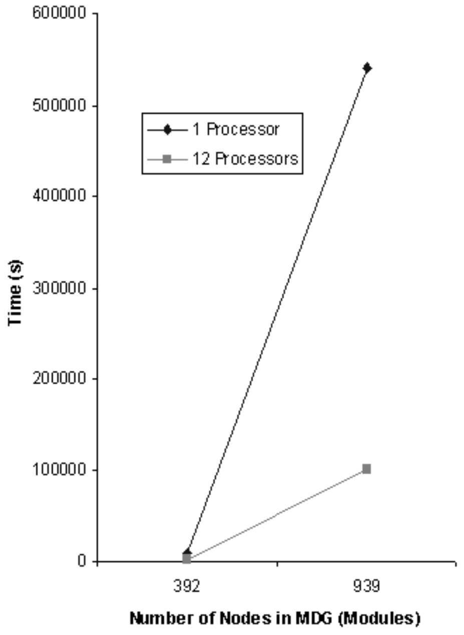


Figure 7. Case Study Results for Large Systems

6. Conclusions and Future Work

This paper describes the latest extensions made to Bunch to improve its performance. Specifically, we introduced a new objective function that can be computed much faster than our original *MQ* function [18]. This paper also shows how we modified our search algorithms so that they can be executed in parallel over a network of workstations.

While the results of our case study were encouraging, we feel that further improvement is possible. Specifically, one of the fundamental design objectives of our technique was to maximize the utilization of all distributed processors. We feel that we were successful, as the CPU utilization of the distributed processors was 100% for the duration of the clustering experiments. However, when we instrumented our source code, we observed that the neighboring server processes were doing a significant amount of distributed I/O (*i.e.,* sending the `Get_node` and `Put_result` messages). Thus, as a future enhancement we would like to integrate an adaptive load-balancing algorithm into the distributed version of Bunch. This capability would enable our technique to provide more work (*i.e.,* the `Get_node` message would return a collection of nodes) to faster processors dynamically.

The new objective function introduced in this paper can be evaluated significantly faster than our original *MQ* function. Although this objective function can be calculated much faster, it still requires examination of all nodes and edges in the *MDG*. Since each neighboring move made by Bunch (discussed in Section 2.2) impacts at most two clusters, we plan to investigate if further performance improvements can be made by updating the *MQ* value incrementally. This enhancement would only require examining the edges in the *MDG* that are connected to the node that was moved to a neighboring cluster.

We are also working on providing the users of our tools with a web-based clustering service. This capability would enable users to submit an *MDG* graph over the Internet (using a Web Browser) to the distributed Bunch Clustering Service. The service would partition the *MDG* and display the result within the user's Web Browser.

7. Obtaining Bunch

Interested readers can obtain a copy of Bunch, which is developed in Java, from the Drexel University Software Engineering Group's (SERG) web page at: http://serg.mcs.drexel.edu. In addition to downloading a copy of our software, the SERG web page provides online user and programmer documentation for our tool, as well as URL links to other tools such as Acacia, dot and dotty.

The SERG web page also includes documentation about the Bunch API so that programmers can integrate clustering capabilities into their own applications.

8. Acknowledgements

This research is sponsored by two grants from the National Science Foundation (NSF), a CAREER Award under grant CCR-9733569, and an instrumentation award under grant CISE-9986105. Additional support was provided by the research laboratories of AT&T, and Sun Mircosystems.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, U.S. Government, AT&T, or Sun Microsystems.

References

[1] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, Feb. 1996.

[2] T. Anderson, D. Culler, and D. Patterson. A Case for Networks of Workstations: NOW. *IEEE Micro*, Feb. 1995.

[3] N. Anquetil and T. Lethbridge. Extracting concepts from file names; a new file clustering criterion. In *Proc. 20th Intl. Conf. Software Engineering*, May 1998.

[4] A. Beguelin, J. Dongarra, A. Geist, B. Manchek, and V. Sunderam. Recent Enhancements to PVM. *International Journal for Supercomputer Applications*, 9(2), 1995.

[5] R. L. Carceroni, W. M. Jr., R. Stets, and S. Dwarkadas. Evaluating the trade-offs in the parallelization of probabilistic search algorithms. In *Proceedings of the 9th Brazilian Symposium on Computer Architecture and High Performance Processing*, Oct. 1997.

[6] Y. Chen. Reverse engineering. In B. Krishnamurthy, editor, *Practical Reusable UNIX Software*, chapter 6, pages 177–208. John Wiley & Sons, New York, 1995.

[7] Y. Chen, E. Gansner, and E. Koutsofios. A C++ Data Model Supporting Reachability Analysis and Dead Code Detection. In *Proc. 6th European Software Engineering Conference and 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Sept. 1997.

[8] CORBA. The Object Management Group (OMG). http://www.omg.org/corba.

[9] DCOM. Microsoft Corporation. http://www.microsoft.com/com.

[10] D. Doval, S. Mancoridis, and B. Mitchell. Automatic clustering of software systems using a genetic algorithm. In *Proceedings of Software Technology and Engineering Practice*, 1999.

[11] S. Haridi, P. V. Roy, P. Brand, and C. Schulte. Programming Languages for Distributed Applications. *New Generation Computing*, 16(3), 1998.

[12] K. Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* McGraw-Hill, 1993.

[13] Java RMI. Sun Microsystems Corporation. http://www.javasoft.com.

[14] J. Korn, Y. Chen, and E. Koutsofios. Chava: Reverse engineering and tracking of java applets. In *Proc. Working Conference on Reverse Engineering*, 1999.

[15] J. Lee, B. Tierney, and W. Johnston. Data intensive distributed computing: A medical application example. In *Proceedings of the 7th International Conference on High Performance Computing and Networking*, Apr. 1999.

[16] H. Lu, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Quantifying the Performance Differ-ences between PVM and TreadMarks. *Journal of Parallel and Distributed Computing*, June 1997.

[17] S. Mancoridis, B. Mitchell, Y. Chen, and E. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings of International Conference of Software Maintenance*, Aug. 1999.

[18] S. Mancoridis, B. Mitchell, C. Rorres, Y. Chen, and E. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Proc. 6th Intl. Workshop on Program Comprehension*, 1998.

[19] MQ Series. IBM Corporation. http://www.software.ibm.com/ts/mqseries/.

[20] H. Müller, M. Orgun, S. Tilley, and J. Uhl. Discovering and reconstructing subsystem structures through reverse engineering. Technical Report DCS-201-IR, Department of Computer Science, University of Victoria, Aug. 1992.

[21] S. North and E. Koutsofios. Applications of graph visualization. In *Proc. Graphics Interface*, 1994.

[22] R. Schwanke. An intelligent tool for re-engineering software modularity. In *Proc. 13th Intl. Conf. Software Engineering*, May 1991.

[23] Tom Sawyer. Graph Drawing and Visualization Tool. http://www.tomsawyer.com.

[24] T. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *Proc. Working Conference on Reverse Engineering*, 1997.