Visualizing Large Communication Graphs

by
Steven S. Rubin

A Thesis
Submitted in partial fulfillment of
the requirements for the Degree of Bachelor of Arts with Honors
in Computer Science

WILLIAMS COLLEGE
Williamstown, Massachusetts
May 22, 2011

Abstract

Communication graphs induced by parallel programs can be overwhelming in size, but are highly structured. For example, trees, grids, and n-cubes are common. In order to understand program behavior, we seek informative graph visualizations that reduce visual complexity. We leverage the concept of ellipses ("..."), an intuitive symbol that abstracts redundant structure.

Using graph grammars coupled with specific procedural methods, we create large, structured graphs of parallel program communication. These graphs that have practical realizations in two dimensions can be visually simplified using an algorithm that finds contextual balls around important nodes. Graphs that are harder to visualize in two-space are more difficult to simplify. This work presents metrics and layout techniques that may aid in the visualization of such graphs. We evaluate these metrics by developing an ideal for want to see in abstracted visualizations of large, structured graphs. The metrics and layout techniques are implemented in `Gephi`, an open-source large graph visualization tool.

Our large graph visualizations are important tools for helping programmers understand interprocess communications of their parallel programs. This work establishes some prototypes for visualization that in the future may harnessed in the design of effective parallel programming environments.

Acknowledgements

I'd like to thank Duane, for coming up with such a fundamentally interesting topic, my parents, for raising me to appreciate the value of a good *thought*, and my friends, for preventing me from getting completely buried in the thesis workload.

Visualizing Large Communication Graphs

Chapter 1

Introduction

1.1 A shift to parallelism

As Moore's Law slows and CPU designers shift focus to optimizing multi-core processors, programmers will be forced to adopt the paradigm of parallelism. When considering parallel programming and parallel algorithms, it becomes clear that certain symmetries in communication arise. For example, in an algorithm that defines an operation to be performed in parallel on data, a master-to-many-slaves tree communication structure is formed. If the slave nodes similarly spawned processes in parallel, the resulting nodes would further add to the tree-like communication structure. Other graph-theoretic structures tend to emerge from parallel algorithms as well [41]. For example, simulation of a heat equation might induce a communication structure whose graph is a grid-like mesh with bidirectional edges. Another highly practical graph structure is the hypercube [43].

Understanding the structure of communication in a parallel program is vital to debugging. Small trees, grids, and hypercubes can be visualized without any loss of detail. However, in massively parallel systems, the communication structures of parallel programs quickly become inefficient to visualize in full detail. A natural step toward solving this problem is to find methods of abstracting the communication structure's visualization.

1.2 Working toward a solution for the visualization problem

Past research (clustering, focus+context [23, 31, 40]) has addressed three primary methods of reducing visual complexity: ghosting, hiding, and grouping. Ghosting deemphasizes nodes, hiding conceals nodes, and grouping merges nodes together into pseudo-nodes. These techniques, paired with a function to calculate a node metric to cluster similar nodes together, can lead to aesthetically pleasing visualizations of otherwise unwieldy graphs [17, 44].

Current visualization techniques of general graphs do not take into account the symmetries that are specific to parallel algorithm design [8, 21]. The parallel programmer wants to do something in the small which will eventually grow to fit a given machine. Thus, as parallel programs are grown recursively and symmetrically, we believe that the ideal visualizations of the communication graphs would leverage these features. If the communication between process 0 and processes $\{1 \dots\}$ are all the same by algorithmic definition, then showing all of these communications in our visualization might be superfluous. Thus a natural goal for these visualizations would be to show only a limited number of these repeated substructures. A programmer who is using parallel algorithms should have an intuition for the desired interprocess communication structure of a program. Therefore we have incentive to give the programmer natural tools to visualize and debug the resulting communication graph.

While clustering may seem like a natural fit for this problem, it will remove the sense of repetition that, while redundant in full detail, is critical to understanding the symmetry inherent in the induced communication graph. Luckily, we already have an indicator for repetition: the ellipsis $(\dots)$. Take, for example, the range $1, 2, 4, \dots, 2^n$. The ellipsis here indicates that a pattern is present, and while the numbers excluded by the ellipsis are not identical, the pattern is easy to deduce. Adding ellipses to visualizations fits our intuition of the induced communication structures of parallel programs.

Figure 1.1 shows a hypothetical communication structure, where each node represents a process. The ellipses play an important intuitive role in this graph which could contain several thousand nodes. In parallel programming, the number of processes will likely be a function of the number of available processors, but the gist of the computation can be seen in a select group of nodes. By adding ellipses, we strive

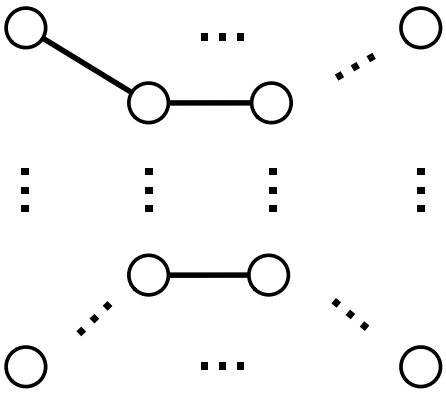


Figure 1.1: A hypothetical "ellipsized" communication graph

to identify relevant groupings of nodes.

It seems likely that, because we lack an absolute metric of the utility of a visualization, the effectiveness of a communication graph would be intuitively defined, and not mathematically rigorous. Thus, instead of defining graph reductions in terms of absolutes, this work will strive to create several simplifying heuristics that interact in informative ways. Applying a given reduction scheme to a graph will reveal some information about the inherent nature of the graph or of the relationship between distinct sections of the graph. Compositions of these simplifications will likely present details of a given graph with adjustable granularity.

1.3 First steps

This thesis investigates the visualization of large communication structures that result from massive parallelism. It makes use of graph grammar formalisms and a notion of abstracted views and layout of graphs. We believe these first steps are important to building parallel programming environments that make visualization an integral feature.

Chapter 2 discusses the background of large graph visualization and parallel debuggers. Chapter 3 uses the process of parallel program design to set our visualization goals. In the following chapter, we establish formalisms for growing graphs, which we

implement in `Java`. Chapter 5 provides a more in-depth background on the crucial concepts of generalized fisheye views and feature classification. The next two chapters discuss our process of visualizing large graphs, and introduce our key concept of the contextual ball. Finding these contextual balls around important nodes provides a rigorous method of deciding which nodes to abstract in any graph that is grown in a grammar-like manner. In our visualizations, we harness the features and the API of `Gephi`, an open-source large graph visualization tool. The final chapter reflects on our progress and outlines potential future research.

Chapter 2

Background

2.1 Graph layout

Graph drawing and graph layout is a thriving field of computer science, boasting several focused journals and an annual conference. The explicit goal of graph drawing is to display general graphs in aesthetically pleasing, informative, typically two-dimensional (though possibly 3-D) representations. Graph drawing's pivotal question is thus, "What is the ideal layout aesthetic for the given application?"

2.1.1 Small graphs

"Graph Drawing" [7] by Battista, Eades, Tamassia, and Tollis is the canonical introduction to graph drawing principles and algorithms. Given a sufficiently small graph, this work is likely to give an efficient and aesthetically pleasing way of visualizing it. We will be concerned with graphs are too large for conventional techniques, but we depend on prior work for the presentation of small graphs.

2.1.2 Large graphs

Drawing large graphs poses a new set of challenges. If applying a standard technique, the entire graph would be displayed with each feature having equal weight. The resulting image would be obfuscated by the density of information. Instead, researchers have developed methods of graph visualization that focus on specific features of the

graph while hiding others. The main approach involves clustering and amplifying the importance of different features of the graph.

Clustering

If several vertices in a graph seem to exhibit similar behavior in the context of the graph as a whole, we can abstract away vertices or form an equivalence class with clustering techniques. The general principle is that we need to partition nodes into "clusters." From that point, layout can proceed by letting our clusters be the nodes of a smaller graph with, one assumes, simpler structure. Clusters afford other options, as well. For example, if we iteratively apply a clustering technique to a graph, we can maintain and display a hierarchical clustering [23]. (A similar hierarchical approach has been used in physics as a scale-independent abstraction mechanism [47].) One way to find clusters is to define *node metrics* on a graph, which encode some structural feature of each vertex. Examples of node metrics include the *degree of interest* function and *Strahler numbers*.

Degree of interest

In many applications, one section of the graph may be more important than the rest. For example, in a visualization of a file system, the working directory is the most important feature. Directories that are ancestors and descendants of the working directory may be visible, while distant relatives may be absent entirely. The intuition of this example is that of focus+context: part of the graph is in focus (the working directory) and the visualization provides a context about that focus (some ancestors and descendants, as well as other important parts of the directory tree). This model is captured mathematically by *fisheye views*.

To apply a fisheye view to a graph, we define an *a priori* interest (API) function and a distance metric (D) on vertices [17]. $API(x)$ defines how interesting, useful, or important a vertex is, disregarding any particular focus of the graph. $D(x, y)$ is a metric measuring logical distance between two vertices. These two functions are combined by defining the degree of interest (DOI) of a vertex x in a graph with focus y to be

$$DOI_y(x) = API(x) - D(x, y)$$

The graph-drawing process can then proceed, using this DOI function as a guide. We

could, for example, emphasize vertices with high degrees of interest, or hide vertices with degrees of interest below a threshold. While fisheye views provide a simple framework for showing focus and context, defining API and D for large graphs is often not obvious.

Strahler and flow metrics

Originally conceived for use in hydrology, the *Strahler number* of a node in a tree or directed acyclic graph is a measure of the branching structure at that node. They are only used for trees and DAGs because their definition depends on the structure of the graph being similar to the structure of rivers: in particular, there must be an obvious notion of flow between predecessor and successor [21, 22]. A formula (there are many similar definitions) for the Strahler number on a node in a tree is

$$S(v) = \max(S(k_1), \ldots, S(k_p)) + \begin{cases} p-1 & \text{if all values } S(k_i) \text{ are equal} \\ p-2 & \text{otherwise} \end{cases}$$

where the k_i are the p successors of v. The Strahler number on each leaf is defined as 1. A similar formula can be defined for DAGs. The *flow metric* on a DAG imagines the graph as a series of pipes and quantifies the amount of flow of a liquid at each node [22]. Each source node v has $M(v) = 1$, while for other nodes,

$$M(v) = \sum_j M(a_j)/\text{number of successors of } a_j$$

While both the Strahler and flow metrics are practical metrics on directed acyclic graphs, process communication in parallel programs induces general graphs that lack strong properties. Regardless, the metrics could still be useful if we find ways to form restricted views of full graphs as DAGs, trees, or other graphs with simpler structures.

Skeletons

A skeleton of a graph is a subgraph of important vertices and edges. Given a set of node metrics, we can combine them to form per-node estimates of importance. The resulting function can be used in combination with some threshold to select a skeleton for the graph.

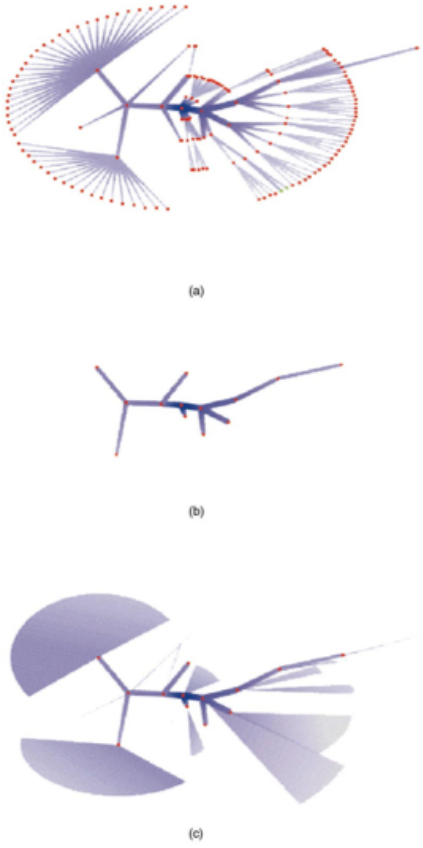


Figure 2.1: Ghosting, hiding, and grouping applied to a skeleton of a tree

Once we have selected our skeleton, and thus identified the most prominent features of the graph, we must address the remaining nodes. Common visualization techniques for these less important nodes are ghosting, hiding, and grouping [23, 31]. If a set of vertices are unimportant, we can *hide* them from our visualization. Likewise, if a set of vertices are of secondary importance, we can *ghost*, or somehow shrink or fade this set in our visualization. Finally, we can represent a set of vertices as a super-node by *grouping* them together. Figure 2.1 from [23] shows representative examples of these visualization techniques on trees.

An issue with these methods of visualization for large graphs is that they depend heavily on abstracting them as trees and DAGs [8, 21, 22]. In doing so, they do not take other structural properties of the graph into account. In general, parallel process communication graphs will contain symmetries, thus an "equivalence class" can likely be found for each node. We strive to find node metrics and other methods of visualization that capture this symmetry and repetition. Alternatively, the above methods can be used on general graphs by first finding a minimum spanning tree and then applying a tree-specific node metric to it. It has been noted that selecting thresholds for combining nodes "without some *a priori* knowledge of the data" is a challenging problem [40]. Fortunately, the nature of parallel programs gives us a wealth of knowledge about the structure of process communication graphs.

Graph grammars

Rather than using a node metric to deduce structural properties of graphs, we can use graph grammars to generate large graphs that have mathematically well-defined structures. The general principle of graph grammars is that we start with a small graph and a set of grammars, or rules, that are applied in succession, thereby growing the graph. Recursively defined graphs have natural links to parallel programming.

Graph rewriting mechanisms have been used to graphically implement parallel programs [5]. Suppose that at each step of the graph rewriting process, we modify a metric of each node. By building a large parallel program in this manner, we hope to obtain a useful node metric for the program's communication graph. This metric could then be used, perhaps in a manner described above, to render larger programs in an understandable way. This is the subject of much of the rest of this work, and is described in detail in Chapter 4.

2.2 Debugging

As multicore systems have begun to dominate the computing landscape, we have been challenged to design debuggers for parallel programs. Improvements in the visualization of large graphs could have a direct impact on the utility of parallel program debuggers; as programs grow from a few processes to thousands or millions, we must find reasonable ways to view or abstract the communication within the

program.

Many parallel program debuggers are post-mortem debuggers, meaning they perform analysis on the trace file from a program's execution. A trace file from a program using MPI [14] (Message Passing Interface, the most common parallel programming API specification) contains temporal information about interprocess communication. The communications in the trace file can be combined to form a process communication graph.

2.2.1 The ATEMPT debugger

The ATEMPT debugger [33] shows the programmer a visualization of the parallel program. A trace file is used as input, so all analysis is done post-mortem. Unlike many parallel debuggers, the ATEMPT debugger displays its analysis in a space-time event graph. Processes are stacked on the space axis, and each process's changing state is shown on the time axis. The graph shows blocking times of different processes, which can illuminate load-balancing issues.

Some features of the debugger are included specifically for applications with many processes. For example, processes that have similar communication patterns can be grouped into a single pseudo-process on the space axis. Symmetries in execution, such as a one-to-many `broadcast` are also abstracted down to a single event in the visualization. While the grouping of processes is theoretically advantageous, the ATEMPT debugger offers no automatic grouping options; the programmer must manually specify sets processes he would like to group. Another problem that this debugger, and parallel debuggers in general face is the "probe effect"—namely, that sufficiently long delays in computation (to gather data for a debugger, in this case) can hide concurrency errors [19].

2.2.2 The DEPICT topological debugger

The DEPICT debugger attempts to capture the important topological structures and symmetries—"pipelines, rings, hypercubes, and trees," for example [28]—that are commonly found in parallel programs. Using post-mortem trace files, the DEPICT tool attempts to identify the process topology of the program and then identifies equivalence classes of processes. The debugger then allows the programmer to view a graphical representation of the topology and equivalence classes. As the programmer

should have an intuition *a priori* about the topology and communication structure in his program, the visualization will immediately bring to light any errors in communication. If a process is not being grouped in the correct equivalence class, an error is present.

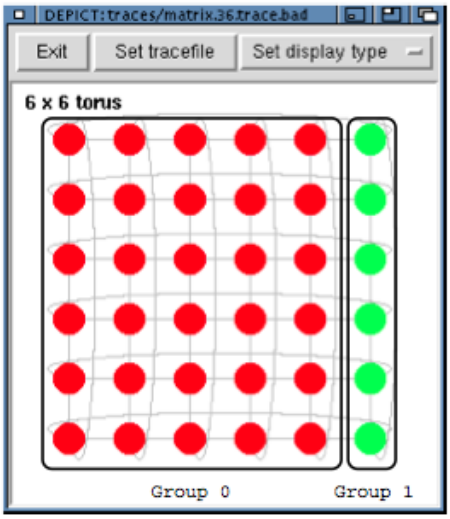


Figure 2.2: Visualization of an incorrect matrix multiply on a torus from DEPICT.

Figure 2.2 from [28] gives an example of a visualization from DEPICT. This is clear and readable, and should be taken as a baseline for our goals in visualization. While DEPICT successfully shows equivalence classes for small numbers of processes, the authors acknowledge that they have not made progress on larger topologies. While debugging on small topologies can be informative, we would ideally have a way to extend these methods to arbitrarily large graphs while maintaining the clarity of these DEPICT visualizations.

2.2.3 Topological debugging by specification

The DEPICT debugger promised methods of identifying topologies, but the implementation itself was limited to two simple topologies—the mesh and the torus. One way we can identify complex process topologies in parallel programs is to have the

programmer give a specification of the topology [29]. A topological specification would provide a template that the debugger could follow when piecing together the communication graph from the trace file.

In addition to defining a specification for the topology, the user is asked to specify "normal" communication patterns within the topology. If these patterns are violated, a debugger alerts the programmer to an error. While this specification-driven model of parallel program debugging is powerful, it requires that the programmer have (a) a firm grasp of exactly what could go wrong in the program and (b) motivation to write the specifications. If a programmer could enumerate all possible errors in a program, he could probably write a program without the errors in the first place. Instead, we hope to find methods that are independent of rigorous specification. We acknowledge that some methods may require that the programmer do some extra work, but full specification-design seems to be a prohibitive upper bound of what the programmer should have to do.

2.3 Graphical pattern recognition and abstraction

The graphics community has also investigated the identification of regular and symmetric structures. We would like to leverage the results of this work, but the problem is complex. Still, the following work exemplifies the approaches we think may be useful in identifying and displaying large graphs with symmetry.

Mehra et al. [35] show a way of identifying abstractions of 3D models of man-made shapes. For example, the Eiffel Tower is identifiable even by a crude rendering of its primary shapes. While the methods in their work are focused primarily on graphics-based techniques for finding smooth approximations of the important shapes of a model, they make an important observation relevant to large graph visualization. Notably, "some objects, perhaps those less familiar to us, have no obvious natural abstraction."

Pauly et al.[38] show a top-down approach to finding structural equivalence classes in 3D models. In finding the building blocks of models, they also identify the mathematical symmetry that appears through rotations, translations, and scaling. To visualize large graphs, we would also like to identify equivalence classes. But, because segments of a graph are not as identifiable as different geometric shapes, we instead approach the problem from the bottom-up.

In work by Agrawala et al. [1] on automating the design of step-by-step assembly instructions, the researchers had to "manually provide the semantic and functional knowledge about the parts." They do conjecture, however, that "it may be possible to automatically group parts that are perceived as roughly symmetric." Any research that identifies symmetric structures would be a boon for the visualization of large, symmetric graphs.

Using this background knowledge, we proceed to investigate why parallel program communication graphs are a natural fit for abstracted visualizations.

Chapter 3

Parallel programs

Parallel algorithms are preferable to sequential algorithms in modern computer architectures because they can harness the presence of multiple cores and processors to execute several computations concurrently. The use of concurrency is, not surprisingly, a new force in program design and, as we will see, places new demands on our ability to understand what our programs ideally and *actually* do.

3.1 Parallel algorithm communication structures

3.1.1 Concurrency seen in communication structure

In order to create successful parallel algorithms, designers have to determine ways to harness the power of concurrency. Naïve translations of sequential algorithms to parallel architectures will not suffice in most cases because of constraints such as data dependencies. In finding ways to work around these data dependencies, algorithm designers will typically manifest concurrency in communication structures. As a simple example, a tree-like communication structure can be used to, in parallel, sum the contents of an array (or perform any "reduction" operation) under a parallel memory architecture in $O(\log n)$ time rather than the $O(n)$ sequential algorithm time complexity. The *shape* of the tree is pivotal to our understanding of the algorithm's potential. The way we view this tree is thus important to establishing the correctness of the algorithm.

3.1.2 The figures used are natural

In designing the communication structure for a parallel algorithm, the designer is advocating a certain way of *thinking* about a problem. Explanations of algorithms are often accompanied by figures that diagram the communication structure. Figures 3.1 through 3.3 show communication structures from parallel algorithms texts.

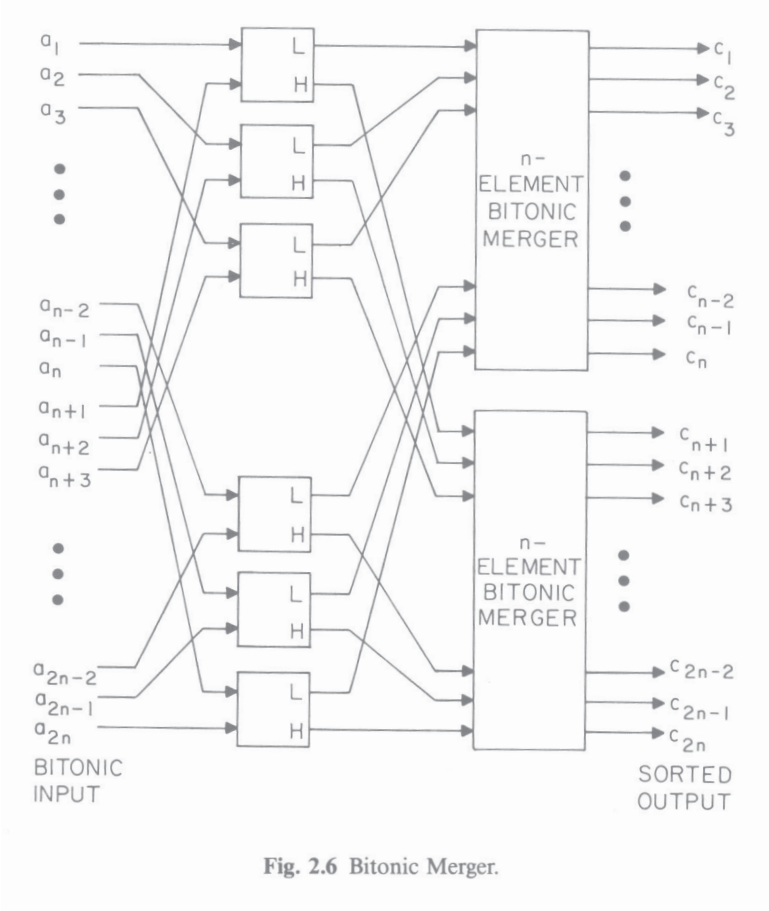


Figure 3.1: From [3], page 32. A bitonic merger for a sorting algorithm, presented with ellipses.

One imagines that the authors of these texts have designed their figures to be ef-

fective. They chose their figures as a *natural* means of communicating the algorithms' functions.

3.2 Algorithm design

3.2.1 A problem of scale

As with sequential algorithms, it is not uncommon for algorithms to be designed, implemented, and tested on smaller-scale versions of their eventual use cases. Once the designers have worked out the bugs in their programs, they will scale their algorithms up to larger sizes. These larger programs are clearly harder to understand, but by designing and testing them in the small, we lose a sense of the bigger, more realistic picture (see Figure 3.2 where the program is visualized in-the-small and -large).

3.2.2 Natural growth and a small viewing window

The symmetries of parallel communication structures are naturally extended as we scale programs up. Because this growth is natural—no new structural features are added as communication extends to more processors—we know that we might still have success debugging the program from a smaller window, as if the program was still small in scale. Not only do we see that we *might* have success debugging for smaller abstractions of large programs, but we argue that we *must* reason about large, complicated programs via a smaller window. Analyzing large communication structures without such a window would lead to confusion, as we cannot reason about structures that we cannot even adequately visualize.

When the program is scaled up, the *code* does not increase in complexity, but the *communication structure* does. This may lead some programmers to assume that, because (1) their code worked on small instances of the program and (2) the code has not changed, then the code will work in the large. That the communication is more complex is, itself, an indicator that there may be boundary cases that go unseen in smaller test cases. Some bugs may only appear in programs in-the-large.

Even given an understanding of the potential issues in parallel algorithm design and parallel programming, effective parallel programming and visualization tools lag behind.

3.2.3 Ellipses in figures

Notice that in the figures we showed from parallel algorithm texts, the authors included several instances of ellipses. As these diagrams were chosen to reflect the clearest understanding of a problem, we can infer that the ellipses are useful and necessary parts of the visualization of communication structures. We want to mimic this behavior and understanding in our visualization techniques.

3.3 Visualization tools

When a parallel program successfully completes execution, it can output a trace file that contains a record of the interprocess communication that occurred. The trace file that is generated is barely human readable, and certainly less-than-useful for debugging. In Appendix A, Figure A.1 shows the `clog` file generated by an MPI program that features only two instances of IPC and was run with only two processes. The log file makes no judgements on the importance of information, nor does it give an indication of the communication structure, the back-and-forth of the program.

The curators of the `MPI` libraries at the Argonne National Laboratory released a post-mortem trace file viewer called `Jumpshot` that visualizes programs as in Figure 3.4 [11]. While this visualization may make sense to the experienced programmer, debugging still appears difficult. If a parallel program is visualized with two dimensions and one of the dimensions is devoted to time, as in `Jumpshot`, then the remaining dimension is highly restricted. `Jumpshot` displays the processes of the program in sequential order with no sense of communication topology. The lines that connect processes, which represent IPC, do not intuitively reveal the structure of the program. While `Jumpshot` gives a more complete picture of the program than the textual log file does, it still fails to analyze the log for higher-level structure and form. `Jumpshot` can give insight into message passing, but does not adequately characterize the big picture. Figure A.2 in Appendix A shows the `Jumpshot` visualization that corresponds to the `clog` file from Figure A.1.

In order to proceed with visualizing this sort of data, we have to decide what we want to see that we are not getting from visualizations like `Jumpshot`. Parallel programs have natural topologies, and we certainly want to visualize these topologies from a program log file. As parallel programs are run on a set of processors that

have a certain topology, it would be helpful to see the program's topology for the sake of comparison. The programmer should have a means for learning whether the program's topology has a clean embedding in the physical processor topology. A display of the program topology should give the programmer an obvious indication of whether his intentions match the reality of his code. Asymmetries in the visualization will probably be indicators of this type of error. Finally, a visualization of a parallel program should not blindly show all the information it has available to it; that is, in displaying a process communication graph, it should not display a node for every process if the graph is large, and therefore less likely to be human-readable.

To address these goals, we hope to show visualizations of large process communication graphs that abstract away unnecessary data. The segments of the graph that are displayed should be the most important with respect to the goal of understanding the communication structure of the program. By omitting the "time" dimension from the 2-dimensional visualization, we will be able to view more complex patterns in process structure. If necessary, "time" can then be added later via a third dimension (with, for example, a scrollbar or scrubber).

As we have mentioned in Section 2.2, prior researchers have attempted to show more meaningful visualizations of parallel programs. The DEPICT topological debugger [28] is perhaps the clearest example of using communication structures to find and fix bugs. Unfortunately, this debugger is limited to small communication structures.

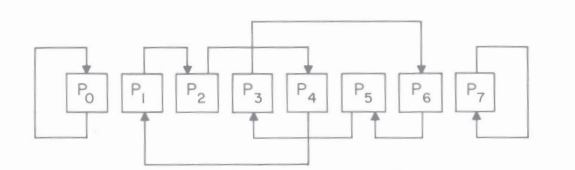


Fig. 4.2 Perfect-shuffle interconnection.

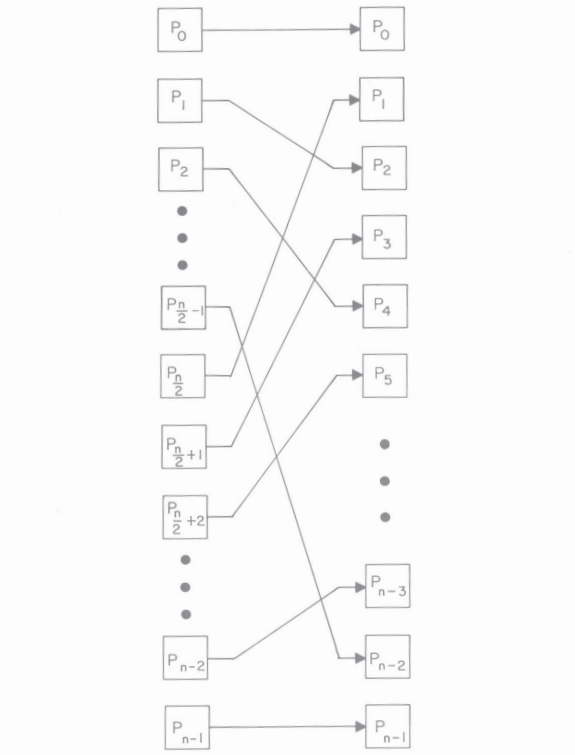


Fig. 4.3 Perfect-shuffle interconnection viewed as a mapping.

Figure 3.2: From [3], page 62. Note the visualization of the program in-the-small and in-the-large.

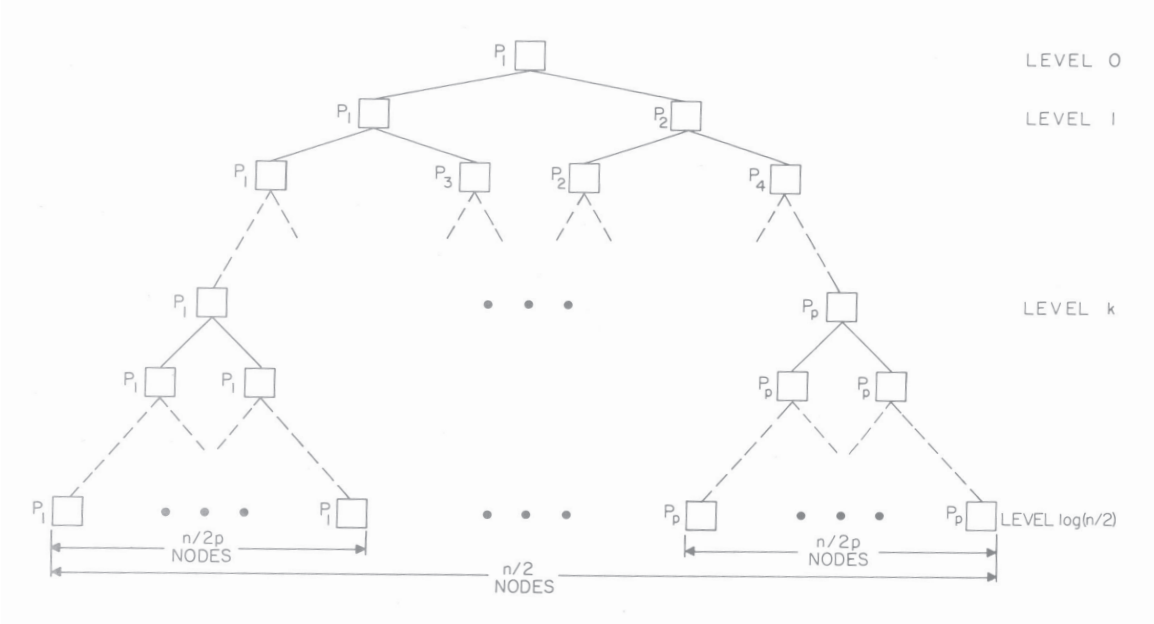


Figure 3.3: From [3], page 188. A figure used to verify the running time of a parallel quicksort algorithm.

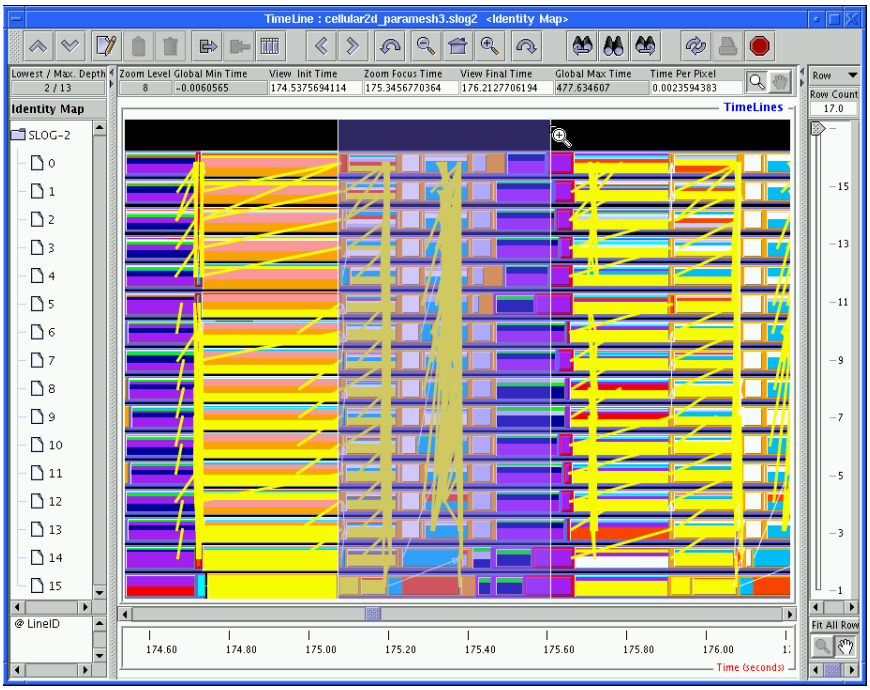


Figure 3.4: A screenshot from Jumpshot4

Chapter 4

Growing graphs

As an initial step toward visualizing large communication graphs, we will describe a rich yet practical way of relating families of large graphs using the notion of growth. Grammars permeate computer science, from theory, to programming languages, to biological growth systems [39]. As we will see here, there is a natural application of grammatical mechanisms to graphs.

4.1 String grammars

A simple and natural class of grammars are *Context-Free Grammars* [46], which appear in formal language theory. A Context-Free Grammar (CFG) is made up of a set of terminal characters and a set of non-terminal characters. A set of rules map non-terminal characters to strings, and a start symbol serves as a "seed" for growing strings. A simple example of a CFG is shown in Figure 4.1.

$$S \to 0S1$$
$$S \to \varepsilon$$

where ε is the empty string

Figure 4.1: A context-free grammar that generates strings of n 0's followed by n 1's

The key notion of a grammar is that it consists of a set of building blocks and a set of rules for growing arbitrarily large, mathematically defined structures from those building blocks. In Figure 4.1, the building blocks are 0, 1, and the empty string, and we have two simple rules that produce an infinite class of strings, namely $\{0^n 1^n \mid n > 0\}$ where 0^n denotes a string of n 0's.

String grammars can be expanded to *L-systems* [39], parallel string rewriting systems. Applicable rules rewrite each part of the string in every derivation of the grammar. Figure 4.2 shows an example of an L-system [37]. It generates a simple, aperiodic tiling of one dimension. This demonstrates the complexity of grammars and the structure of growth systems.

$$L \to LS$$
$$S \to L$$

gives
L
LS
LSL
LSLLS
LSLLSLSL
⋮

Figure 4.2: A simple L-system for generating aperiodic strings

4.2 Graph grammars

Similar rewriting mechanisms can be applied to graphs using *aggregate rewriting (AR) Graph Grammars*. At the most basic level, we can define a graph grammar to function just as a parallel-rewrite L-system would. In Figure 4.3 we give a grammar that generates the class of all complete binary trees. With each application of the grammar, we, in parallel, rewrite every vertex labeled with "L" to the simple three-vertex tree. All of the connections in the initial vertex labeled with "L," which indicates a leaf, will be added to the new vertex labeled with "R," which indicates an interior node

in the tree. We say that the "R" node *inherits* the connections of the left "L" node. Even with this straightforward application of graph grammars, we need the notion of inheritance, which establishes the context of the rewritten string and how it appears in the result string. Figure 4.4 shows the first few graphs in the derivation grown by this grammar.

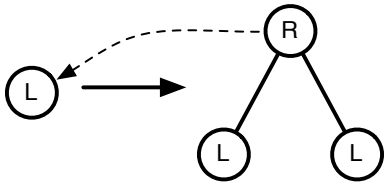


Figure 4.3: A graph grammar that generates complete binary trees

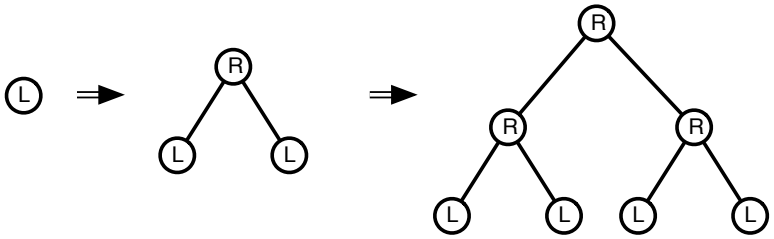


Figure 4.4: The first few trees in the derivation generated by Figure 4.3

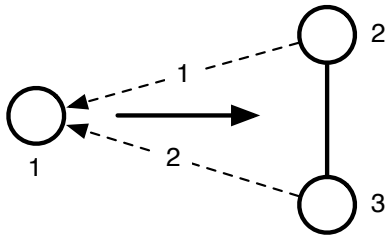
This mechanism alone can create a class of graphs that we could call "context-free graphs"—each production is applied without information about the remainder of the graph. In addition, every family of graphs (or "derivation") generated by this mechanism is monotonic: nodes and edges are never deleted. The generated graphs preserve features with each application of a rule. This is an important feature we depend on in development of our formalisms on clustering and highlighting nodes. The family of trees generated by the rewriting rule in Figure 4.3 are balanced and have bounded degree. These features are artifacts of the structure of the rewriting rule.

Languages of graphs that are generated by these mechanisms have a structure that parallels languages developed by L-systems. There are fundamental graphs that are not in any language of graphs that these grammars can generate. A full theory

of the structure of L-systems and graph-rewriting systems extends on the hierarchies of string-based languages [42]. It is, for example, easy to see that all these parallel systems can simulate equivalent structures to string grammars. In order to generate more subtly connected graphs, like hypercubes, we must add additional mechanisms to the grammatical system.

4.2.1 Partitioning: controlling duplicate inheritance

In our grammatical rules, we can add numerical labels to each vertex and then define a partition function based on these labels. Figure 4.5 shows an example of a rule with partitioning. Each ϕ_i represents a different partition; vertices that appear in different partitions' domains, like 2 and 3 in this example, will not be connected upon application of the rule. On the second application of the rule, each node in the 1-cube will be replaced with another 1-cube, and then the nodes labeled 2 will be connected and the nodes labeled 3 will be connected [5]. This creates a 2-cube, and upon further applications of the production, will create an n-cube. If we did not include these surjective partial functions that define partitions, we would grow the family of complete graphs on 2^n vertices. Figure 4.6 illustrates the distinction.



$$\phi_1(2) = 1$$
$$\phi_2(3) = 1$$

Figure 4.5: A graph grammar that generates hypercubes

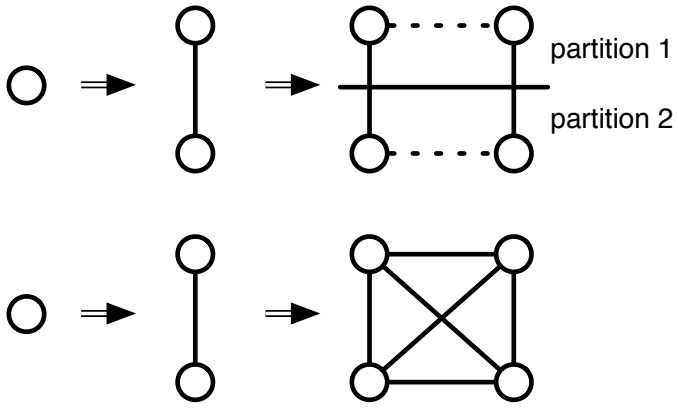


Figure 4.6: The families of graphs generated by the grammar in 4.5 with (top) and without (bottom) the partitioning functions.

4.2.2 Junctions

Another mechanism that can be used to control the growth of a structure developed by graph grammars can be applied to labeled junctions between each meeting of an edge and a node. If we use nodes to represent processors, then junctions are the graph-equivalent of processor ports. Junction partitioning and inheritance work in the same manner as node partitioning and inheritance; a second parameter is added to ϕ, the partition function, for junction inheritance.

Junctions allow for the growth of, for example, grid-like structures that we will see later are often more easily described through high-level procedures. As with any rewriting system or grammar, we would like to be able to describe its power by determining the class of objects—in this case, graphs—that it can generate. As string rewriting systems are Turing complete and as linear graphs that we can create with graph grammars mimic string rewriting systems, we know that our notion of graph grammars is Turing complete. Because of our interest in large, symmetric, recursively-generated graphs, the AR system's ability to produce variations of trees, cubes, meshes, and other typical process topologies is likely sufficient.

4.2.3 Parallel growth of node and edge attributes

If we were to construct large, representative graphs using the above graph grammar productions with no modifications, we would be left with huge graphs and no direct access to the grammatical history that generated the structures. To overcome this difficulty, we could update properties on nodes as the graph is being grown. In the cube example, we add a property called `address` to every node. The initial node starts with its `address` equal to the empty string. In the rule (see Figure 4.5) at node 2, we set `address` = `address` $\cdot$ 0. Likewise, at node 3 we set `address` = `address` $\cdot$ 1 where $\cdot$ here is the string concatenation operator. Properties like `address` can easily be added at every step that rewriting takes place. It is easy to imagine the growth of annotations associated with a graph's features.

4.2.4 Further annotations

One important extension that we will later find important is the annotation of a subgraph of the daughter as the "primary image" of the mother graph. This might, for example, be simply the domain of the inheritance function ϕ_0. Other annotations might indicate "extreme" or "corner" features that, after iterated application, identify important boundaries of the ultimate graph.

We implemented the graph grammars and the following grammar-like procedural methods in `Java`.

4.3 Procedural methods

As we have demonstrated, aggregate rewrite graph grammars are a potent mechanism for describing a wide variety of structured graphs. By tweaking the rules and the partitioning functions, we can generate several families of graphs. If we look at the structure of the productions that are applied to a graph, we can make insightful observations about the resulting graph family. For example, if there is a surjective inheritance function with non-trivial domain, the resulting graph will preserve the connectedness of the original. So, as a theoretical construct, an AR grammar provides the basis for many useful observations about graphs.

Practically, however, we might include a number of procedural methods for rewriting graphs. These procedures must maintain properties useful to, say, parallel programmers. They must also remain simple enough to support coherent and logically "obvious" growth of graph structures. Supplanting grammars with more complicated procedural methods would be counterintuitive. We will document several procedural operations that extend the capabilities of AR graph grammars in important ways.

4.3.1 Cyclification

The degree of every vertex in an n-cube is n, so the degree of every vertex increases linearly with the increase of n while the number of processors is growing exponentially. In an actual multiprocessor machine, this is not a practical topology because it will become increasingly difficult to connect each processor its logical neighbors. Cube connected cycles are a rewriting of an n-cube topology that yields a graph where every vertex has degree at most 3.

To construct a cube connected cycle, each vertex v in a cube is replaced by a cycle of n vertices. Each of the cycle-nodes uniquely inherits one of the n edges previously adjacent to v. Thus every vertex has degree at most 3. Figure 4.7 shows this procedure applied to a single node, and Figure 4.8 shows this procedure applied to a 3-cube. Physical realizability has an impact on routing time: the maximum distance between two nodes has increased from n to $n(n+3)/2$.[1]

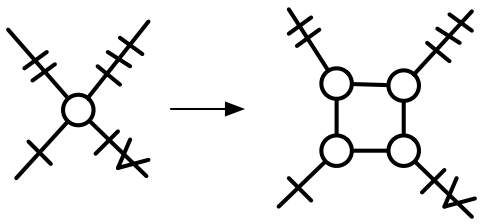


Figure 4.7: The cyclification of a node

The cyclification procedure can be applied to any graph to reduce its maximum degree to 3, and therefore to increase its feasibility as a real-world processor topology.

[1]Each step of a route of length n in the original cube has length n, plus an additional $n/2$ for each node of the original cube it must pass through, including the starting node. Thus the maximum distance between two nodes in the cube connected cycle is $(n+1)(n/2)+n = n(n+3)/2 = O(n^2)$

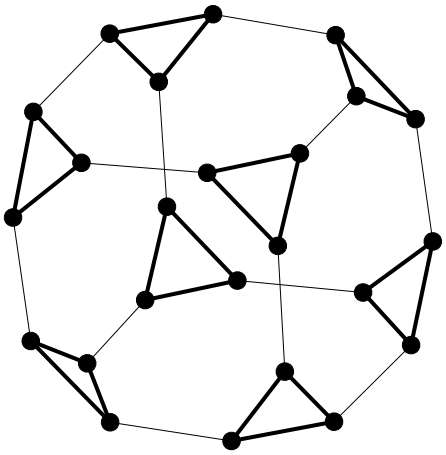


Figure 4.8: A 3-cube-connected-cycle. The bold edges are the created 3-cycles.

We should, however, add cycles under a constraint: we should not rewrite nodes with degree of three or fewer. Otherwise we would add processors that make communication paths longer and do not reduce the degree of any nodes. Figure 4.9 shows a 4 by 4 grid that has been augmented with cycles.

4.3.2 Vectorization with permutation functions

Suppose that a parallel program implemented a "shuffle" or permutation that distributed data between nodes (see Figure 3.2). This could be graphically represented by stacking multiple copies of our graph and having connections between them represent the shuffling interprocess communication. In another scenario, suppose that a finite element parallel program is computing the temperature on an object in 3-space. The heat at each point of the object is based on the heat of neighboring points. This communication graph will be a 3D mesh—a stack of 2D grid slices. Both of these examples make use of stacking graphs on top of one another, a process we call *vectorization*.

Let the vectorization function be $\mathcal{V}(G, f, k)$ where G is the graph we want to stack and f is the permutation function that defines the way in which we connect the k

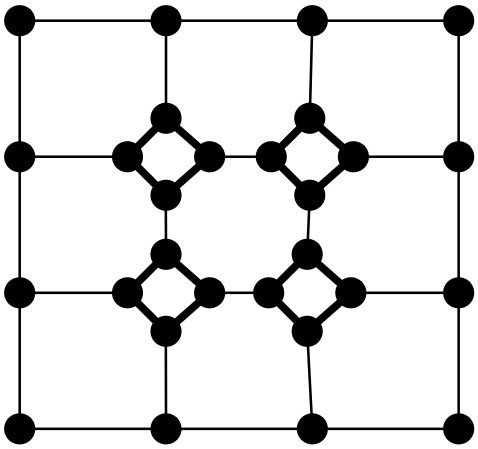


Figure 4.9: A 4x4 grid that has been cyclified

copies of G. Formally,

$$f : V(G) \to V(G)$$

is an automorphism of $V(G)$, the set of the vertices of G. Assuming that the vertices are labeled from 0 to $n-1$, we can define various shuffling and swapping functions.

To create the 3D mesh for our heat calculation, begin with G a two-dimensional grid and apply $\mathcal{V}(G, f, k)$ where $f : v \mapsto v$. We iterate this process $k-1$ times, generating one new copy of G at each step.

Likewise, to create a simple shifting between levels, we could set $f(v)$ to equal the node with v's label incremented by 1 mod n. So a simple shuffle takes

$$v_0 v_1 \dots v_{n-1} \mapsto v_1 v_2 \dots v_{n-1} v_0$$

Figure 4.10 shows a vectorized line with the aforementioned permutation function. Additionally, we can apply these permutation functions to graphs of arbitrary complexity. One example is shown in Figure 4.11, which is a vectorized binary tree with three levels.

In order to accommodate more complicated permutations between layers of the stacked graph, we could expand the vectorization to $\mathcal{V}(G, \mathcal{F}, k)$ where $\mathcal{F}$ is a set of bijective functions f_i where $0 \leq i < k$ that also depend on the value of k at a given iteration of the vectorization proecess. Our above formulation has the simplifying

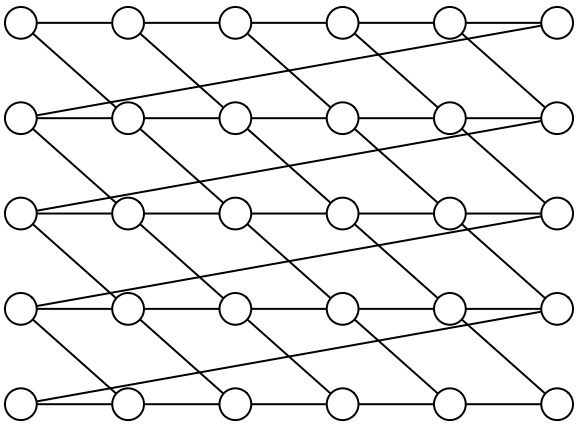


Figure 4.10: A vectorized line with permutation function $f(v) = u$ such that $u.label = v.label + 1 \mod n$

constraint that $f_0 = f_1 = \cdots = f_{k-1}$. This extended mechanism could be used to generate butterfly graphs and other complex sorting networks [16].

4.3.3 Grids and meshes

Note that while grids and meshes can be generated grammatically [5], they are simpler to construct procedurally with a series of nested `for` loops. Using this method does not preclude us from accessing features of large, grammatically generated graphs as discussed in Section 4.2.3 above. We can label the graph as though it had been grown from rows and columns using grammar-like mechanisms. In particular, it is likely useful to be able to identify the relative "age" of nodes in this simulated derivation and the obvious development of boundary features.

4.3.4 Compositions

It is worth noting that we can combine our methods of growing graphs. If, for example, a communication structure has a tree-like graph attached to each node of a hypercube topology, we can grow the corresponding graph by composing our cube and tree grammars.

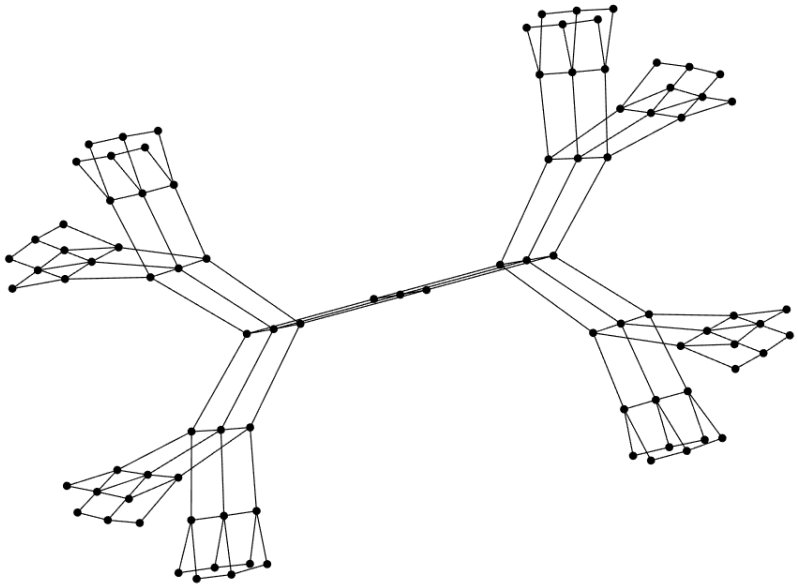


Figure 4.11: A vectorized binary tree

4.4 Gephi

We have built a system to generate graphs via a combination of Aggregate Rewriting graph grammars and of the novel procedural methods that we have mentioned. The graphs, including all of the added node properties, are rendered in `GML` (Graph Modeling Language), a portable plain-text file format that is simple to generate [24]. Once graphs are generated, we may import them into `Gephi`, an open-source `Java` graph visualization application [6].

`Gephi` is a tool for graph analysis that supports graphs up to 50,000 nodes. While this capacity may not be enough for all of the graphs that we eventually want to consider, it is sufficient for us as a testbed for ideas. Figure 4.12 shows the program in action.

`Gephi` seems primarily concerned with social network analysis, and other large,

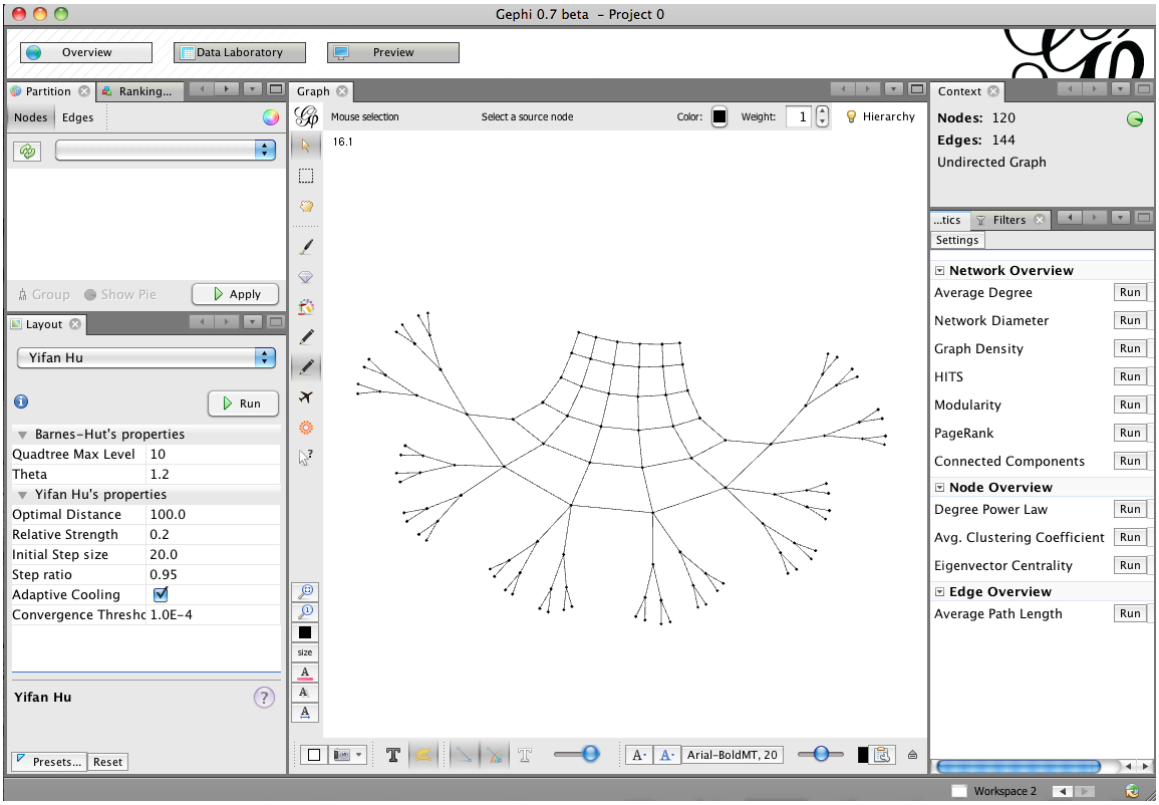


Figure 4.12: Gephi

non-symmetric graphs. Many of its built-in features serve to identify similar segments of these networks. After finding these clusters, `Gephi` provides tools for visually clustering and physically grouping sets of nodes together. Surprisingly, these similarity-finding techniques do not work well on our graphs because of their structured symmetries.

The program also provides a few layout mechanisms that give a fairly robust way of laying out graphs. The algorithm that we have found most useful is the Yifan Hu algorithm, an efficient, high-quality force-directed approach to graph drawing [25]. Without alteration, this algorithm allows us to visualize moderately large graphs, and strengthens our intuition as to whether a particular graph has a simple two-dimensional visualization.

`Gephi` is extensible, and provides a complete API. In later chapters, we will develop node metrics. In `Gephi`, these are called "statistics" on graphs. Using the Statistics API, we can cleanly implement new algorithms that assign values to new properties of nodes. We can then use the filtering interface in `Gephi` to show or hide parts of the graph based on the values of these new properties. With these filters, we can use sliders that will show nodes based on ranges of values, or we can show nodes based on the boolean value of a property. We can also compose filters with operators like `INTERSECTION` and `UNION`. Once we have used the filters to hide the nodes that we do not want to see, we then apply a layout algorithm to get a clean view of our simplified graph. `Gephi` also allows us to create new layout algorithms. All of the figures generated for this thesis are a result of our modifications of the `Gephi`-based toolchain.

Even with the built-in tools provided by `Gephi`, our large graphs are hard to visualize. We need a means to control how we place our focus and where we abstract sections of graphs.

Chapter 5

Theoretical foundations for large structure visualization

There is no precedent in the current state of graph drawing for visualizing highly structured, symmetric graphs. Here we present a few ideas about how we want to approach this problem. Both generalized fisheye views and the idea of feature classification within graphs give us useful ways to conceptualize visualizations that we might consider successful.

5.1 Fisheye views

In Section 2.1.2 we discussed the general use of a degree of interest function as a node metric that is useful for graph visualization. We noted that while useful, the method suffered from the problem of defining the *a priori* importance and distance functions. We can naïvely let the distance function $d(x, y)$ between the focus x and a node y be the length of the shortest x, y-path in the graph. This method of defining distance is not resilient in light of changes in the repeated substructure of the graph. For example, we may want distance to reflect distance between certain types of repeated subgraphs rather than between individual nodes; distance can be measured in different units other than edges. In visualizing cube-connected cycle generalizations of hypercubes, for instance, we may want to use the simple distance function from the underlying n-cubes rather than the naïve distance function of cube-connected cycles.

The *a priori* Interest (API) function, on the other hand, lacks even a naïve approximation. Any API function is making a decision about the inherent importance of a node in a graph, a task which, as we will see in the next section, is nontrivial and is one of the essential questions we ask when we try to visualize large graphs. In order to find relevant API and distance metrics, further insight must be made into the graphs.

5.1.1 Applicability to large graph visualization

The problem of large graph visualization seems to be aligned with the type of problem that can be solved with fisheye views. Furnas, the pioneer of generalized Fisheye (FE) Views notes a very important distinction to make when discussing fisheye views: the *what* versus the *how* [18]. The generalized fisheye view is a model for finding fisheye subsets of data: Furnas's seminal paper in generalized fisheye views puts forth the degree-of-interest (DOI)-based approach of finding these subsets [17].

In [18], Furnas claims that the majority of the work done for `focus+context` visualization problems has used some formulation of FE-DOI subsets. While the research in the field covers a broad spectrum of visualization techniques (distortion views, zoom views, view+overview, view+closeup, and multi-resolution displays [18]), they can all be formulated in terms of the FE-DOI function. These various papers are primarily concerned with addressing the question of *how* to display these FE-DOI subsets. The work of generalized fisheye views was intended for the application to data where we are lacking Euclidean geometry, such as graphs [18], and is therefore more concerned with the problem of *what* to include in the FE-DOI subset.

Furnas then puts forth three criteria for the applicability of a generalized fisheye strategy to a structure [18]. Our large, highly structured graphs satisfy these criteria, which are shown in Figure 5.1.

First, our graphs can be static, as they are built from the top-down as communication graphs from trace files of parallel programs, or from the bottom-up grammatically or procedurally. We could, if desired, add a temporal dimension to the graphs induced from trace files, but this is not a necessity. We think that, for visualization, the important view of a communication graph is the projection of the messages passed between processors. As we have discussed above, in the worst case, we can apply the naïve distance function between nodes to the graph. Of course, we can develop more complicated distances in our graphs, but they are not necessary to satisfy the criteria.

Criteria for applicability of the generalized fisheye model:

1. Some reasonably static structure with a notion of distance,
2. Some notion of independent level of detail or *a priori* importance for different parts of the structure, and
3. Interaction can be considered as focused at a point (or small region, or small number of points) in the structure.

Figure 5.1: Furnas's criteria for the applicability of the generalized fisheye model

Second, most of our graphs have some clearly important nodes. By creating graphs from grammars, importance that we assign to nodes of the daughter graph in the initial grammatical productions can be maintained and updated (additively, or otherwise) through each iteration of the grammatical generation process. Even in completely symmetric graphs like hypercubes, we can assign an *a priori* importance based on attributes of the grammatical generation. For example, for the hypercube, we can assign importance based on the n-bit `address` that identifies each node.

Our problem has a reasonable instance of a focus. If a parallel program has an error in a given processor or process, we often want to better understand the communication around that node. Thus we can identify a structural focus. If we want to investigate the structure between disparate processors, we can specify multiple foci.

Generalized fisheye views are intended for structures that satisfy the three conditions of Figure 5.1, and the existence of FE-DOI subsets of a structure give no indication as to the best way to visually represent the structure. In discussing some basic approaches for representing data that is hidden from view in a one-dimensional structure, Furnas notes that ellipses can be used to take the place of some of the lost information in a less-destructive way than simply omitting all unseen data [18]. This basic use of ellipses gives us precedent for our use of ellipses in visualizing large graphs. The following section shows that there is some precedent for using fisheye-like abstractions in important problems of understanding complex graph-like structures.

5.2 Application: Route maps

As an example of how FE views can effectively reduce visual complexity, we consider the presentation of route maps. Work by Agrawala [2] has created route maps that are tailored to give drivers a clearer picture of directions (see Figure 5.2). While this process modifies shapes, lengths, and angles of the roads, the changes are made under constraints that do not fundamentally change the route (ie. change turn directions, or add/delete road intersections). A key feature of the route maps is that they restrict their focus to the route in question and generally abstract away the surrounding roads. Not all roads that are not in the route are made invisible: as the contexts of the roads on the route among other roads *not* on the route are fundamentally important visual cues, they are included in places where they can be fit without disrupting the original route map.

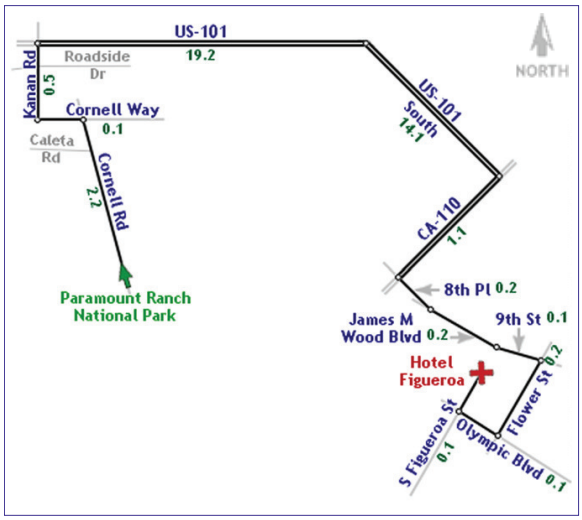


Figure 5.2: A route map generated by the techniques in [2]

While the procedure for reducing the visual complexity is the primary focus of Agrawala's research, methods of addressing route context are directly related to our goals for large graph visualization. The decision to show context via roads not in the route shows a three-way partition of information. Each road falls into one of three categories:

1. A road in a route is *important*: it is visible.
2. A contextually relevant road is *important*: it is visible, but less prominent.
3. A road is *unimportant* and we see no indication of it in the visualization.

5.3 Feature classification

Borrowing from the classification of information in route map visualization, we adopt a basis for visualizing large, highly structured graphs. Consider this model at a fundamental level: when attempting to visualize a large graph, or large structure in general, there will be certain fundamentally important pieces or patterns of the structure that we need to see in order to orient ourselves. There will be a collection of structural features that may add context to or clarify important parts of the structure, but they are not inherent to understanding the structure as a whole. Finally, there are features that are superfluous or unnecessary with respect to the features already identified as important.

Often we can find structures that have seemingly *no* unimportant features. In fact, for some graphs, the entire structure may be fundamental to its understanding. For small graphs such as the one seen in Figure 5.3, the task of partitioning the vertices into three classes is trivial: every vertex is important. Without any given vertex, we would be uninformed about a fundamental feature of the graph. Note that the three-way classification model remains valid, even for small graphs.

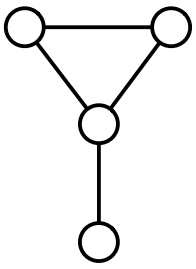


Figure 5.3: An example of a small, uniformly-important structure

Large graphs that are highly structured, whether they are constructed via the grammatical or procedural methods described in Chapter 4, are likely to have a

nontrivial classification of nodes. Grammatical methods for building graphs produce, by definition, graphs that contain repeated patterns, allowing for the separation of the final products into equivalence classes or "representative" components. Because our procedural methods of building graphs are all effectively iterative processes, the resulting graphs are highly patterned.

We will be able to find clearly important or representative features of the large graphs that will comprise the primary structure of our visualization. One method that we have of creating the three-way classification is by identifying a large set of nodes as important, and then pruning that set down to a small number of visible important nodes. The other important nodes are encapsulated in our application of ellipses to the visualizations. Ellipses allow us to indicate where repetitions of important information are occurring, and to do so in a way that does not clutter the visualizations with an overabundance of visible nodes. Finally, many nodes will not be in the initial large set of important nodes. These are the *unimportant* nodes: they do not directly contribute to the fundamental understanding of the visualization in question. Despite their lack of value to us, the ellipses we add will still, in most cases, stand in for unimportant nodes. Figure 5.4 shows a Venn diagram of our three-way classification model.

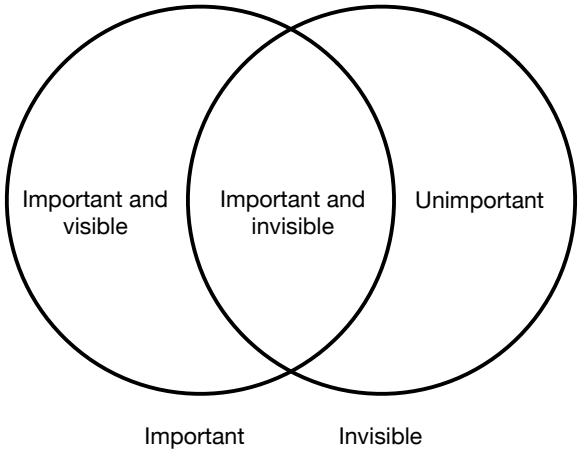


Figure 5.4: The three-way classification model for large graphs

The decision to omit unimportant nodes from a visualization is obvious. The goal is to show meaningful abstractions of large graphs, and if we included unimportant

nodes, our visualizations would quickly grow to an intractable size. Likewise, we cannot even include all of the nodes that we have deemed important. In many cases, the size of the set of important nodes is directly proportional to the size of the input graph. Techniques for visualizing graphs that showed the entire set of important nodes would therefore be unsustainable.

5.3.1 Non-uniqueness of classifications

To find the important features of large graphs, we can use techniques that vary in terms of justification from intuitive to mathematical. In the former case, we can make strong, reasoned arguments for why certain parts of our graphs are more important than others. Such arguments can be based on several things, such as different focal points of the graph, as well as our natural intuition for what the abstracted and abbreviated graphs *should* look like. For graphs where our intuition is weaker, we can harness various metrics for determining what nodes are important. When using these metrics, we have to be careful that they have a reasoned basis. Applying metrics that have no intuitive merit will likely result in visualizations that, while showing us *something*, will not give us natural representations of the graphs in question.

While some features of large graphs may be consistently important in all scenarios across all domains, the derived feature classification of a graph is generally a function of its ultimate use. In fisheye views, we calculated a degree of interest from a given focus. If we attempt to identify a three-way classification of a graph while keeping a specific focal vertex in mind, we must bias our methods toward that vertex. As a naïve example, if a method of finding the important nodes of a graph returns a set of nodes that do not include our focal vertex, then that method is not adequate for the given scenario.

Given different techniques, whether mathematical or intuitive, and different scenarios, like altering the focal points, we see that the classifications can vary significantly for a given large graph. In the next chapter, we begin to discuss our attempts to construct meaningful visualizations of large graphs, guided by the principles of fisheye views and feature classification.

Chapter 6

Large graph layout

In Chapter 4 we discussed numerous ways to construct large, practical graphs. The created graphs not random: they are highly structured by their use in parallel programming. Some are easy to draw and conceptualize in two dimensions and others are not. Many communication graphs that have a well-defined boundary, such as trees and grids, can be simplified for visualization in clean, informative ways. Graphs that are not easily projected onto two-dimensions present a greater challenge. In this chapter, we focus on large graphs that are easy to draw in two dimensions—trees and grids—and present an algorithm for abstracting structural patterns in graphs.

6.1 Trees

In a tree, suppose that one node is of particular importance. This could be the case if we are debugging a parallel program and there is an error in a specific process. The model of fisheye views seems applicable to this situation—we have a clearly defined focus and thus would want to find an appropriate distance function. Before we attempt to derive such a distance function, we will first consider what we should see in the visualization of a large tree. As a point of reference, Figure 6.1 shows a basic rendering of a depth-8 tree in `Gephi`. For trees that are any larger, this visualization becomes difficult to display in one frame.

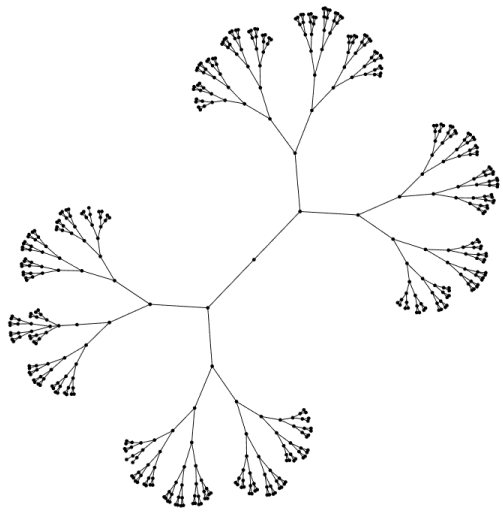


Figure 6.1: A visualization from Gephi of a tree with depth 8

6.1.1 Building a tree visualization

In a tree data structure, the root has an inherently vital role, so we assume that the root of the tree should be visible. We also assume that the focus should be visible. Apart from these two nodes, there are no other nodes that are fundamental to the display of the graph. Our goal is thus to find nodes that inform the context of the focus and the root without detracting from the importance of those nodes.

The focus's ancestors and descendants, beginning at the root and ending at a set of leaves, are probably of heightened importance. Additionally, if the tree was constructed for use in a parallel algorithm, the nodes that are at the same depth in the tree as the focus are also important because they, along with the focus, have similar functionality. Finally, the leftmost and rightmost nodes at each level are important because they establish the general shape of the tree. The leaves of the tree give further information about the shape of the tree. If we displayed all of the nodes that we have now defined as important, we would end up with a set of nodes that, although elided, grew larger and therefore was harder to visualize for greater initial tree sizes. We would end up with a visualization like Figure 6.2.

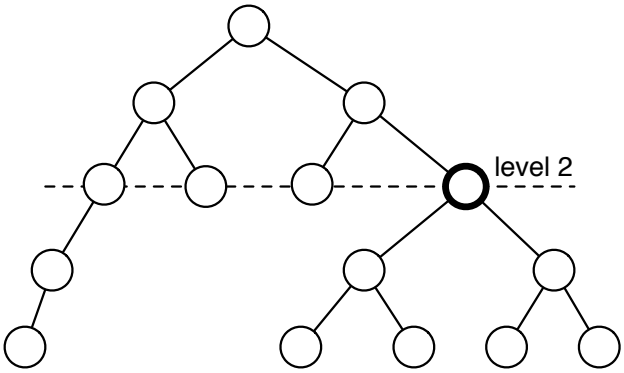


Figure 6.2: A 5-leveled binary tree with only important nodes shown

If we try to visualize this for a bigger tree where the focus is close to the root, we will end up displaying a huge subtree stemming from the focus. We must find a way to visualize the tree without showing all of these seemingly "important" nodes.

First, notice that in Figure 6.2 we see every element on the focus's level. We probably don't need to see all of these nodes; instead, we can leverage the fact that trees have natural boundaries. The left and right (or first and last, in a non-binary tree) children at each node establish an orientation to the graph. To understand that an entire level of a tree is important, it suffices to show the leftmost and rightmost nodes of the level, in addition to the focus. We can then use ellipses to signify that there are important nodes that we are not showing because of space constraints.

In visualizing the ancestors and descendants of the focus, there is an inherent asymmetry: if the focus is at level k, then it has k ancestors, whereas it can have exponentially many descendants. To abbreviate the path from the root to the focus, we can simply remove the majority of the intermediate nodes and replace them with ellipses. For the descendants, we can treat the subtree rooted at the focus as a new tree, and visualize it using these same methods. Essentially, the subtree will be visualized by showing a rough outline of its shape, using ellipses to remove complexity. The removal of large subsets of visible important nodes leaves us with a tree visualization like that of Figure 6.3.

Using ellipses to abbreviate graph visualizations establishes the three-way classification of nodes in a graph: visible important nodes, hidden important nodes, and unimportant nodes. As a general rule, *most* nodes should be unimportant, and *most*

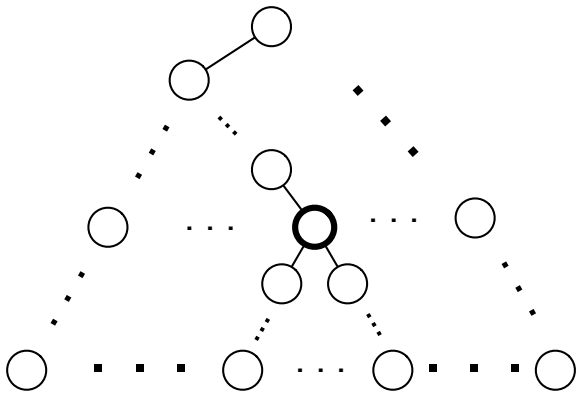


Figure 6.3: The visualization of a large tree using ellipses

important nodes should not be shown. The unseen important nodes should be clearly implied by the important nodes that we do see. The nodes we show should be the minimal set of important nodes that clearly imply the entire set of important nodes.

Note that this visualization works intuitively because trees have a natural planar symmetry. There is a definable left and right side of a tree (accessible via the grammatical construction), and trees are easy to draw in the plane regardless of size. Because trees are relatively easy to draw, they are easy to abbreviate. In Figure 6.3, the ellipses denote that certain boundaries of the tree are not displayed for the sake of clarity.

6.1.2 Multi-focal extensions

There is no fundamental reason why we need to limit our large tree visualization techniques to problems with a single focus. By adding multiple foci, we obtain a visualization of the interplay between a few nodes in a large graph. With two foci, we get a sense of the relative hierarchy between the nodes, the distance between them in the graph, common ancestors, and whether there is a path from the root to a leaf that contains the foci. These traits all inform the communication between two processes in the bigger picture of a parallel program.

As an example, suppose that there is an interesting event at the first focus. Judging by some algorithm we implemented, the first focus should receive data from the

second focus at some point in the execution of the program. By visualizing the communication graph (a tree, in this case) with two foci, we could see that, for example, there is not a hierarchical line of communication between the two processes—we must have misunderstood in our implementation of the algorithm.

6.1.3 A generalization for finding the context around a node

While we gave an argument for finding the context around a focus in a tree, we present, now, a more generalized framework for finding such context. We rely on the fact that graph grammars fully expose the nature of growth of our large communication graphs.

Suppose we have a graph G that was generated using grammatical methods. Thus G is some graph in a derivation of graphs $\{G_i\}$. A graph grammar production includes inheritance functions that copy the entire graph to the next graph in the derivation. Denote one of these inheritance functions as the primary inheritance function. Our graph rewriting system ensures that graphs in the derivation grow monotonically upon successive applications of the production (see Sections 4.2 and 4.3). We may also think of the growth of a graph as an accretion of features. As a result, each feature of graph G can be thought of as first appearing in some derivation step.

Therefore, given a focus f, there exists some i where G_i is the first graph in the derivation where f appeared. In the preceding derivation steps, $G_{i-a}, G_{i-a+1}, \ldots, G_{i-1}$, we can identify nodes that are, under some inheritance function, in the range of $\phi(f)$ in the rewriting process. These nodes are the immediate ancestors of f. Repeating this process with the immediate ancestors of f, we identify the ancestors of f arbitrarily far back in the derivation sequence. The image of these ancestral nodes in $G_i, G_{i+1}, \ldots, G_{i+r}$ identifies an increasingly large *context* about the focus f. Algorithm 1 and Figure 6.4 demonstrate this process.

The identification of the image of G_{i+r} in G, viewed along with some *a priori* interesting nodes in G, will give a sense of the important local context around f, as well as a picture of the greater graph. This method creates, in a sense, a ball in G that encloses the most important features around f. The parameters a and r form an interval in the derivation that can be adjusted based on the granularity that we desire to see in our visualization.

Algorithm 1 Display a ball around f in G

1: Find i such that f first appears in G_i.
2: Find all images of f under inheritance in G_{i-a}. Call these A.
3: Derive, from A, the subgraph of G_{i+r} that is C, the image of A. These features all have ancestors in A. The shape of this subgraph for many grammars resembles the family.
4: Highlight the *image* of C in G. These features are the important features about f in G.

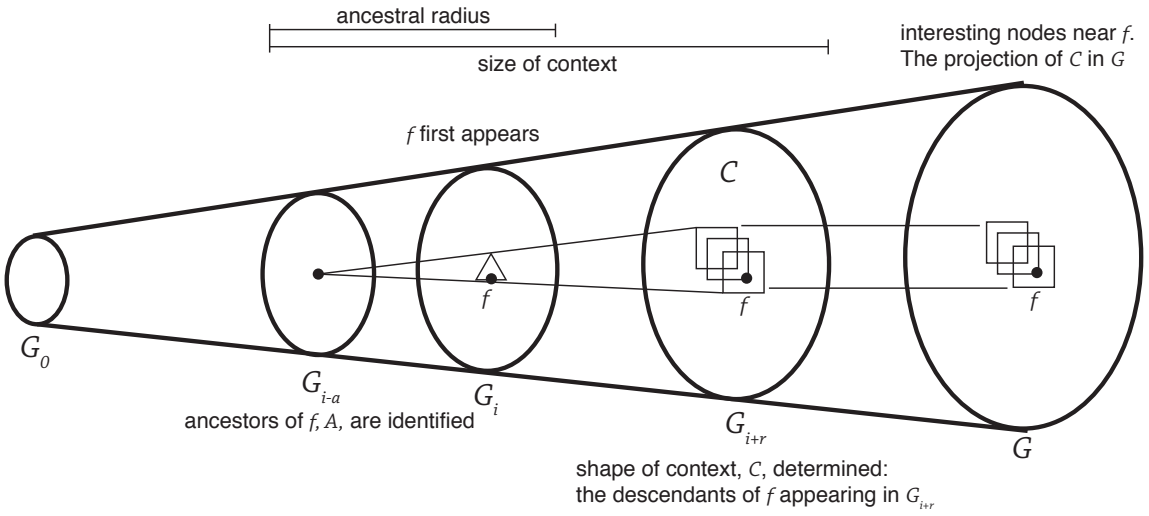


Figure 6.4: Finding a contextual ball around f

6.1.4 Constructing the sets

Given our intuition for the visualization of a large tree, we can algorithmically define the set of important nodes and within those nodes, define the important nodes that will be visible.

First, suppose we have focus f and root a in tree T with vertex set V. We have that l and r are the leftmost and rightmost leaves of T, respectively. Then let

$$\begin{aligned} Level &= \{v \in V \mid \text{level of } v = \text{level of } f\} \\ Sides &= \{v \in V \mid v \text{ is in the path from } a \text{ to } l \text{ or from } a \text{ to } r\} \\ Bottom &= \{v \in V \mid v \text{ is a leaf}\} \\ Context &= \{v \in V \mid \exists \text{ path from } r \text{ to a leaf that contains } f \text{ and } v\} \\ Important &= Level \cup Sides \cup Bottom \cup Context \end{aligned}$$

The set $Important$ is the collection of nodes that we have intuitively described as important: derivation information, extremal values, final nodes, and the seed for a contextual ball, respectively. We apply another set of rules in order to determine which important nodes are visible in our visualization.

$$Visible = \{v \in Important \mid d(v, f) < \rho\} \cup (Sides \cap Level) \cup (Sides \cap Bottom) \cup \{r\}$$

While the contextual ball gives us a sense of the important nodes around the focus, it can be applied to extremal nodes, as well. For example, in a tree, we can use Algorithm 1 to get a context around the leaves, with $G_i = G$ and $r = 0$. We can generally use this process for all nodes identifiable as extremal like, for example, nodes that appeared initially in the first and last derivations.

Additionally, $Visible$ should include the sides of the subtree rooted at f that is a subset of $Context$. In this formula, d is some distance function, and ρ is a scaling factor. The formula ensures that we see the important nodes that are close to the focus, as well as the important nodes that define the shape of the tree with respect to the focus. The important nodes that we see close to the focus are a specific case of finding a contextual subgraph, as we have described above.

Previous work on large tree visualization attempts to fit the entire tree on the screen via clustering [8] or folds in subtrees to create simpler visualizations [21]. If a graph is clustered, its interior structure is unknown to the user. Likewise, if subtrees of a graph are hidden via folding, the user may lose sight of the general shape of the

graph. Our method inserts ellipses such that the structure of unseen areas of the graph is implied.

Our method is, in disguise, a means of pinpointing an FE-DOI subset of our graph with respect to focus f. The distance function here is complicated, and thus easier to explain set theoretically, as above. Instead of naïvely letting distance be the length of the shortest path between f and v, the distance becomes nearer if v is on the same level of the tree as f *or* if v is a root-leaf path that contains f. These are the nodes that, in a tree structure, are most closely related to f. By finding this subset set-theoretically, we can easily see how the three-way classification forms; otherwise, we would have to define a distance function that hid the appropriate nodes of the tree when the threshold of the fisheye view was set accordingly. This adds unnecessary complications to an intuitive process.

6.1.5 Adding ellipses

As we claim that our visualizations gain their usefulness because of our use of ellipses. We now explain how they are added to the graph. For trees, we add ellipses when we remove nodes from the various subsets of important nodes that we constructed: *Level*, *Sides*, *Bottom*, and *Context*. Because each of these four sets is constructed based on an intuitive visual model of trees, all of the ellipses we add stay within these sets; none will cross over from, say, *Side* to *Bottom*. Because these ellipses are replacing sets of nodes that were both important and closely associated to each other before they were hidden, we claim that they will clearly indicate important patterns that are found in the graphs.

Ellipses could be incorporated in alternative ways, as well. Suppose that ellipses are simply highly deemphasized exemplars for otherwise hidden equivalence classes of nodes. By showing these ellipses in place of the less important, hidden classes of nodes, we provide hints that some nodes are missing.

Additionally, suppose that we find a visualization via the method outlined in Algorithm 1. We can then add ellipses extending "backward" from the nodes that first appear in G_{i-a} and "forward" from those that first appear in G_{i+r}. In this case, the ellipses are not visually well-defined, but they succeed as a means of abstracting the grammatical growth of large communication graphs.

6.2 Grids

Like trees, grids have well-defined boundaries. In fact, grid boundaries are even clearer because for trees, we must assert that the leaves have an ordering. While this is theoretically easy enough, in practice it requires extra bookkeeping. Regardless of how we construct grids, there will be naturally occurring boundaries at the edges. For square meshes, the distinctions of north, south, east, and west are arbitrary (so long as north is opposite south, of course).

6.2.1 Building a grid visualization

Using the concepts we developed from building a tree visualization, we can easily define important subsets of nodes in a grid. Suppose we have a focus f at row i and column j in a grid G. The important subsets of the grid are the boundaries of the grid, the row and column that contain f, and the internal context of f. The last subset mentioned, the internal context of f, is a square-shaped set with f at its center. In other words, it is the set of all nodes that fall within c rows and columns of f. This internal context can be seen as a specific example of finding a contextual subgraph, as we have described above. Figure 6.5 shows the internal context of f with $c = 1$.

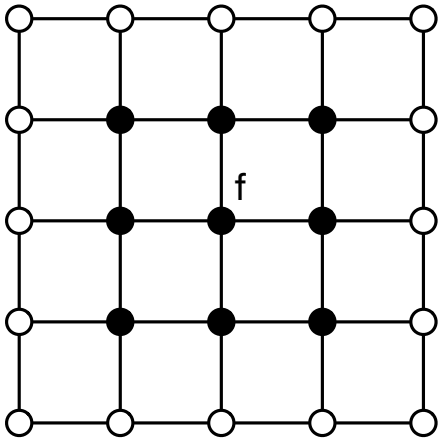


Figure 6.5: black nodes are the internal context of f with $c = 1$ in a subgraph of G

6.2.2 Constructing the sets

Given focus f at (i, j) of $m \times n$ grid G with vertex set V, we have the following important subsets:

$$\begin{aligned}
Row &= \{v \in V \mid v \text{ is at } (i, *)\} \\
Column &= \{v \in V \mid v \text{ is at } (*, j)\} \\
Top &= \{v \in V \mid v \text{ is at } (1, *)\} \\
Bottom &= \{v \in V \mid v \text{ is at } (m, *)\} \\
Left &= \{v \in V \mid v \text{ is at } (*, 1)\} \\
Right &= \{v \in V \mid v \text{ is at } (*, n)\} \\
InternalContext &= \{v \in V \mid v \text{ is } \leq c \text{ rows or columns away from } f\}
\end{aligned}$$

which gives

$$\begin{aligned}
Boundary &= Top \cup Bottom \cup Left \cup Right \\
Context &= InternalContext \cup Row \cup Column
\end{aligned}$$

and $Important$ is the union of these sets. Then we have that the $Visible$ subset of important nodes are those nodes contained in at least two of the seven subsets, or within a distance threshold ρ from f. Our formulation of the visible nodes allows more boundary nodes to be visible if f is closer than ρ or c to the boundary. As with trees, we then apply ellipses in place of elided nodes only within each of the seven important subsets.

6.3 A generalization

Methods for visualizing large graphs with well-defined boundaries take a general shape. The important nodes can be broken down into two subsets: the boundary and the context. While we derived these subsets via intuition for trees and grids, we have access to these features—or at least some approximation of them—for all of the large graphs we generate with grammars or our procedural methods.

The boundary of a graph can be found using the node properties we tracked in Section 4.2.3. In constructing trees via grammars, we keep track of a both an `address`

and a `generation` count. The newest generation of nodes in a tree are clearly the leaves, which are part of the boundary. Furthermore, node `address`es that are either all 0's or all 1's (in the case of a binary tree) indicate the left and right shells of the boundary. Nodes that are on the boundary of a grid are marked as such during the grid's iterative generation procedure. Our generation procedures are defined so that traits of the graphs, such as their boundaries, can be easily extracted.

In attempting to determine the context around f, we can use two different techniques. First, we can apply the generalized framework for finding contextual subgraphs, as described in Section 6.1.3. Alternatively, we can leverage node metrics on the graph to give us this relevant information. For graphs that have clear boundaries, we have found that simple distance metrics work well. Our graphs are highly symmetric, so more complicated metrics such as the aforementioned Strahler number do not tend to lead us toward a more enlightened abstractions. Graphs that do not have easily defined boundaries require more subtle approaches.

Chapter 7

Large graph layout without boundary

When we constructed the visualization of a large tree, we depended on the fact that trees are easy to draw in the plane. Likewise, two and three-dimensional grids have a natural planar visualization even as they grow large. While some parallel algorithm topologies may resemble these planar communication structures, there are certainly cases where this is not true. The n-cube, for example, is trivial to visualize for $n = 1$, $n = 2$ or $n = 3$, as seen in Figure 7.1. Even 4-cubes have a reasonable two-dimensional projection (Figure 7.2). However, as n grows further, finding a meaningful two-dimensional projection of an n-cube becomes difficult.

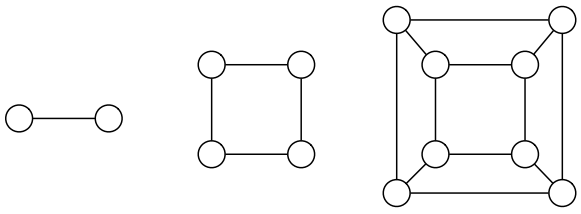


Figure 7.1: Simple planar drawings of 1, 2, and 3-cubes

Hypercubes do not have natural two-dimensional projections in part because their highly symmetric structures do not lend themselves nicely to a definition of boundary. Both trees and grids have a clear definition of boundary (sides, tops, and bottoms)

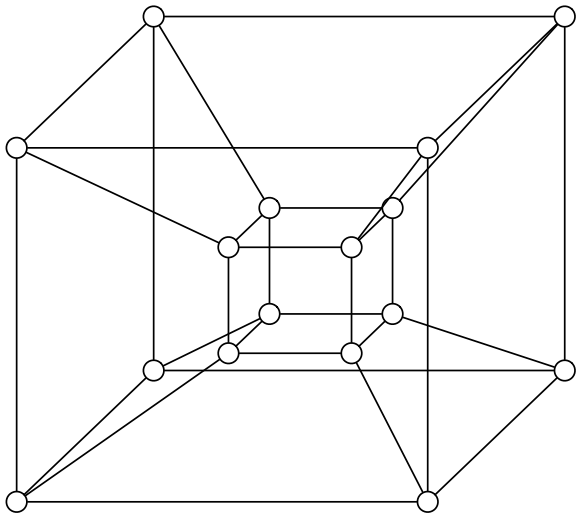


Figure 7.2: A two-dimensional projection of a 4-cube or tesseract

that hypercubes simply do not have. Our desire to find clean and abbreviated two-dimensional visualizations of hypercubes is driven by their prevalence and their utility. One reason that hypercube architectures have been successful communication structures is that common parallel algorithm topologies—notably grids and trees—have embeddings into the hypercube. For example, if a grid has d dimensions and p is the sum of the log of the size of each dimension, then the grid can be embedded into a p-cube [43]. We still want to be able to visualize cubes because (a) they will not always be embeddings of these simpler structures and (b) we can embed grids of arbitrary dimension in cubes, and even though such grids *have* natural boundaries, we have not shown a practical method of displaying them in two dimensions.

The contextual ball algorithm (Algorithm 1) can be applied to these graphs as well, but it is more challenging to see what features are important. Thus we present additional metrics that may give informative views of these difficult graphs.

7.1 Imposing boundaries on the unbounded

While cubes do not have natural boundaries, we have still, via the growth of the cubes, updated properties on each node. Specifically, each node has an `address`, a

unique n-bit binary string. All n-bit addresses are used by the 2^n nodes in an n-cube, and the distance between nodes A and B is the Hamming distance of the addresses of A and B, or the minimum number of bit-flips necessary to change the address of A to the address of B. The ordering of the collective `address`es is arbitrary. We could, for example, split every node's `address` in half and swap the halves, creating a new address for every node. This would not change any of the `address` properties.

But even though the ordering of the address bits is arbitrary, we have chosen one of these orderings when growing the cube. Specifically, we have chosen the ordering that corresponds directly with the growth mechanism as conceived by AR graph grammars. Given this ordering, we can impose boundaries on hypercubes. We could let every node with $address = 0a_1a_2\ldots$ be on one boundary, while nodes with $address = a_0a_1\ldots 0$ are on another boundary. In keeping with the general plan we outlined in the previous chapter, we will visualize large graphs with respect to a focus or foci. To aid in our layout techniques, we can suppose that our focus f with address $ab\ldots yz$ lies on the intersection of two boundaries: those that have address $a\ldots$ and those that have address $\ldots z$. By using a property of our focus, we can construct an (admittedly arbitrary) boundary around that focus. Obviously this boundary, which contains $2^n - 2^{n-2} = 3(2^{n-2})$ nodes, is growing exponentially with n. We will have to employ other techniques in order to use this conceptual boundary to generate an effective visible subset of the nodes.

7.2 A property-based layout algorithm

We want a way to visualize a graph using properties that we have derived. Such properties will typically be related to the focus of large graphs, so the visualizations we generate will be customized based on a given application. Given two numeric properties on graphs, we have developed an intuitive layout algorithm, which can be seen in Algorithm 2. The primary use of this algorithm is to show relationships between different node metrics.

The algorithm is simple: it assigns coordinate values to visible nodes directly based on its values for two distinct node metrics. Before implementing this algorithm, we would be forced to view the visible segments of our graphs by one of `Gephi`'s standard layout mechanisms, which do not account for the additional structural information we have added via node metrics.

Algorithm 2 Property-based layout on two metrics

```
N_x ← an x-normalization factor
N_y ← a y-normalization factor
for each node n do
  if n is visible then
    n.x ← n.metric_1/N_x
    n.y ← n.metric_2/N_y
  else
    n.x ← undefined
    n.y ← undefined
  end if
end for
```

7.3 Useful node metrics

We lack an intuition about what exactly we want to see when we are visualizing a graph that lacks boundary. Instead of forcing our perceptions onto a visualization, we attempt to find interesting concepts which we can extend to node metrics. These metrics give us a toolkit with which we can attack our visualization problem. Whereas we cannot conclusively say that one means of visualization is unequivocally *the best*, we seek to find interesting and relevant ways to explore these difficult structures.

These metrics present alternatives to the method described in Algorithm 1 for finding contextual subgraphs around important nodes. Each metric assesses the graph in a different way, but they all depend on the features guaranteed by grammatical or grammar-like growth. Generally, they use the `address` property, which is annotated on every node throughout graph derivation.

7.3.1 Single-source shortest path

If we assume that every edge of our graph has weight 1, we can trivially construct the naïve distance metric between our focus f and every other node in the graph. To find the single-source shortest path distances from source f, we implement Dijkstra's algorithm in `Gephi`. This metric gives a baseline for all other metrics, and it allows us to visualize other metrics versus our conventional notion of distance via our property-

based layout algorithm.

7.3.2 Edit distance

The edit distance between two strings is the cost of transforming the first string into the second via a series of insertions, deletions, and substitutions. Each of these actions is assigned a cost, and through dynamic programming, we can determine the most cost-effective transformation. This notion of edit distance is formally known as the *Levenshtein distance*.

We can apply edit distance to graphs by distinguishing a specific property for comparison. Our process of growing graphs has given each node a unique `address` property, so we can apply the edit distance metric to that. This differs significantly from the naïve shortest path distance metric because in computing edit distance, we can make jumps between vertices that are not along existing edges. For example, take two nodes with addresses 0110 and 1011. One possible shortest path between these two nodes is

$$0110 \xrightarrow{1} 1110 \xrightarrow{2} 1010 \xrightarrow{3} 1011$$

which has length 3. However, the edit distance between these addresses can be found by first deleting the trailing 0 and then inserting a leading 1, or:

$$0110 \xrightarrow{1} 011 \xrightarrow{2} 1011$$

Thus the edit distance between these two addresses is 2 (provided that the cost of deletion and the cost of insertion are both 1).

While the shortest path distance is more intuitive in the graph-theoretic sense, viewing other nodes with small edit distances from the focus could give insight into the structure of the rest of the graph. Furthermore, there may be other properties that could be constructed via the growth of graphs for which edit distance is a strong indicator of structural importance. Our example uses the `address` property, which by definition is tied to the simple graph-theoretic idea of adjacency. If nodes x and y are adjacent in one of our graphs, they will have `address`es that are related either by one insertion, one deletion, or one substitution. In that sense, edit distance with substitution cost 1 will mimic shortest path distance for the immediate neighbor of f. However, as nodes get further away by shortest-path distance, they will be strictly not further away in edit distance.

7.3.3 First bit set

The `address` of a node is a binary encoding of the way the graph was grown. By extracting features from the `address` or other properties, we can develop simple pictures of complicated graphs through abstraction. One such simple feature of the `address` property is "first bit set." Simply put, the first bit set of an `address` is the first bit from the left that is nonzero. If the `address` is all 0's, then the first bit set is -1. This measure partitions the graph into a series of subgraphs that reflect the growth of the larger graph.

For example, finding first bit set on an n-cube gives n subgraphs: an $n-1$-cube, an $n-2$-cube, etc. all the way down to a 1-cube, and two 0-cubes. These cubes correspond to cubes with first bit set values of $\{1, 2, \ldots, n-1, n, -1\}$. These subgraphs conveniently show the growth of the hypercube class of graphs; by revealing the substructure of graphs, the "first bit set" property allows the user to better understand how the graph was formed.

This formulation is not focus-specific, but it can trivially be modified in order to relate to f. Instead of calculating the first bit set, we calculate the first bit that deviates from the address of f. By doing this, we are essentially realigning all the addresses with f's address as the new zero, and computing first bit set from there.

Not all graphs that we could grow will be some composite of binary features. For example, while a binary tree has a natural binary `address` growth, a ternary tree does not. We can extend the idea of first bit set to properties with alphabets of size k by defining $k-1$ new properties. If the alphabet for a ternary tree's `address`es is $\{0, 1, 2\}$, then we can define "first bit 1" and "first bit 2." As we have one of these properties for each of the $k-1$ degrees of freedom, they give us the same power as "first bit set" for binary addresses. Luckily, computer scientists tend to prefer binary systems, as they are the physically realized standard. We do not expect to need to reason about the structures over these larger alphabets.

There are parallels between "first bit set" and the contextual ball we identify in Algorithm 1. The "first bit set" metric identifies i in step 1 of the algorithm. Given a node with `address` abc in G_i, its image in G has address $abc000\ldots$ Likewise, the descendants of abc in G are all nodes prefixed with abc. The ancestors of a node with `address` $abc\ldots xyz$ are all nodes in any of the G_i with `address` a prefix of $abc\ldots xyz$.

7.3.4 Strahler numbers

We mentioned Strahler numbers in our earlier discussion of large graph visualization, and we noted that they are directly applicable to large trees and directed acyclic graphs. Unfortunately, these are graphs that tend to have boundary already. We have other methods of finding visualizations that capitalize on important substructures of trees. Strahler numbers on highly structured graphs are not very useful as a means of demonstrating high-bandwidth parts of the graph because every set of functionally equivalent nodes in a graph will have identical Strahler numbers.

Still, we could find an interesting use for Strahler numbers. For the majority of our work, we are assuming the worst case scenario: that our communication graph has unweighted edges, and therefore every spanning tree is a minimum spanning tree. Instead suppose that we have edge weights that represent communication frequency or throughput in a parallel algorithm. If two processes exchange values frequently, the edge between their corresponding nodes will have a high weight. We then can compute a nontrivial minimum spanning tree over the multiplicative inverses of the edge weights. This will give us a spanning tree that includes important communication lines between processes.

We can then compute the Strahler numbers of each node by rooting our minimum spanning tree at the focus f of the graph. The Strahler numbers will give us a symmetric view at how the important lines of communication throughout the graph are structured with respect to f. Even though this will be, as noted above, a highly structured assignment of Strahler numbers to nodes, we can group together nodes with identical Strahler numbers into equivalence classes. We can then use the generated equivalence classes to show only clustered segments of the graph. Without a notion of importance or frequency of communication on edges, Strahler numbers provide an incomplete and redundant view of graphs.

7.3.5 The "symmetry from boundary" metric

In attempting to visualize a hypercube, we quickly realize that there may not be a natural visualization. By using the theory of imposing a boundary on an unbounded graph, we develop a novel method of viewing an important substructure of a cube from a focus f. Consider the following:

1. We want to see more nodes close to f while retaining a sense of context to the

rest of the graph.

2. Let g be the node that is furthest away from f. It is important because f and g are on opposite "sides" of the graph and can give us bounds on the structure.

3. By combining (1) and (2), we argue that we want to see progressively more nodes as we move from g to f.

We will construct a subset of important nodes from focus f by starting with g. Note that the address of g is the address of f with each bit flipped. From g select adjacent nodes that differ in address from g only at either the leftmost or the rightmost bit of the `address` property. This gives two nodes. For each of those nodes, select nodes that differ in address only in either the leftmost yet-uncompared or the rightmost yet-uncompared bit. This will select three more nodes: if g has `address` 00000, then the first nodes selected have addresses 00001 and 10000, and the second set of nodes selected have addresses 10001, 00011, and 11000. Notice that 10001 is selected by both 10000 and 00001. After n recursive layers of this procedure, we will have n nodes that select only f. As we select nodes, we recursively assign a value progressively further from zero, either positive or negative, depending on how many leftmost bits and how many rightmost bits of the `address` have been changed from g's address.

This value becomes the "symmetry from boundary" metric (SB) (see Algorithm 3). For example, if g has `address` is 00000, the SB of the node with `address` 10001 will be $1 - 1 = 0$ and the SB of 11101 will be $3 - 1 = 2$. The $VisibleSet$ created by the algorithm specifies which nodes we want to see in our visualization.

7.3.6 Combining metrics

Figure 7.3 shows a visualization of a 6-cube generated by pairing SB with the shortest-path distance metric and the property-based layout algorithm. This graph has focus f with `address` 111111. The nodes selected by the SB algorithm are the visible set, the SB is the x-axis, and shortest path distance from g is the y-axis.

Applying two metrics together like this can inform properties of the larger graph. Once we have collapsed a graph into a visualization like Figure 7.3, we can apply earlier techniques of simplifying large graphs. These techniques will introduce ellipses into our visualizations of large graphs without boundary. We could remove

Algorithm 3 Calculating the "symmetry from boundary" (SB) metric

```
f ← the focus
g ← node with negated address of f
VisibleSet ← {g}
i ← 1
while f ∉ VisibleSet do
  for each node n distance i from g do
    if address of n differs from address of g on only leftmost and/or rightmost
    bits then
      l ← # of leftmost bits changed from g to n (the "left-Hamming distance")
      r ← # of rightmost bits changed from g to n
      n.SB ← (l − r)
      Add n to VisibleSet
    end if
  end for
  i ← (i + 1)
end while
```

some of the middle layers of the subgraph shown in Figure 7.3, which would give us a natural way to reduce the visual complexity. Any visualization that uses the above metrics can be attached to a slider in `Gephi` that changes the threshold of visibility. Using these sliders, we can move between visualizations of varying granularity.

Users should be able to grow and annotate graphs however they want. The above metrics will give them ways of looking at general annotations that will define some measure of distance or relationship between at least one node and all others. Our particular operations are natural in a parallel environment because they depend on a connected, highly symmetric graph. These metrics are clearly distinct, and there may be properties that can be used other than `address`, which was accessible for the families of graphs on which we focused.

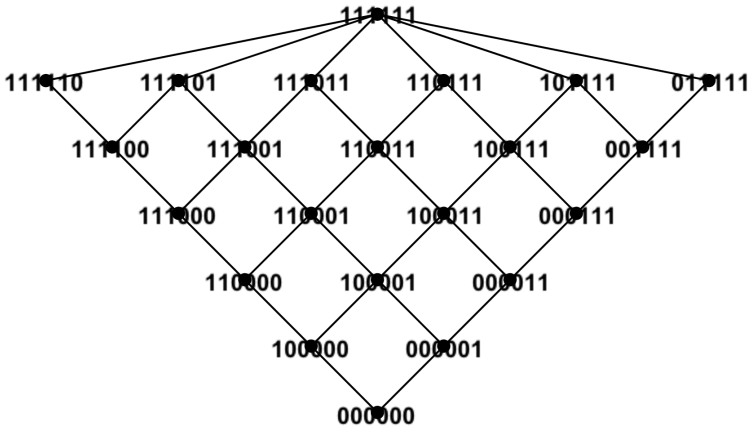


Figure 7.3: The property-based layout algorithm applied to a 6-cube with focus 111111 using the SB metric and the shortest-path distance metric

7.4 Evaluation

We evaluate results by discussing what we do and do not see in their visualizations, and we can discuss where we would ideally see ellipses.

7.4.1 Metrics

Shortest path distance

Applying a single-source shortest path algorithm to our graph from our focus is a distance metric. In our visualization, we can interactively vary the weight of nodes and visibility threshold of this metric. By adjusting this slider from 0 up to the largest calculated distance, the user can see how the graph forms around the focus.

Figure 7.4 shows a 6-cube with nodes that have distance 2 or less from shown f. This does look like a corner of a cube, but we have no sense of a greater context.

We can add a sense of context by assigning a second focus to another important node. Figure 7.5 uses our property-based layout algorithm with distance from one focus on the x-axis and distance from the other on the y-axis. Notice that these visualizations of cubes will show a grid-like "slice" of the graph; several nodes are

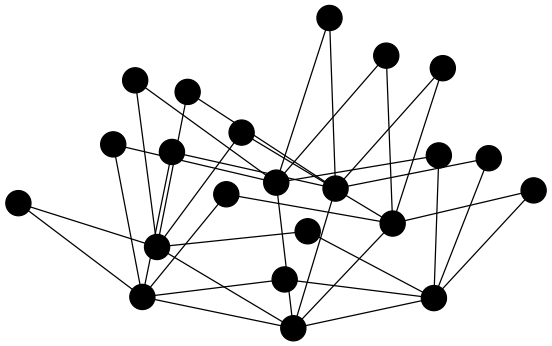


Figure 7.4: A 6 cube showing nodes within distance 2 of the focus

overlapping in the visualization. This is promising: it stands to reason that the overlapping nodes are equivalent with respect to the two foci. If we were dealing with a larger graph than a 6-cube, we could reduce the visual complexity further by applying our method from Chapter 6 for visualizing large graphs with boundary via the three-way classification and ellipses.

Edit distance

Edit distance is a metric that treats the properties of nodes, such as `address`, as strings. The `address`-based edit distance between two nodes is not a solely a function of the graph itself, but of the semantics of the `address` property. This inconsistency between the edit distance metric and the graph itself leads to problems in interpreting the visualization.

Figure 7.6 shows the 6-cube with an edit distance metric applied. Here we are showing nodes that have low edit distance (≤ 2) and nodes that have high edit distance (≥ 4). We show these two sets of nodes in order to give the user a sense of the global context around f. We note that there are some nodes that are shortest-path distance greater than 2 that have edit distance of 2 or less. While this is part of the reason we have selected to try edit distance—it provides a different, more flexible idea of "closeness" in a graph—it turns out to be difficult to understand in a visualization. Because the `address` is an artifact of the shortest-path distance between nodes in the graph, using a different-but-not-too-different measure of distance can be perplexing.

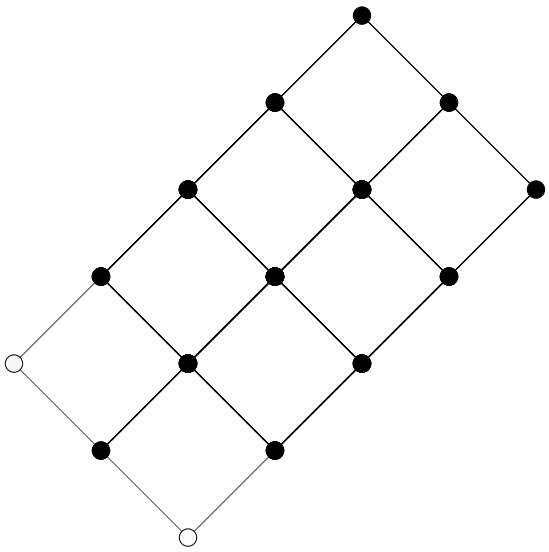


Figure 7.5: A 6 cube showing distance from two foci (in white)

It remains possible that edit distance could be useful for node properties that have a deeper meaning than merely encoding the graph's edge set.

First bit set

The "first bit set" measure applied to cubes decomposes the cube into a series of smaller cubes. This can be useful for visualization to see the smaller substructures that comprise the larger structure. Because our graphs were systematically grown, the large graphs will duplicate the patterns of the smaller subgraphs. By understanding the shape of communication of the smaller graphs, we can get a better picture about the shape of the larger graph. Hypercubes are, after all, mathematically beautiful. If we understand how a 3-cube works, it is not a great leap to consider the 4-cube. By allowing us to see the decomposition into cubes, first bit set can remind us of the simpler structure within, the structure that we already understand.

Strahler numbers

In our discussion of Strahler numbers, we have mentioned that they can only be applied to trees and DAGs. We then hypothesized that, using a spanning tree induced by the most heavily frequented edges in a parallel program communication graph, the Strahler numbers can inform us about equivalence classes in large, structured graphs. These classes would not merely be related to the structure of the graph, but the actual communication that was occurring.

We could also find a spanning tree or spanning DAG of the graph as some function of the way the graph was grown. This would guarantee that we were computing the Strahler numbers given an important subtree of the graph even if we did not have access to communication frequency data.

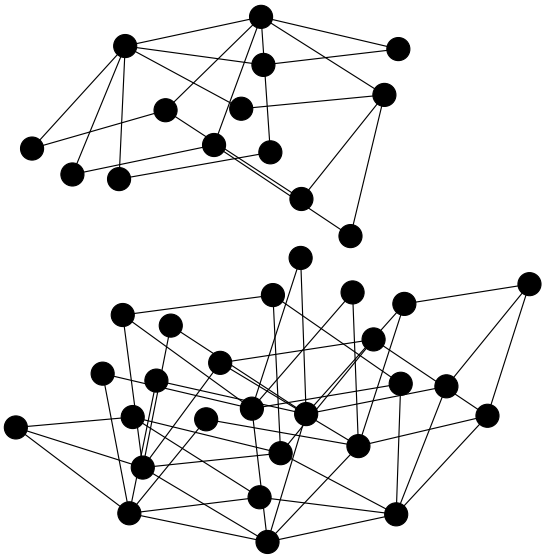


Figure 7.6: A 6-cube. We see nodes with edit distance not equal to 3

The SB metric

Figure 7.3 shows the SB metric applied to the 6-cube. This visualization could be further simplified by using ellipses to remove some of its middle layers. In larger

graphs, this would be necessary in order to keep the visualization at a manageable size. Otherwise, this metric combined with the shortest-path distance metric and our property-based layout algorithm seems to give a good picture of the cube.

We get a sense of the context near the focus and, in decreasing amounts, nodes that are further away. This extends all the way to g, the node that is furthest away from the focus. This visualization significantly reduces the number of nodes we see in an n-cube, from $2^n = O(2^n)$ to $(n(n-1)/2) + 1 = O(n^2)$. With the addition of ellipses, we can easily reduce this to $O(n)$ nodes, which seems like a reasonable goal. Seeing a $O(1)$-node visualization for an n-cube would suggest that we can fully understand a cube without knowing its size, which seems like a flawed conjecture. We cannot get a full picture of a cube without having a sense of the distance between its "sides."

The contextual ball

Algorithm 1, the contextual ball algorithm, is well-defined for any graphs that are grown grammatically. Applying it to a cube gives us a series of smaller cubes around our selected important nodes, such as the focus and its negation. Adjusting a and r affects the detail of the visualization by changing the size of these smaller cubes. As a highly symmetric graph, each node of a cube has the same structural properties. However, because our cubes are products of grammatical derivation, foci that first appear in, for example, G_1 or G will not be viewed the same way as "interior" nodes with this algorithm. This can be corrected: if $a + r$ is equal for two nodes in a cube, their visualizations with both be $(a + r)$-cubes.

Figure 7.7 shows a naïve visualization of an 8-cube. The nodes in black represent a contextual ball around some important nodes (arbitrarily 00000000 and 11111111). Figure 7.8 then shows this same visualization, but abstracts the nodes that do not appear in the contextual balls. This subgraph gives us an intuition about the communication structure within a large cube. The two components visually appear close together. This is a byproduct of our initial naïve layout, but it still hints at the mathematical fact that, although our foci are as distant as possible, the shortest path between them is still small.

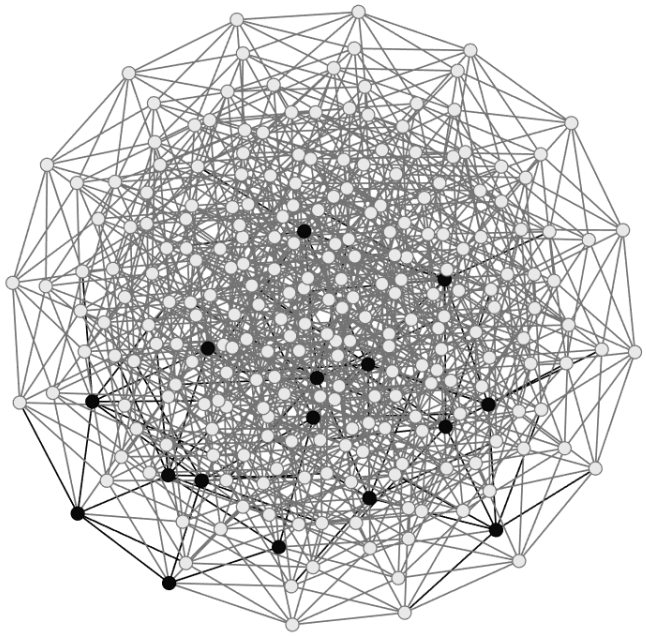


Figure 7.7: An 8-cube. The black nodes represent contextual balls around key nodes.

7.4.2 Different kinds of graphs

We have developed a theory of visualization for graphs that have have a clear boundary and intuitive planar representations. Our model denotes certain nodes as important in the local scope of the focal points, and other nodes as important in the greater scope of the entire graph. Because these graphs have been grown according to the highly structured specifications of our grammars or procedures, we can abstract these important nodes into a smaller set of representative nodes. These nodes, chosen to represent the graph's greater structure but also to indicate its important details, become the visible subset of nodes, the nodes we show in our visualization. Ellipses are then added to signify abbreviated structures within the graph.

While this model seems to function at an intuitive level, we struggle to find the important subsets of nodes with certain types of graphs. Notably, graphs that do

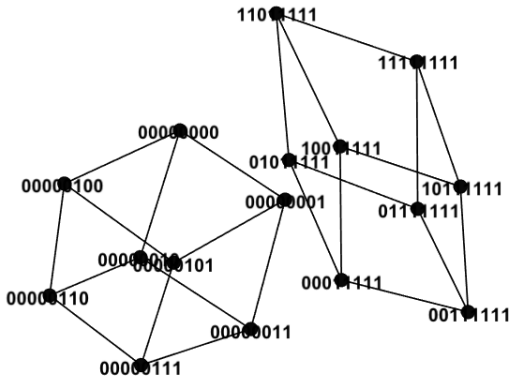


Figure 7.8: The same 8-cube as in the previous figure, but abstracting nodes that do not appear in the contextual balls.

not have a clearly defined boundary or those that cannot easily be visualized in two dimensions are problematic. If we attempt to, as suggested by our model and the generalized fisheye view model, find nodes that were *a priori* of high importance in a hypercube, we find ourselves lacking a satisfying solution.

In order to find such important nodes in these problematic graphs, we propose the use of node metrics to limit the nodes that are visible. We do not expect to be able to find one such metric that is universally useful over all of our grown classes of graphs. Instead, by presenting a toolkit of metrics, we hope to offer new options for visualizing these graphs.

We have also proposed a general method for finding a contextual subgraph around any individual node in a graph that has been grown via a grammar or grammar-

like procedure. Harnessing the structural symmetries of graphs—even those that we can not naturally visualize in two dimensions—allows us to determine a set of fundamentally important nodes given foci and extremal points.

Chapter 8

Conclusion

To again use the quote from Mehra et al. [35]: "some objects, perhaps those less familiar to us, have no obvious natural abstraction."

Simplifying the visualizations of graphs that have no obvious visualizations in the first place is a hard problem. Our main contribution to this is a set of tools for growing graphs and then aiding in the visualization process of the difficult families of graphs. We have also put forth a general model for effectively visualizing large, highly structured graphs that have an obvious notion of boundary using our conceptual ball algorithm. Our model provides a rationale for an approach to visualizing large graphs with boundary. By finding the three-way classification of important/visible, important/invisible, and unimportant/invisible vertices, we gain a condensed, abstracted view that does not remove meaning from the large graphs.

Dimensional limitations

The harder problem that we faced is that of visualizing our large grown graphs that do not have obvious boundaries. Our goal for these types of problems is to find a useful, intuitive, and reasonably sized visualization of complicated structures that have no clear two-dimensional visualization. If the large structure as a whole does not have a natural projection onto two dimensions, it is unlikely that we are going to find a bulletproof method of extracting easily visualizable subsets in a way that will clearly indicate the patterns of the larger structure. It is this observation that prevents us from offering a unified theory on how to visualize all large, highly structured graphs.

Although we are dealing with only a small subset of related graphs (notably those that can be grown from grammars and procedures in highly structured ways), there is likely not a universally successful way of reducing the visual complexity of these graphs. However, we have demonstrated a method of finding contextual subgraphs around key nodes for graphs. We are not guaranteed that these subgraphs will be easy to visualize, but by restricting the parameters a and r in Algorithm 1, we will likely be able to find subgraphs with effective two-dimensional visualizations.

Additionally, we have put forth several metrics and a layout algorithm that could feasibly be used to aid in the visualization process. The methods, while not exhaustive, serve alongside the method of finding contextual subgraphs as a launching point for visualization and for new metrics.

Users

As we have mentioned, there is no one successful solution to this problem. Additionally, judging the success of a solution is nontrivial. As with any problem of perception or human-computer interaction, the success of the end result will vary from person to person. In implementing any of our visualization strategies in larger systems—be they debuggers or otherwise—designers should take into account the specific needs of users.

8.1 Future work

Our work was limited in scope, but there are several avenues in which this research could be expanded.

8.1.1 Parallel debuggers

An obvious extension of this research would be to create a better parallel debugger. Harnessing and honing the methods and metrics that we used in this research could contribute to the creation of a parallel program debugger that shows ellipses-based visualizations of complicated process communication graphs.

We can induce the communication graph of a parallel program given its post-mortem log file. By visualizing the ensuing graph, we can learn a few things: first, we

can see what the program's communication looks like around our focus, the process in question. Second, we can ensure that the entire graph appears as it should given the implemented algorithms. Programmers should have a good idea of the "shape" of the algorithms that they implement, or how the processes are meant to communicate. Books on parallel algorithms tend to include pictures of algorithm topologies, often with ellipses included. By matching the log-based visualization with the actual topology, the programmer can quickly decide whether there are serious bugs in the program's communication structure.

8.1.2 Identifying graph topologies

In order for parallel debuggers to successfully use our visualization techniques, they must have some way to detect graph topologies. Without knowing the topology, we would be unable to use the properties that have been grown with the graph, thus significantly hampering the usefulness of our visualization techniques. We have seen an example of a debugger that uses simple, predefined mesh and torus topologies [28] and a method for debugging that forces the programmer to provide a topological specification for the program [29]. Neither of these is optimal, as the former method relies on simple (and small) graphs, and the latter method forces the programmer to do a lot of extra work.

In our work, we have grown useful graphs using grammars and special procedures. In order to achieve the identification task at hand, we hope to find an inverse of our grammars and procedures: an algorithm that can effectively take a large, unidentified graph and learn the productions that would create it. While this task seems daunting, our restriction of these graphs to those that appear commonly in parallel programming should prove to be helpful. The programmer may also be able to provide a "seed" that aids the algorithm in finding the graph's appropriate origin. For example, if the programmer knows that his program implements a shuffle network, he could tell the debugger to search for graph topologies based on the idea of permutation. Once we have identified how a graph was grown, we can grow the properties with it, and proceed with our visualization strategies.

One possible stumbling point may be that if the parallel program contains communication bugs, then the topology of the induced communication graph will not mimic the programmer's desired topology. There are two ways to handle this problem. First, we can ignore it and identify the graph topology as programmed. By visualizing this

topology, the programmer should be able to spot the communication errors that cause the program to deviate from the desired algorithms. Alternatively, we could attempt to find a fault-tolerant means of topology identification that corrects for the communication error and shows the user a guess at the correct topology, identifying the areas where the program deviates from the displayed topology. The latter method sounds significantly more difficult, and would likely come as an extension to the former.

The work in this thesis should provide a sound basis for pursuing any of these extensions. Parallel programmers *will* want to see effective visualizations of their designs, and we have provided foundations toward that goal.

Appendix A

Large figures

Figure A.1: An example of a simple clog file. This is from a program that approximates π, run with only 2 processes.

```
GUI_LIBDIR is set. GUI_LIBDIR = /home/cs-students/ssr2/mpich2-install/lib
CLOG-02.44
is_big_endian=true
is_finalzed=true
block_size=65536
num_buffered_blocks=128
max_comm_world_size=2
max_thread_count=1
known_eventID_start=0
user_eventID_start=600
known_solo_eventID_start=-10
user_solo_eventID_start=5000
known_stateID_count=300
user_stateID_count=0
known_solo_eventID_count=2
user_solo_eventID_count=0
commtable_fptr=66560
```

```
RecHeader[ time=1.9073486328125E-6, icomm=0, rank=1, thread=0, rectype=11 ]
RecTshift[ timeshift=-0.0 ]
RecHeader[ time=1.9073486328125E-6, icomm=0, rank=0, thread=0, rectype=11 ]
RecTshift[ timeshift=-0.0 ]
RecHeader[ time=7.867813110351562E-6, icomm=0, rank=1, thread=0, rectype=9 ]
RecComm[ etype=CommWorldCreate, icomm=0, rank=1, wrank=1,
 gcomm=2127016245-1.304385520082285E9-siri ]
RecHeader[ time=7.867813110351562E-6, icomm=0, rank=1, thread=0, rectype=9 ]
RecComm[ etype=CommSelfCreate, icomm=2, rank=0, wrank=1,
 gcomm=256982527-1.304385520084197E9-siri ]
RecHeader[ time=8.821487426757812E-6, icomm=0, rank=0, thread=0, rectype=9 ]
RecComm[ etype=CommWorldCreate, icomm=0, rank=0, wrank=0,
 gcomm=2127016245-1.304385520082285E9-siri ]
RecHeader[ time=8.821487426757812E-6, icomm=0, rank=0, thread=0, rectype=9 ]
RecComm[ etype=CommSelfCreate, icomm=1, rank=0, wrank=0,
 gcomm=341175986-1.304385520082293E9-siri ]
RecHeader[ time=8.821487426757812E-6, icomm=0, rank=0, thread=0, rectype=4 ]
RecDefConst[ etype=-201, value=-1, name=MPI_PROC_NULL ]
RecHeader[ time=1.1920928955078125E-5, icomm=0, rank=0, thread=0, rectype=4 ]
RecDefConst[ etype=-201, value=-2, name=MPI_ANY_SOURCE ]
RecHeader[ time=1.1920928955078125E-5, icomm=0, rank=0, thread=0, rectype=4 ]
RecDefConst[ etype=-201, value=-1, name=MPI_ANY_TAG ]
RecHeader[ time=4.482269287109375E-5, icomm=0, rank=0, thread=0, rectype=5 ]
RecBare[ etype=54 ]
RecHeader[ time=4.482269287109375E-5, icomm=0, rank=0, thread=0, rectype=5 ]
RecBare[ etype=55 ]
RecHeader[ time=4.601478576660156E-5, icomm=0, rank=1, thread=0, rectype=5 ]
RecBare[ etype=54 ]
RecHeader[ time=4.696846008300781E-5, icomm=0, rank=1, thread=0, rectype=5 ]
RecBare[ etype=55 ]
RecHeader[ time=4.696846008300781E-5, icomm=0, rank=0, thread=0, rectype=5 ]
RecBare[ etype=48 ]
RecHeader[ time=4.696846008300781E-5, icomm=0, rank=0, thread=0, rectype=5 ]
RecBare[ etype=49 ]
```

```
RecHeader[ time=4.792213439941406E-5, icomm=0, rank=1, thread=0, rectype=5 ]
RecBare[ etype=48 ]
RecHeader[ time=4.792213439941406E-5, icomm=0, rank=1, thread=0, rectype=5 ]
RecBare[ etype=49 ]
RecHeader[ time=4.887580871582031E-5, icomm=0, rank=1, thread=0, rectype=5 ]
RecBare[ etype=12 ]
RecHeader[ time=20.354533910751343, icomm=0, rank=0, thread=0, rectype=5 ]
RecBare[ etype=12 ]
RecHeader[ time=20.354538917541504, icomm=0, rank=0, thread=0, rectype=5 ]
RecBare[ etype=13 ]
RecHeader[ time=20.354540824890137, icomm=0, rank=0, thread=0, rectype=5 ]
RecBare[ etype=24 ]
RecHeader[ time=20.35454487800598, icomm=0, rank=1, thread=0, rectype=5 ]
RecBare[ etype=13 ]
RecHeader[ time=20.354549884796143, icomm=0, rank=1, thread=0, rectype=5 ]
RecBare[ etype=24 ]
RecHeader[ time=20.354556798934937, icomm=0, rank=1, thread=0, rectype=5 ]
RecBare[ etype=25 ]
RecHeader[ time=20.354556798934937, icomm=0, rank=1, thread=0, rectype=5 ]
RecBare[ etype=12 ]
RecHeader[ time=20.354556798934937, icomm=0, rank=0, thread=0, rectype=5 ]
RecBare[ etype=25 ]
RecHeader[ time=22.2727370262146, icomm=0, rank=0, thread=0, rectype=5 ]
RecBare[ etype=12 ]
RecHeader[ time=22.27273988723755, icomm=0, rank=0, thread=0, rectype=5 ]
RecBare[ etype=13 ]
RecHeader[ time=22.272742986679077, icomm=0, rank=1, thread=0, rectype=5 ]
RecBare[ etype=13 ]
RecHeader[ time=22.272742986679077, icomm=0, rank=0, thread=0, rectype=5 ]
RecBare[ etype=-9 ]
RecHeader[ time=22.272745847702026, icomm=0, rank=1, thread=0, rectype=5 ]
RecBare[ etype=-9 ]
RecHeader[ time=22.272774934768677, icomm=0, rank=0, thread=0, rectype=2 ]
RecDefState[ stateID=6, startetype=12, finaletype=13, color=cyan,
 name=MPI_Bcast, format=null ]
```

```
RecHeader[ time=22.272775888442993, icomm=0, rank=0, thread=0, rectype=2 ]
RecDefState[ stateID=12, startetype=24, finaletype=25,
 color=MediumPurple, name=MPI_Reduce, format=null ]
RecHeader[ time=22.272775888442993, icomm=0, rank=0, thread=0, rectype=2 ]
RecDefState[ stateID=24, startetype=48, finaletype=49, color=white,
 name=MPI_Comm_rank, format=null ]
RecHeader[ time=22.27277684211731, icomm=0, rank=0, thread=0, rectype=2 ]
RecDefState[ stateID=27, startetype=54, finaletype=55, color=white,
 name=MPI_Comm_size, format=null ]
RecHeader[ time=22.272777795791626, icomm=0, rank=0, thread=0, rectype=3 ]
RecDefEvent[ etype=-9, color=orange, name=MPE_Comm_finalize, format=null ]
RecHeader[ time=22.272789001464844, icomm=0, rank=0, thread=0, rectype=2 ]
RecDefState[ stateID=280, startetype=560, finaletype=561,
color=maroon, name=CLOG_Buffer_write2disk, format=null ]
RecHeader[ time=1.0E8, icomm=0, rank=0, thread=0, rectype=0 ]
End Of File
Total ByteSize of the logfile = 2080
```

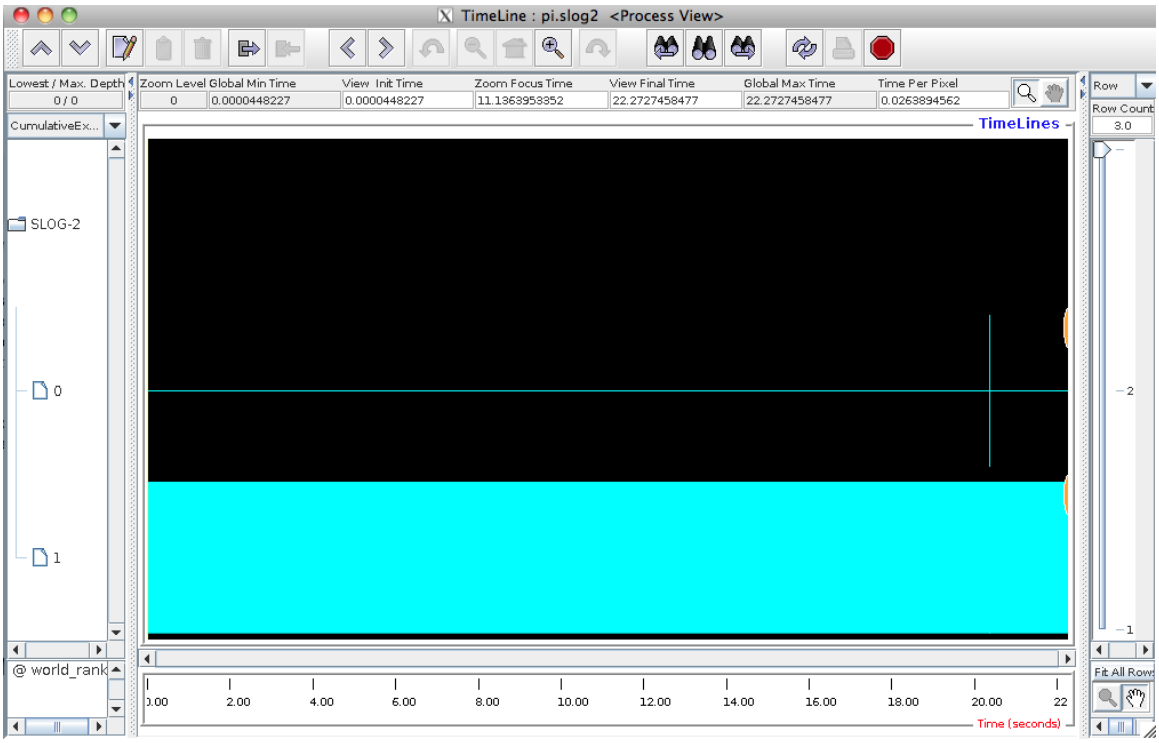


Figure A.2: A screen capture from jumpshot, showing the logfile in Figure A.1

Bibliography

[1] Agrawala, M., Phan, D., Heiser, J., Haymaker, J., Klingner, J., Hanrahan, P., and Tversky, B. Designing effective step-by-step assembly instructions. *ACM Transactions on Graphics (TOG) 22*, 3 (2003), 828–837.

[2] Agrawala, M., and Stolte, C. Rendering effective route maps: improving usability through generalization. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2001), SIGGRAPH '01, ACM, pp. 241–249.

[3] Akl, S. *Parallel sorting algorithms*. Academic Pr, 1985.

[4] Auber, D., Delest, M., Domenger, J. P., Duchon, P., and Fdou, J. M. New strahler numbers for rooted plane trees. In *Vienna University of Technology, Birkhauser* (2004), pp. 203–215.

[5] Bailey, D. A., Cuny, J. E., and Loomis, C. P. Paragraph: Graph editor support for parallel programming environments. *International Journal of Parallel Programming 19* (1990), 75–110. 10.1007/BF01407832.

[6] Bastian, M., Heymann, S., and Jacomy, M. Gephi: An open source software for exploring and manipulating networks. In *Proceedings of the International AAAI Conference on Weblogs and Social Media* (2009), ICWSM '09.

[7] Battista, G. D., Eades, P., Tamassia, R., and Tollis, I. G. *Graph Drawing: Algorithms for the Visualization of Graphs*, 1st ed. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1998.

[8] Brodlie, K. W., Duke, D. J., (editors), K. I. J., Beermann, D., Munzner, T., and Humphreys, G. Scalable, robust visualization of very large trees, 2005.

[9] Carlsson, G., Cruthirds, J., Sexton, H., and Wright, C. Interconnection networks based on a generalization of cube-connected cycles. *IEEE Transactions on Computers 34* (1985), 769–772.

[10] Ce, Y., Zhen, X., Ji-zhou, S., Xiao-jing, M., Yan-yan, H., and Huabei, W. Paramodel: a visual modeling and code skeleton generation system for programming parallel applications. *SIGPLAN Not. 43*, 4 (2008), 4–10.

[11] Chan, A., Gropp, B., and Lusk, R. Performance visualization for parallel programs: Viewers, May 2011. `http://www.mcs.anl.gov/research/projects/perfvis/software/viewers/index.htm`.

[12] Cuny, J., Forman, G., Hough, A., Kundu, J., Lin, C., Snyder, L., and Stemple, D. The ariadne debugger: scalable application of event-based abstraction. In *In Proc. ACM SIGPLAN/ONR Workshop on Parallel and Distributed Debugging, ACM SIGPLAN Notices* (1993), pp. 85–95.

[13] Dongarra, J., Gannon, D., Fox, G., and Kennedy, K. The impact of multicore on computational science software, 2007.

[14] Dongarra, J., Otto, S., Snir, M., and Walker, D. An introduction to the MPI standard. *Communications of the ACM* (1995).

[15] Efe, K. Embedding large complete binary trees in hypercubes with load balancing, 1996.

[16] Feng, T. A survey of interconnection networks. *Computer 14*, 12 (1981), 12–27.

[17] Furnas, G. W. Generalized fisheye views. *SIGCHI Bull. 17*, 4 (1986), 16–23.

[18] Furnas, G. W. A fisheye follow-up: further reflections on focus + context. In *Proceedings of the SIGCHI conference on Human Factors in computing systems* (New York, NY, USA, 2006), CHI '06, ACM, pp. 999–1008.

[19] Gait, J. A probe effect in concurrent programs. *Softw. Pract. Exper. 16* (March 1986), 225–233.

[20] Gibbons, A., and Rytter, W. *Efficient parallel algorithms*. Cambridge Univ Pr, 1989.

[21] Herman, I., Delest, M., and Melanon, G. Tree visualisation and navigation clues for information visualisation, 1998.

[22] Herman, I., Marshall, M. S., Melancon, H. M., Duke, D. J., Delest, M., and p. Domenger, J. Skeletal images as visual cues in graph visualization. In *in Data Visualization '99, Proceedings of the Joint Eurographics and IEEE TCVG Symposium on Visualization* (1999), Springer–Verlag, pp. 13–22.

[23] Herman, I., Melancon, G., and Marshall, M. S. Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics 6* (2000), 24–43.

[24] Himsolt, M. Gml: A portable graph file format, April 2011. `http://www.fim.uni-passau.de/fileadmin/files/lehrstuhl/brandenburg/projekte/gml/gml-technical-report.pdf`.

[25] Hu, Y. Efficient, high-quality force-directed graph drawing. *Mathematica Journal 10*, 1 (2005), 37–71.

[26] Huang, X., and Lai, W. Clustering graphs for visualization via node similarities. *J. Vis. Lang. Comput. 17*, 3 (2006), 225–253.

[27] Huband, S., and McDonald, C. Debugging parallel programs using incomplete information. *Cluster Computing, International Workshop on 0* (1999), 278.

[28] Huband, S., and McDonald, C. Depict: A topology-based debugger for mpi programs. In *High-Level Parallel Programming Models and Supportive Environments*, F. Mueller, Ed., vol. 2026 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2001, pp. 109–121. 10.1007/3-540-45401-2_9.

[29] Huband, S., and McDonald, C. Parallel program debugging by specification: Research articles. *Concurr. Comput. : Pract. Exper. 16* (May 2004), 551–585.

[30] Janecek, P., and Pu, P. A framework for designing fisheye views to support multiple semantic contexts. In *AVI '02: Proceedings of the Working Conference on Advanced Visual Interfaces* (New York, NY, USA, 2002), ACM, pp. 51–58.

[31] Kimelman, D., Leban, B., Roth, T., and Zernik, D. Reduction of visual complexity in dynamic graphs. In *Graph Drawing*, R. Tamassia and I. Tollis, Eds., vol. 894 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1995, pp. 218–225. 10.1007/3-540-58950-3_373.

[32] Krammer, B., Bidmon, K., Müller, M. S., and Resch, M. M. Marmot: An mpi analysis and checking tool, 2004.

[33] Kranzlmüller, D., Grabner, S., and Volkert, J. Event graph visualization for debugging large applications. In *SPDT '96: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools* (New York, NY, USA, 1996), ACM, pp. 108–117.

[34] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (1978), 558–565.

[35] Mehra, R., Zhou, Q., Long, J., Sheffer, A., Gooch, A., and Mitra, N. Abstraction of man-made shapes. *ACM Transactions on Graphics (TOG) 28*, 5 (2009), 137.

[36] Melanon, G., and Herman, I. Dag drawing from an information visualization perspective. In *Data Visualization '00* (1999), Springer-Verlag, pp. 3–12.

[37] Minnick, L. Generalized forcing in aperiodic tilings. Undergraduate thesis at Williams College, 1998.

[38] Pauly, M., Mitra, N., Wallner, J., Pottmann, H., and Guibas, L. Discovering structural regularity in 3D geometry. *ACM Transactions on Graphics (TOG) 27*, 3 (2008), 1–11.

[39] Prusinkiewicz, P., and Lindenmayer, A. *The Algorithmic Beauty of Plants (The Virtual Laboratory)*. Springer, Oct. 1991.

[40] Quigley, A., and Eades, P. Fade: Graph drawing, clustering, and visual abstraction. In *Graph Drawing*, J. Marks, Ed., vol. 1984 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2001, pp. 77–80. 10.1007/3-540-44541-2_19.

[41] Quinn, M. *Designing efficient algorithms for parallel computers*. 1986.

[42] Rozenberg, G., and Salomaa, A. *Mathematical Theory of L Systems*. Academic Press, Inc., Orlando, FL, USA, 1980.

[43] Saad, Y., and Schultz, M. Topological properties of hypercubes. *IEEE Transactions on Computers 37* (1988), 867–872.

[44] Sarkar, M., and Brown, M. H. Graphical fisheye views of graphs. In *CHI '92: Proceedings of the SIGCHI conference on Human factors in computing systems* (New York, NY, USA, 1992), ACM, pp. 83–91.

[45] Shafi, A., Carpenter, B., and Baker, M. Nested parallelism for multi-core hpc systems using java. *Journal of Parallel and Distributed Computing 69*, 6 (2009), 532 – 545.

[46] Sipser, M. *Introduction to the Theory of Computation*. International Thomson Publishing, 1996.

[47] Wilson, K. Problems in physics with many scales of length. *Scientific American 241*, 2 (1979), 158–179.

[48] Wu, A. Y. Embedding of tree networks into hypercubes. *Journal of Parallel and Distributed Computing 2*, 3 (1985), 238 – 249.