Parallel Process Distribution
of Robotic Software Applications

Nicholas Anthony Degani
Advisor: Arvind Krishnamurthy
May 11, 2003

We discuss the importance of parallel process distribution across modern networks and general work done in the area. We explore several algorithms for graph partitioning, including breadth first searches, spectral bisection and the Kernighan/Lin Algorithm. We discuss the Porter program and Professor Scassellati’s robotic software application network that formed the concrete link for this research.

We modify the Kernighan/Lin algorithm to improve graphs with weighted nodes and weighted edges. We explore the use of functionally weighted graphs, and a possible method for partitioning them using the modified Kernighan/Lin algorithm. We discuss the need to generalize the process of mapping one graph onto another, since processor graphs do not always display a regular topology, and present an algorithm for generalize graph-to-graph mapping based on the modified Kernighan/Lin algorithm.

Finally, we discuss the implications of the work, and future work that could be done in the area of parallel process distribution.

1 Introduction

Parallel processing is a constantly changing field, as new hardware creates new paradigms for study. At the birth of the field, manufacturers created specialized hardware pre-wired to execute processes in parallel, and thus the field largely focused on hardware design and implementation. As the field matured, more venues have opened themselves to those interested in parallel architectures; researchers have connected processors directly to each other with shared memory resources, the cluster model of networked IBM clones has risen in popularity, and even the inter-connected channels of the World Wide Web have become a subject of interest to students of parallel processing in the form of distributed computing systems. Now students of Computer Science are faced with how best to utilize these parallel resources, and particularly how to distribute processes across these resources.
If a programmer has a set of related processes he’d like to run on a machine, how can he best speed up the output of these processes by using a parallel processors? What would be the optimal number of processors to use in such a machine, and how should they be connected? How should the processes be distributed across the processors? Also, how would any of these solutions change if fundamental parameters relating to the interrelated processes changed? For example, if the amount of information communicated from Process A to Process B doubled (and so on down the line), how would these solutions change?

We seek to explore the theory and implications of parallel process distribution as highlighted in these questions. Particularly, we draw on an existing problem, how best to distribute robotic software applications across a particular parallel processor network, to confront the potentials of parallel process distribution and to examine various algorithms from the context of real-life problems and solutions.

2 Background

We have conducted our research by attempting to balance theory with reality. The major background necessary for this project then lies in both theory and application. The theoretical underpinning of the project is the much-developed field of graph partitioning. The application that grounds this project in reality is Professor Scassellati’s network of parallel processors used to execute his robotic software applications.

2.1 Graph Partitioning Theory

Graph partitioning is a well-studied field within computer science, with several algorithms for finding a good solution to dividing a graph into roughly equal parts while minimizing the values of the edges cut.

Generally we would like to be able to divide a graph into as many sections as we would like while balancing the constraints. In reality, the majority of partitions are done by repeatedly bisecting the graph into two equal pieces. This reduces the calculation cost at each level, provides an incentive to parallelize all bisections after the first, and makes the overall problem manageable.

There are three methods primarily used in bisecting a graph without nodal coordinates: the breadth first search method, the spectral bisection method, and the Kernighan/Lin algorithm. While all three methods provide valuable insight into the problem, Kernighan/Lin presents the best opportunity for expansion and exploration of the current problem.

2.1.1 The Breadth First Search Method

One method for dividing a graph into \(2^n\) equal parts is using a breadth first search technique to build a tree. Choosing an arbitrary node as a starting point, execute a breadth first search to build a tree, assigning a level to each
node based on its distance from the root. Since a breadth first search performs an exhaustive search on adjacent nodes at each level, there should never be more than one level of distance between any two adjacent nodes.

Once a BFS tree has been constructed from a graph, the graph can be bifurcated between two levels, such that the value of the nodes at levels above the bifurcation roughly equals the value of the nodes at levels below the bifurcation. Thus \( G = G_1 \cup G_2 \) and \( |G_1| \approx |G_2| \).

Note that the breadth first search method does not take into account the weights of the edges it cuts, nor does it provide a method for improving upon the original solution, thus restricting its value in studying the distribution of software applications across a parallel network.

### 2.1.2 The Spectral Bisection Method

The spectral bisection method works first by finding the Laplacian matrix of a graph, \( L(G) \). The Laplacian matrix marks the connections between nodes, such that \( L(i,j) = -1 \) if \( i \neq j \) and \( i \) is adjacent to \( j \). Also, \( L(i,i) = \) the total number of connections between \( i \) and any other node. Thus the Laplacian matrix is both symmetric, and the values of all rows and columns add up to zero.

It is known that the second eigenvalue of the Laplacian matrix of a graph, \( \lambda_2(L(G)) \), is the algebraic connectivity of graph \( G \). Thus the vector \( v_2 \) corresponding to \( \lambda_2(L(G)) \) is the eigenvector of algebraic connectivity, and has special properties. Most notably for our purposes, we can divide a graph into two groups with the minimal amount of connectivity between the groups if we sort using the following scheme:

- If \( v_2(n) < 0 \), place \( n \) in \( G_1 \).
- If not, place \( n \) in \( G_2 \).

While the spectral bisection method works better than a breadth first search at evenly dividing a graph, its inner-working are too complex to adapt to a situation with weighted nodes and edges. For a further discussion of the spectral bisection method and the vibrating string theory it is based on, see Demmel [2].

### 2.1.3 The Kernighan/Lin Algorithm

The Kernighan/Lin Algorithm addresses the most obvious deficiency of the breadth first search method by providing a method to improve upon an initial allocation of nodes between two subgraphs.

Given a graph \( G = (N,E,W_E) \) with nodes and weighted edges, perform the BFS method to obtain an initial partitioning of the graph such that \( G = G_1 \cup G_2 \) and \( |G_1| = |G_2| \), in which \( |G| \) is the number of nodes in \( G \). Now let \( C = cost(G_1, G_2) = \sum W_E \forall E(G_1, G_2) \), that is the cost of the partitioning is equal to the weights of all the edges that cross the partitioning. We seek to minimize \( C \) for a given \( G \).

To do so, let \( X \) be a subset of nodes of \( G_1 \) and \( Y \) be a subset of nodes in \( G_2 \), such that \( |X| = |Y| \). If we were to switch which subgraphs \( X \) and \( Y \)
are allocated to, we would not change the number of nodes in each of the two subgraphs. However, we could then calculate a new cost of partitioning with \((G_1 - X) \cup Y\) and \((G_2 - Y) \cup X\): if the cost of the new subgraphs is less than the cost of the old subgraph, then we should accept the new subgraphs in place of the old subgraphs.

The trick of the Kernighan/Lin algorithm is efficiently finding subsets of nodes \(X\) and \(Y\) to swap. Let \(Ex(n)\) equal the external cost of leaving node \(n\) in subgraph \(G_1\) (i.e. \(\sum W_{E \in E(n, G_2)}\)) and \(In(n)\) equal the internal savings of leaving node \(n\) in subgraph \(G_1\) (i.e. \(\sum W_{E \in E(n, G_1)}\)). The value of switching node \(n\) into subgraph \(G_2\) is \(D(n) = Ex(n) - In(n)\). \(D(n)\) can be similarly calculated for all nodes in \(G_2\).

With these \(D(n)\) values assigned to each node, the comparison of two subgroups becomes simple. The value of switching two nodes \(X\) and \(Y\) between \(G_1\) and \(G_2\) is:

\[
gain(X, Y) = D(X) + D(Y) - 2 \times W_{E(X, Y)}.
\]

Note that since \(X\) and \(Y\) remain in different subgroups, the benefit of removing \(W_{E(X, Y)}\) disappears for switching.

The Kernighan/Lin Algorithm thus steps through the problem of improving a partitioning as follows:

\[
\begin{align*}
\text{The Kernighan/Lin Algorithm:} & \\
\text{Compute cost}(G_1, G_2) & \quad \text{for initial partition.} \\
\text{Do} & \\
& \text{Compute } D(n) \text{ for all } n \text{ in graph.} & \quad \text{O}(|N|^2) \\
& \text{Unmark all nodes in the graph.} & \quad \text{O}(|N|) \\
& \text{While(Unmarked nodes exist)} \{ & \quad |N|/2 \text{ iterations} \\
& & \quad \text{Find two unmarked nodes } X \text{ and } Y, \\
& & \quad \text{that maximizes } gain(X, Y). \\
& & \quad \text{Add } X, Y \text{ and } gain(X, Y) \text{ to ordered list.} \\
& & \quad \text{Mark nodes } X \text{ and } Y. \\
& & \quad \text{Update } D(n) \text{ for all unmarked nodes} \\
& & \quad \text{as if } X \text{ and } Y \text{ had switched.} \\
& \} & \quad \text{O}(1) \\
& \text{Pick } j \text{ maximizing Gain, the sum of the first } j \text{ gains} & \quad \text{O}(|N|) \\
& \text{on the ordered list.} & \\
& \text{If Gain > 0 \{ & \\
& & \quad \text{Update } G_1 = G_1 - X + Y. \\
& & \quad \text{Update } G_2 = G_2 - Y + X. \\
& & \quad \text{Update } cost(G_1, G_2) = cost_{old}(G_1, G_2) - Gain. \\
& \} & \quad \text{O}(1) \\
& \} \quad \text{While (Gain > 0)}
\end{align*}
\]

Note that the slowest part of the Kernighan/Lin Algorithm is the discovery of unmarked nodes that maximize the \(gain(X, Y)\). Also note that the Kernighan/Lin Algorithm is the basis for much of my later explorations.

\(^{1}\)Adapted from Demmel [2].
2.2 A Sample Network for Robotic Software Applications

In sharp contrast to the abstracted concepts of theory lies the network of parallel processors that Professor Scassellati uses in his robotic software applications. Examining the configuration of Professor Scassellati’s network and the implementation of interprocess communication provides a useful foundation for further examination of the algorithms used in parallel process distribution.

2.2.1 Porter

Professor Scassellati has developed a set of real-time robotic software applications that communicate using a message-passing program known as Porter. Porter passes messages between named processes, which can exist on any processor; that is, Porter provides a layer of abstraction so that robotic software programmers do not need to worry about process distribution, whether processes are on the same or different processors.

Porter treats communicating processes like sockets, taking input and output ports and connecting them. It can connect multiple ports at once, allowing a process to send its output to several other processes.

An example of using Porter: We start up the cam process to receive visual input from what the robot is “seeing.” We then start up the motion process to examine two images and determine any difference between them; that is, the motion process can be used to see if the robot has moved or is moving by examining the differences between the images it “sees” over time. We could then link these two programs using Porter, such that the output images of the cam program are passed as messages to the motion program. The output of the motion program can then be “Portered” to another process, either to continue processing or to control the actions of the robot based on any movement perceived in its input images.

Porter can also disconnect processes that are currently receiving input and direct said input to another process. Doing so follows from Porter’s ability to connect or disconnect message-passing from processes already begun. Doing so would allow the dynamic allocation of processes across a network if doing so were required by the system architect.

2.2.2 The Physical Network

To speed the processing of the robotic software applications, Professor Scassellati has developed a network of independent, parallel processors interconnected in a complex network. The parallel network architecture consists of five sets of four processors. Within each set all four processors are interlinked with six bidirectional cables; each processor also directly accesses a switch which links the five sets.

Each of the processors runs QNX, is a real-time operating system with an extremely lightweight kernel. QNX is a variant of Unix, sacrificing generality and the bells and whistles of full-fledged Linux for the efficiency and speed of a lightweight operating system. QNX is generally best suited for embedded
systems that demand quick process switching, and thus is ideal for the heavy computing demands of real-time robotic software applications and the study of parallel process distribution.

In examining the distribution of processes over the network, note first that each process consists of code, several threads and data. Each process currently is confined within a single processor, which presents the opportunity for dividing each process over several processors as a project. However, such a task would be largely the implementation of well-known algorithms with little theoretical work. Instead, I choose to focus on the distribution of these processes across the various processors of the network, creating an algorithm to automate the initial placement of these processes. This approach directly addresses a weakness of the current situation, in that processes must currently be hand-placed on the network.

Another quirk in the current architecture is that several processes are drivers that directly interface with external devices. For example, the \texttt{cam} program interfaces with the “eye” of the robot, capturing whatever the robot sees and encoding it into the image object used by other processes. The \texttt{cam} program, since it interfaces with a device, must be located on the processor that is connected to said device. Another driver program, \texttt{gyro}, interfaces with a gyroscope that measures the robot’s tilt and helps it from falling. Like \texttt{cam}, \texttt{gyro} must be on the processor connected to the gyroscope.

Other than drivers, there are also several behavior programs that are not bound to any particular processor. These behaviors can and should be allocated to maximize the performance of the robotic software applications. It is the distribution of these unbound processes across the parallel processor network that becomes an interesting question.

For example, the \texttt{cam} driver for the robot’s “eye” is connected to one processor, while the input for a motor is connected to a different processor. Say we want to drive the motor so that the robot tries to steady itself and stop moving. We would would Porter the \texttt{cam} driver to behavior \texttt{motion}, which finds the difference between the previous image and the current image, i.e. it analyzes a stream of images for motion. We then Porter the output of \texttt{motion} to a new behavior that creates messages to move the robot in the opposite direction of the motion. Finally, we would Porter this program to the \texttt{motor} driver so that the robot could respond to the commands generated by my intermediate program. Since the two driver programs, \texttt{cam} and \texttt{motor} are bound to certain processors, we cannot decide where to place them. However, we can choose where to place the two behaviors, \texttt{motion} and the new behavior, on the network; it is the optimal placement of these behaviors that we seek to automate.
3 Theory and Application of Process Distribution

There are several possible directions to explore in examining process distribution. First, we expand the Kernighan/Lin algorithm to account for graphs with unequally weighted nodes. Next, we examine the implications on graph partitioning theory of basing communication costs and processing requirements on an independent variable. Finally, we generalize the modified Kernighan/Lin algorithm to map a process graph onto an irregular physical processor graph.

3.1 Graph Partitioning with Unequally Weighted Nodes

The Kernighan/Lin Algorithm can be adapted to address graphs with weighted nodes, in addition to weighted edges.

Given a graph \(G = (N, E, W_N, W_E)\) with weighted nodes and edges, perform the BFS method to obtain an initial partitioning of the graph such that \(G = G_1 \cup G_2\) and \(|G_1| \approx |G_2|\), in which \(|G| = \sum W_N \forall N \in G\). Now let \(C = \text{cost}(G_1, G_2) = \sum W_E \forall E(G_1, G_2) + k \times ||G_1| - |G_2||\), that is the cost of the partitioning is equal to the weights of all the edges that cross the partitioning plus \(k\) times the difference in node weights of the two graphs. \(k\) is a factor corresponding to the importance of balancing the weights between the graphs. We seek to minimize \(C\) for a given \(G\) and \(k\).

To do so, let \(X\) be a subset of nodes of \(G_1\) and \(Y\) be a subset of nodes in \(G_2\), such that \(|X| \approx |Y|\). If we were to switch which subgraphs \(X\) and \(Y\) are allocated to, we would not change the value equality of the two subgraphs. However, we could then calculate a new cost of partitioning with \((G_1 - X) \cup Y\) and \((G_2 - Y) \cup X\); if the cost of the new subgraphs is less than the cost of the old subgraph, then we should accept the new subgraphs in place of the old subgraphs.

The trick of the Kernighan/Lin algorithm is efficiently finding subsets of nodes \(X\) and \(Y\) to swap. Let \(E_x(n)\) equal the external cost of leaving node \(n\) in subgraph \(G_1\) (i.e. \(\sum W_E \forall E(n, G_2)\)) and \(I_n(n)\) equal the internal savings of leaving node \(n\) in subgraph \(G_1\) (i.e. \(\sum W_E \forall E(n, G_1)\)). Also let \(D_i(n)\) equal the change in relative node weights between \(G_1\) and \(G_2\) (i.e. \(||G_1| - |G_2|| - ||G_1| - |G_2| - 2 \times n||\)). The value of switching node \(n\) from subgraph \(G_1\) into subgraph \(G_2\) is \(D(n) = E_x(n) - I_n(n) + k \times D_i(n)\). \(D(n)\) can be similarly calculated for all nodes in \(G_2\).

A variant of the Kernighan/Lin Algorithm can then be used to step through the problem of improving a partitioning. Note that the the subsets \(X\) and \(Y\) to be interchanged are the nodes found in \(G_1\) that should switch to \(G_2\) and vice versa to maximize the Gain of the switch.
The Modified Kernighan/Lin Algorithm

Compute \( \text{cost}(G_1, G_2) \) for initial partition. 

\[ O(|N|^2) \]

Do \{ 

Compute \( D(n) \) for all \( n \) in graph. 

\[ O(|N|^2) \]

Unmark all nodes in the graph. 

\[ O(|N|) \]

While(Unmarked nodes exist) \{ 

Find one unmarked node \( X \) that maximizes \( D(X) \). 

\[ O(|N|) \]

Add \( X \) and \( D(X) \) to ordered list. 

\[ O(1) \]

Mark \( X \). 

\[ O(1) \]

Update \( D(n) \) for all unmarked nodes as if \( X \) had switched. 

\[ O(|N|) \]

\}

Pick \( j \) maximizing Gain, the sum of the first \( j \) gains on the ordered list. 

\[ O(|N|) \]

If Gain > 0 \{ 

Update \( G_1 \) and \( G_2 \) based on which nodes switch. 

\[ O(|N|) \]

Update \( \text{cost}(G_1, G_2) = \text{cost}_{\text{old}}(G_1, G_2) - \text{Gain} \). 

\[ O(1) \]

\} 

While (Gain > 0)

Note that the slowest part of the Kernighan/Lin Algorithm has been substantially modified in this variant. Since ensuring an equal weight between the two graphs has been incorporated into the definition of \( D(n) \), we only need to find one node at a time, making the algorithm run in \( O(|N|^2) \) and not \( O(|N|^3) \).

Note that this modified Kernighan/Lin algorithm can be used to map networks of processes onto regular, interconnected networks of processors, like the sample processor network shown in Figure 1. Doing so assumes that all the processors are the same speed, and that all processors should be used.

3.2 Graph Partitioning with Functional Weights

Another interesting situation to examine in graph partitioning is the case in which the weights of edges and nodes is no longer preset; instead, let these values depend on some independent variables. Thus each node and edge would have an associated function, relating the value of the independent variables to the weight. While each independent variable would be set before the actual start of the processes, it would be profitable to determine the best average division of nodes, so that the computations would not need to be redone each new time the variable changes.

This model reflects the possibility of adjusting certain input parameters of a system of robotic software applications. For example, if the programmer decided to double the resolution of the images created by \texttt{cam}, how can our algorithm account for this change?

Under this model, we shall first simplify the discussion by limiting our graph to one independent variable. Therefore, the weight of the first node would be
represented as $W_{N1}(x)$, denoting that $N1$’s weight depends on the independent variable $x$; similarly, $W_{E1}(x)$ represents that first edge’s weight.

To further simplify the model, we shall assume that all functions are linear, and can be modelled as $W_{N1}(x) = a_{N1} \cdot x + b_{N1}$ or $W_{E1}(x) = a_{E1} \cdot x + b_{E1}$. Thus, given a graph $G = (N, E, W_N(x), W_E(x))$ with functionally weighted nodes and edges, how do we bisect it into two parts while minimizing the expected cost?

To ensure that the bisected graph will perform well at the extremes of the independent variable, we first partition the graph twice – once at an extremely low $x$ value and once at an extremely high $x$ value. To approximate the lower extreme, set $x = 0$, thus creating a graph of $b$ values wholly independent of the independent variable. Now run the modified Kernighan/Lin algorithm to determine the optimal partition of nodes assuming an extremely weak independent variable. To approximate the upper extreme, set $x = \infty$, i.e. create a graph wholly based on the $a$ variables. After running the modified Kernighan/Lin algorithm on this “high-value” graph, we have two distinct partitions of the original graph representing the ideal partitions for the extreme values of $x$.

Without any further information about $x$, the best we can then do is declare certain nodes to be “partitionally certain.” Comparing the two extreme partitions, any nodes partitioned into the same subgraph both times can be considered “partitionally certain,” i.e. in any reasonable partition of the functionally weighted graph those nodes will appear in that subgraph. In other words, if $N1$ is allocated into $G_1$ for extremely low $x$ values and into $G_1$ for extremely high $x$ values, then it is reasonably certain that in any good partition of the functionally weighted graph, $N_1$ will appear in $G_1$.

Given a list of “partitionally certain” nodes is tremendously valuable as it can greatly reduce the calculations required to calculate a full partitioning of
the graph once $x$ is determined. If $x$ is given some discrete value, or given an expected value, we can then calculate an optimal graph that accounts for extreme values of $x$. To do so, we merely evaluate the weight functions for each node and edge with the determined value of $x$. Then we choose an initial partition of $G$ such that the “partitionally certain” nodes are allocated to the correct subgraphs. Next, run the modified Kernighan/Lin algorithm on the graph; however, mark all the “partitionally certain” nodes so that they cannot be considered for side-switching.

Using the concept of “partitionally certain” nodes then allows two innovations for functionally weighted graphs in which $x$ has some expected value. First, it provides a stable allocation of at least half the nodes as a foundation for further analysis, reducing the computation time for on-the-spot partitioning by one-half (since the number of unmarked nodes is cut in half, halving the length of the loop). Second, using “partitionally certain” nodes ensures that modifying $x$ from its expected value towards the extremes will have less impact than otherwise, since the preferences of the extremes will have been incorporated into the distribution of the “partitionally certain” nodes.

### 3.3 Mapping Graphs onto Irregular Graphs

Computing costs and constructing algorithms becomes far more difficult in cases where the physical network is not regular. That is, mapping a network of processes onto a real network of processors is often not a simple partitioning problem. Instead, the distributor must account for the unique configuration of the processor network, and pay heed to the communication costs and processor speeds. Parallel process distribution becomes understandably more difficult in such a situation.

To better understand the process of mapping a network of processes onto processor networks, figures 2 and 3 present sample process networks for our analysis. Figure 2 presents the most simple case of an input process passing its information to a motion detector behavior, which then forwards its results to an output motor process. Figure 3 is far more complex, with two input processes connecting to three behaviors, which then feed three output programs.

![Figure 2: A simple set of processes.](image)

A mapping, then, consists of associating each of the processes of the target graph with a processor on the physical network. Mapping processes as such creates a blue-print for designers in distributing their parallel processes. Addi-
tionally, it should be possible to automate the taking of the results of a mapping and actually starting programs to correspond to those results. Figure 4 illustrates a sample mapping of Figure 2’s process network onto the physical layout of Figure 1. Since Figure 1 is a portion of the real network utilized by Professor Scassellati, it should be possible to automate the implementation of this distribution pattern.

The algorithm to map one set of processes onto a set of processors is based on the modified Kernighan/Lin algorithm. First note that both the process network and the processor network must be weighted, such that the weights on the process network represent the amount of data to be processed or communicated,
whereas the weights of the processor network represent the speed of processing and speed of communication.

Assuming that there is a constant stream of data through the process network and that processes have equal priority, we seek to minimize the processing time of the data. The relative processing time for any individual processor can be approximated by summing the weights of the processes allocated to that processor and dividing by the relative speed of the processor. We can assume that communicating processes on the same processor have negligible communication speed; however, inter-processor communication requires travelling the physical network at some finite speed. To simplify calculations, we assume that the communication time across a wire is equal to the total amount of information on the wire divided by the speed of the wire. Our goal for a given set of processes and a given network of processors is to minimize the total time it takes to process and communicate the information.

To develop the algorithm for such a mapping, we draw on our earlier work with the modified Kernighan/Lin algorithm and the functionally weighted graphs. Before indulging in the heart of the algorithm, first note that due to the design requirements of device drivers, certain processes must be bound to certain processors. These bound processes are equivalent to the “partitionally certain” processes of a functionally weighted graph, and set boundaries on parallel process distribution across the network.

Let $G$ be a weighted node and weighted edge graph of the processes we’d like to distribute. Let $P$ be a weighted node and weighted edge graph of the processors we’d like to distribute $G$ on. Let $M$ be an initial mapping that accounts for any bounded processes but otherwise provides no guarantees on initial allocation.

Let $\text{cost}(G, P, M)$ be the cost in time of mapping $G$ to $P$ with the pattern set in $M$. Thus $\text{cost}(G, P, M)$ is the processing time used by each physical processor ($\sum W_N(N_G \rightarrow N_P)$) plus the communications time used by each wire ($\sum W_E(E_G \rightarrow E_P)$). Calculating $\text{cost}(G, P, M)$ should take $O(|N|) + O(|E|)$ time, since each node and edge must be examined once.

Let $D(n, p)$ be the advantage derived from switching process $n$ from its current processor $o$ to processor $p$. Let $\text{Proc}(o, p)$ be the total processing time of processors $o$ and $p$ before $n$ switches and let $\text{Proc}_n(o, p)$ be the total processing time of processors $o$ and $p$ if $n$ were to switch. Let $\text{Com}_o(n)$ be the total communication time of the network if $n$ is on processor $o$ and $\text{Com}_p(n)$ be the total communication time of the network if $n$ switches to processor $p$. The value, then, of switching process $n$ from processor $o$ to processor $p$ is $D(n, p) = \text{Proc}(o, p) - \text{Proc}_n(o, p) + \text{Com}_o(n) - \text{Com}_p(n)$.

A new variant of the Kernighan/Lin Algorithm can then be used to allocate processes from a network to processors:

**The Mapping Kernighan/Lin Algorithm**

Compute $\text{cost}(G, P, M)$ for initial mapping.

$$O(|N|) + O(|E|)$$

Do { 

12
Compute $D(n, p)$ for all $p$ of all $n$ in graph. $O(|E||N_G|^2)$

Unmark all unbound nodes in $G$. $O(|N_G|)$

While (Unmarked nodes exist) {
    $|N|$ iterations
    Find one unmarked node $n$ that maximizes $D(n, p)$. $O(|N_G||N_P|)$
    Add $n$ and $D(n, p)$ to ordered list. $O(1)$
    Mark $n$. $O(1)$
    Update $D(n, p)$ for all unmarked nodes as if $n$ had switched. $O(|N_G||E|)$
}

Pick $j$ maximizing Gain, the sum of the first $j$ gains on the ordered list. $O(|N_G|)$

If Gain $> 0$ {
    Update mapping based on which nodes switch. $O(|N_G|)$
    Update $cost(G, P, M) = cost_{old}(G, P, M) - Gain$. $O(1)$
}

} While (Gain $> 0$)

One of the major advantages to the Mapping Algorithm, is that while it is slow, it is able to map parallel process networks onto irregular topologies, such as that shown in Figure 5. Figure 5 models a more complex processor distribution that Figure 1, based on the actual processor network used by Professor Scassellati.

Figure 5: A complex array of processors.
4 Future Work

There is much more work that can be done in examining both statically and
dynamically determined process distribution across parallel processor networks.
For example, while the equations studied for functionally weighted graphs as-
sumed a simple, linear correlation between independent variables and loads,
future studies should expand from these roots, studying what changes might be
needed to theory and to the model for more complex functions.

While some tests have been performed to verify the accuracy of these algo-
rithms, only the mapping algorithm has currently been implemented and tested.
Further work could be done to implement and test the other algorithms that
we have proposed, and to test them for performance. The major question, of
course, is does this additional computational work lead to real benefits?

Additionally, much work could be done to integrate the theory with the real-
life application used to “ground” the project in reality. We found it extremely
difficult to balance the twin demands of theory and application in this project,
especially when the application consisted of studying hundreds of lines of code
already created for another purpose. Since Porter allows the dynamic connec-
tion and disconnection of running processes; thus it should be possible for a
program to dynamically allocate processes across the network, modifying the
distribution of processes as new information about load is obtained. Hopefully
future students will be able to integrate this model into the a dynamic alloca-
tion program for the Porter system as used in Professor Scassleti’s lab. Doing
so, however, would require a knowledge of Porter, the robotic processes and the
QNX operating system that was beyond the scope of this project.

5 Conclusion

Ultimately we found this project to be very rewarding, both in our ability to
explore the algorithms of parallel programming and in dealing with a real-life
situation. There is still much work to be done in the field of parallel process
distribution, and these new algorithms, expanding the Kernighan/Lin model
beyond its traditional bounds through the examination of weighted nodes, func-
tionally weighted graphs, and mappings onto irregular network topologies, need
to be refined and more thoroughly tested.

Perhaps most enjoyable about this project, however, was its ability to span
the full range of our education within the Electrical Engineering/Computer Sci-
ence major. Having originally intended to do low level parallel programming,
the project evolved to encompass our work in algorithms, parallel program-
ing, digital systems, systems programming and even digital image processing.
Hopefully this study will encourage future students to take a similarly broad
approach to their own studies and research.
6 References


