Decentralized Optimal Power Pricing: The Development of a Parallel Program [*†]

S. Lumetta
L. Murphy X. Li
D. Culler I. Khalil

Department of Computer Science
571 Evans Hall
University of California at Berkeley
Berkeley, CA 94720

Abstract

For MPP's to solve new and interesting problems, they must support the development of sophisticated algorithms on very large data sets. Successful development depends strongly on the speed of the execute-fix cycle. Sequential machines cannot provide sufficiently fast execution of large problems, but many programming systems available on MPP's today neglect the significance of time spent fixing an algorithm during development. Those systems which do address the fix time commonly demand drastic sacrifices in execution speed. Between these two extremes is the middle ground where development must occur. We have implemented a new algorithm to solve an optimization problem for an electrical power system, a problem large enough to require significant computational resources. To help abstract the communication and layout requirements of the problem away from the main algorithm, we have developed a small object system library. The results are an efficient and easily modifiable solution to the problem and a general approach to solving this class of problems.

*This material is based upon work supported under a National Science Foundation Presidential Faculty Fellowship Award, a Graduate Research Fellowship, and Infrastructure Grant number CDA-8722788, as well as Lawrence Livermore National Laboratories Inst. for Scientific Research Grants #UCB-ERL-92/69 and #UCB-ERL-92/172. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of either organization.

†Versions of this work are to appear in the Proceedings of Supercomputing 1993 and the IEEE Parallel and Distributed Technology magazine.

1 Introduction

Use of MPP's divides into two classes: the brute force acceleration of simple numeric kernels and the development of novel solution techniques for problems large enough to require significant computational resources. The former demands only efficient execution, but the latter requires the ability to abstract away from the issues of layout and communication and the ability to quickly modify the algorithm under development. We have implemented a new algorithm to solve one of these latter problems—a large-scale optimization problem for an electrical power system.

The power system optimization problem is stated as follows: given a power network represented by a tree, with the power plant at the root and the customers at the leaves, use local information to determine the prices which will optimize the benefit to the community. In an operational system, the problem must be solved in a few seconds, and a simulation must demonstrate that these constraints are reasonable. The size of the network—10,000 customers served from a single plant or substation—is typical of realistic systems. To solve the problem, we implemented a novel strategy of iterative optimization put forth in [7]. The size of the problem, coupled with the real-time constraints, convinced us that solution on a workstation would be neither productive nor worthwhile.

Prior to our implementation, the algorithm had only been tested on toy systems three orders of magnitude smaller than our own. We expected difficulties to arise both in implementing the algorithm itself and in adjusting the algorithm to work on the larger system. As with any large problem, we wanted to divide the problem into smaller subproblems which could be solved independently. Specifically, we hoped to isolate the layout and communication problems from the main algorithm. We found, however, that the abstractions available to us were insufficient to allow easy segmentation. To meet the need for flexible but efficient code, we designed and implemented a system for fine-grained synchronization of data objects between processors. After encapsulating the layout and communication problems of the program within our object system, the remaining code became much cleaner and easier to modify.

Our implementation runs on the Connection Machine-5 at Berkeley. The CM-5 connects 64 Sparc processors with private local memories in a fat-tree network, providing a SPMD, message-passing paradigm with which to program (note that these 64 processors are not enhanced with vector capabilities as are some CM-5 installations). The CM Active Message layer[9] insulates the application from the private nature of the processor memory, allowing any processor to access memory on any other processor without the latter explicitly recognizing the former.

The remainder of this paper is organized as follows: Section 2 describes the problem and algorithm; Section 3 explains the goals of the object system and how they were met; Section 4 discusses the interactive process of development and the results obtained from the work. Section 5 summarizes the project, gives our conclusions, and suggests avenues for further research.

2 Pricing of Electric Power

Electric power distribution systems almost always have a tree structure, with a unique point of supply at the root (the *substation*) and the customers at the leaves. Intermediate nodes represent switches, tappoints, and transformers, where the path of electrical power is split. The tree structure for this application is shown in Figure 1. Ten main feeders run from the root, each branching off twenty lateral nodes. Each lateral node is the head of a line of five branch nodes, and each branch node has ten leaves. In total, there are $1,201$ internal nodes and $10,000$ customers. This system is a typical size for medium-to-large distribution systems.

The pricing problem is to set the price for each customer's power consumption so that the economic efficiency of the whole community is maximized. Customers are assumed to act locally to maximize their own benefits, based on the prices currently offered to them. The local information used by a customer to determine their maximum benefit is *private* to that

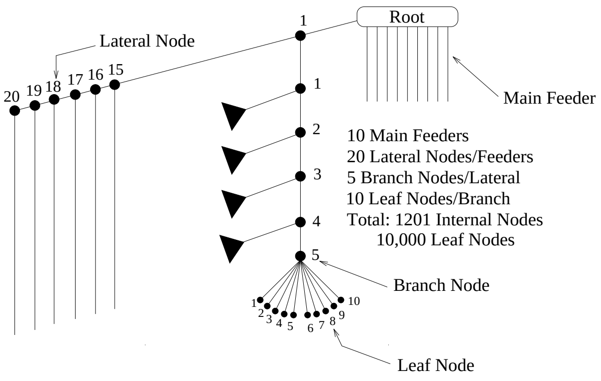

Figure 1: Power Distribution Tree

customer. On the other hand, customers are not concerned with the operation of the power system. They are only exposed to the effects of their actions on the system through the prices sent to them by the root. Thus, the pricing problem is inherently decentralized.

Previous pricing schemes either ignored the decentralized nature of the problem by assuming all the information necessary for solution was available in one location, or attempted to learn this information by using past behavior or by 'probing' the system. In [7], a pricing scheme was presented which sets the prices in such a way that individual benefit maximization by the customers also results in maximization of community benefits. This scheme addresses the issue of decentralized knowledge by using a new distributed pricing algorithm, emphasizing local computations.

The algorithm requires a number of parameters to be chosen beforehand. These include the local optimization conditions for the leaves (customer benefit functions and constraints), the power loss parameters for each line, the electrical constraints imposed by the power system, and the cost of supply of electrical power at the substation.

2.1 Problem Formulation

We formulate the pricing problem as an optimization problem at two levels, those of the customer and of the community. The mathematical details can be found in Appendix A.

	Partial list of variable names
P_i	Real power demanded by i^{th} customer
Q_i	Reactive power demanded by i^{th} customer
π_{iR}	Price for real power demand P_i
π_{iX}	Price for reactive power demand Q_i
P_r^{out}	Real power flow out of line r (sum of flows to

	children)
Q_r^{out}	Reactive power flow out of line r (sum of flows to children)
P_r^{loss}	Real power loss in line r
Q_r^{loss}	Reactive power loss in line r
P_r^{in}	Real power flow into line r (before loss)
Q_r^{in}	Reactive power flow into line r (before loss)
θ_R	Lagrange multiplier for the global real power equality constraint
θ_X	Lagrange multiplier for the global reactive power equality constraint

First, for each customer i, the cost of consumption is the price of electricity multiplied by the demand : $\pi_{iR} \cdot P_i$ for the real power consumed, and $\pi_{iX} \cdot Q_i$ for the reactive power. The problem is to find the real and reactive power demands P_i and Q_i which maximize individual benefit minus cost of consumption, for given prices π_{iR} and π_{iX}. There is a linear equality and a quadratic inequality constraint associated with each customer. Individual benefit maximization thus consists of solving 4 simultaneous nonlinear equations for each i.

The benefit of electricity consumption to the community is the sum of the individual benefits. The substation cost of supply is a negative benefit to the community. Hence the net benefit to the community is the sum of the individual benefits, minus the cost of supply.

This net benefit is maximized, subject to balance constraints on the flow of power in each lossy line. For a lossy line r, the balance constraints take the form:

$$\begin{aligned} P_r^{in} &= P_r^{out} + P_r^{loss} \\ Q_r^{in} &= Q_r^{out} + Q_r^{loss} \end{aligned}$$

The power losses in r are quadratic functions of the power flows into r; i.e., P_r^{loss} is a quadratic function of P_r^{in} and Q_r^{in}, and similarly for Q_r^{loss}. Thus, given the values of P_r^{out} and Q_r^{out}, these balance constraints are coupled quadratic equations in P_r^{in} and Q_r^{in}. In our application, there are $1,200$ pairs of power balance constraints.

Since the optimal values of the prices are not known in advance, we send prices down the tree from the root, compute the customer responses to these prices at the leaves, and propagate the effects of these responses on the system back up the tree to the root. The pricing algorithm is *iterative*, where each iteration consists of a downward sweep followed by an upward sweep.

Each P_r^{out} and Q_r^{out} is the sum of the flows to descendant nodes of r. The upward sweep begins by calculating demand at the leaves, where downstream flows are defined to be zero. Once the demands of all children of an internal node have been calculated, the demand of that node may be calculated to satisfy the balance constraints on the line above. This process leads to an upward sweep, propagating up the tree until it reaches the substation, where a convergence check is applied. If this check fails, the prices are updated, and propagated down the tree to the leaves (customers), which compute new demands. Then the upward sweep begins again.

The outline of the algorithm, broken into four steps, is shown below.

1. **Initialization**
   $k \leftarrow 0$
   Guess initial values of the multipliers associated with the substation convergence check, $\theta_R(0)$ and $\theta_X(0)$.

2. **Pass prices down and compute demand**
   (a) Compute the prices for customer i, using $\theta_R(k)$ and $\theta_X(k)$.
   (b) Compute customer i's demand, $P_i(k)$ and $Q_i(k)$, by solving the 4 simultaneous nonlinear equations associated with i.
   (c) Begin upward sweep at the leaves.
   (d) When $P_r^{out}(k)$ and $Q_r^{out}(k)$ have been calculated for r, compute $P_r^{in}(k)$ and $Q_r^{in}(k)$ by solving 2 coupled quadratic equations.
   (e) Propagate the computed power flows towards the root.

3. **Convergence test (done at the root node)**
   Compute the new values of total system demand.
   Use the new total demands to check for convergence of θ_R and θ_X.
   If converged, stop. Otherwise, go to step 4.

4. **Update the multipliers (done at the root node)**
   Use an update rule to generate $\theta_R(k+1)$ and $\theta_X(k+1)$.
   $k \leftarrow k+1$
   Go to step 2

The algorithm is described graphically in Figure 2.

3 Object System

Traditional message-passing systems present more trouble to programmers than do sequential or shared memory machines because of the need to design and implement message-passing protocols for communication and synchronization on a per-problem basis. The most efficient implementations require that this communication code be fully integrated with the rest of the program, but for development one must be capable of easily modifying the algorithm without redesigning

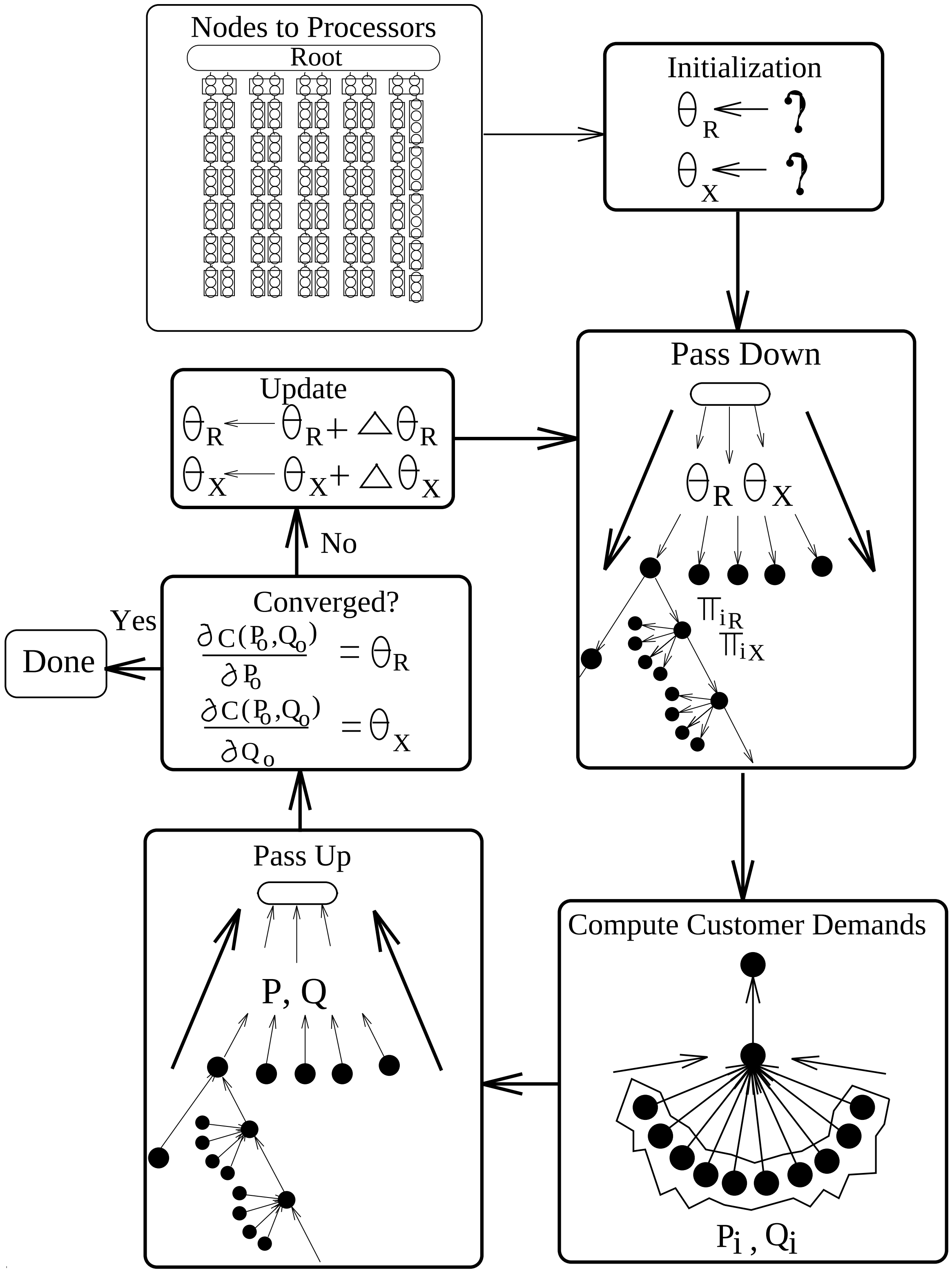

Figure 2: Algorithm to Iteratively Optimize an Electrical Power System

the communication pattern. One needs abstractions which are general enough to allow separation of the code sections and yet close enough to the hardware communication primitives that a reasonable efficiency is maintained. We began to work out a model for these abstractions, keeping in mind that while an object system needs to make correct and efficient programming easy, it need not make misuse impossible.

The central goal of the object system is to separate the problem of data layout from the algorithm itself by providing a global object space abstraction, yet to allow an optimized layout to be reflected in the execution time of the program. Data locality occupies a crucial position in efficient programming of MPP's, and the object system must provide good methods for the common case of local data so that the programmer is capable of optimization.

Design choices for such systems often depend on the particular objects being considered. The most obvious choice of object for the power problem is the node, each of which consists of about 100 bytes. This fine grain-size puts fairly strong constraints on the amount of tolerable overhead per object, but also implies that several accesses will be made to an object when it is used, making it more efficient to duplicate remote objects in local memory than to repeatedly reference them remotely. In addition, we would like to access objects directly on reference, without introducing extra levels of indirection.

3.1 Previous Systems

Several systems have already been developed for distributed parallel programming, including Ivy [4], Linda [3] and Tarmac [1] [5]. Shared virtual memory systems such as Ivy move entire memory pages between processors, clearly inappropriate for a problem in which an object averages 100 bytes. Linda is based on the tuple space abstraction, and requires that all shared data be encoded as tuples. The application has no say in placement of data or communication patterns, however, so the programmer can not optimize the program with an appropriate data layout.

The Tarmac system came closest to meeting the goals of our system. Tarmac provides a model of shared global state called mobile memory, which allows uniquely-identified and arbitrarily-sized objects to be created, moved, and copied. The original Tarmac abstraction [5] made no attempt to provide synchronization capabilities, but the CM-5 implementation [1] corrects this lack. However, the extra overhead involved in several of Tarmac's design decisions interferes with the goals of our system: under the mobile memory abstraction, Tarmac objects are not bound to any processor—each object can reside and be moved from processor to processor. When a processor wishes to access an object, it must follow a chain of 'hints' as to the object's current location (i.e., one or more levels of indirection). Keeping with this model, only a single copy of an object exists at any time—copying an object results in the creation of a new, uniquely-identified object. The object in Tarmac is thus inherently consistent, but at the cost of inefficiency in access and the possibility of objects thrashing between processors. Since our application does not require the mobile memory abstraction, we do not wish to pay the overhead to support these abstractions. We prefer a system which exploits the programmer's knowledge of access patterns by binding each object to a processor and which allows multiple copies of objects to exist when required for efficiency.

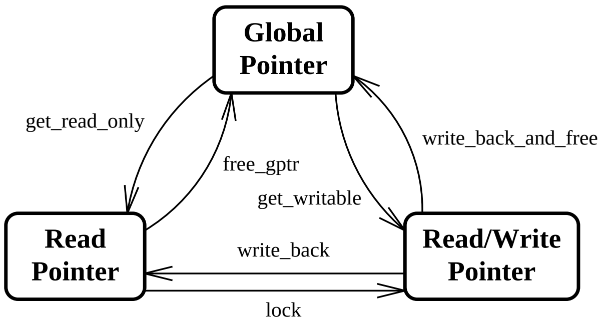

Figure 3: Gptr state model

After reviewing the object systems mentioned, we decided that none came sufficiently close to meeting the goals presented above for our program. The remainder of this section describes the object system we designed to meet our goals more effectively.

3.2 Pointer Model

The system is based on the *global pointer*, consisting of a home processor number and a pointer in the address space of the home processor.[1] The user is provided with routines to allocate data objects of arbitrary size and to access them asynchronously for reading and synchronously for writing.

When an object is created using **gmalloc**, a structure called a *gptr* is returned. A gptr structure can

[1]The reader familiar with Split-C [2] will recognize the similarity to the Split-C memory model—the main difference here is that our code is completely at the user level. Without compiler support, some concepts are difficult to implement as elegantly as one would like. We hope to port our system to Split-C in the near future.

be in one of three states, including the global pointer state, as shown in Figure 3. In this state, the gptr can be passed freely between processors, copied, stored in objects, and treated much the same as any other type of data. The data object itself, however, cannot be referenced while a gptr is in the global pointer state. The gptr can be transformed into either of the two other states via object system procedures.

The other two states of the gptr are used when a local copy of a data object is present on the processor. The first provides only the capability to read the data, and is called a *read pointer*. The second provides the additional capability to write to the data object, and is called a *read/write pointer*. While in either of these states, the gptr should only be used to reference the data; the gptr in these states has no significance to other processors, and may be corrupt even for the same processor at a later time. Thus, read pointers and read/write pointers can not be stored in objects or passed between processors.

Synchronized accesses can be accomplished by means of the read/write pointer. At most one gptr referring to a particular object can be in this state at any time, and the object itself is considered to be locked. Requests to change state from a global pointer to the object are denied while the object is locked, but no attempt is made to invalidate or update older *versions* of the object which may be present on other processors. Since managing data consistency at the library level must be general enough to provide consistency models for every program, and since this generality implies overhead not only for what is used but also for what is not, we felt that implementing a data consistency model was best left to the programmer. We encountered no problems in building an appropriate model for the power problem using the object system.

The object system library provides procedures to transform gptrs in any given state into any of the other states. Obtaining either a read pointer or a read/write pointer from a global pointer is done with the **get_read_only** and **get_writable** procedures. If the object is local to the processor requesting the state change, the gptr is simply changed (assuming the request was successful), and the data object itself is used as a virtual local copy. If the object is remote, the data is copied into local memory and the gptr is changed to reference this copy. Since most objects will be referenced more than once in a short period of time, it is more efficient to make a copy of a small object than to repeatedly request data from a remote processor.

A read pointer may be returned to the global pointer state by a call to **free_gptr**, or a read/write pointer for the object may be obtained by a call to **lock**. To avoid sending unnecessary messages, the latter does not obtain the latest version of the object. If the latest version is needed, one must first release the version being held with **free_gptr**.

Read/write pointers must write the modified data back to the data object to change to another state. **write_back** simply writes back the data and maintains a local copy of the object with a read pointer. **write_back_and_free** writes the data back and discards the local copy, returning the gptr to the global pointer state.

By direct modification of the gptr, the system avoids the expense of lookup for each reference. Macros are provided to determine the state of a given gptr, although in most cases the programmer will already know.

3.3 Implementation

A gptr consists of two 32-bit fields, one indicating the processor number of the processor on which the global data object is located, and the second field pointing to the object in the home processor's address space. The scheme used to differentiate between gptr states relies on the home processor (*pnum*) field. If the pnum is greater or equal to 0, the gptr is the global pointer state. Recall that no direct access is possible from this state. The constants READ and READ_WRITE are used in place of the processor for the read and read/write states. In both cases, the address field of the gptr becomes a local pointer to the data (or a copy of the data if the object was remote). Figure 4 shows the structures used for objects and local copies along with a global pointer and a read pointer.

In addition to the user-visible gptr structure, the library uses a second structure internally to manage synchronization and local caching of data objects. The global header, or *gheader*, structure is appended to the front of each data object as it is created, and also appears on the front of each local copy. The actual pointer value in both global and local gptrs points beyond this header.

The gheader structure contains four 32-bit fields. The first field holds the size of the data object, and is used to simplify requests for state changes which might require data transfers. Only a single bit is used in the second field, a flag for locking the object. The flag in the global object is set whenever a gptr is changed to the read/write state and cleared when the gptr is returned to the global pointer state. The lock bit in the local copy mirrors that of the

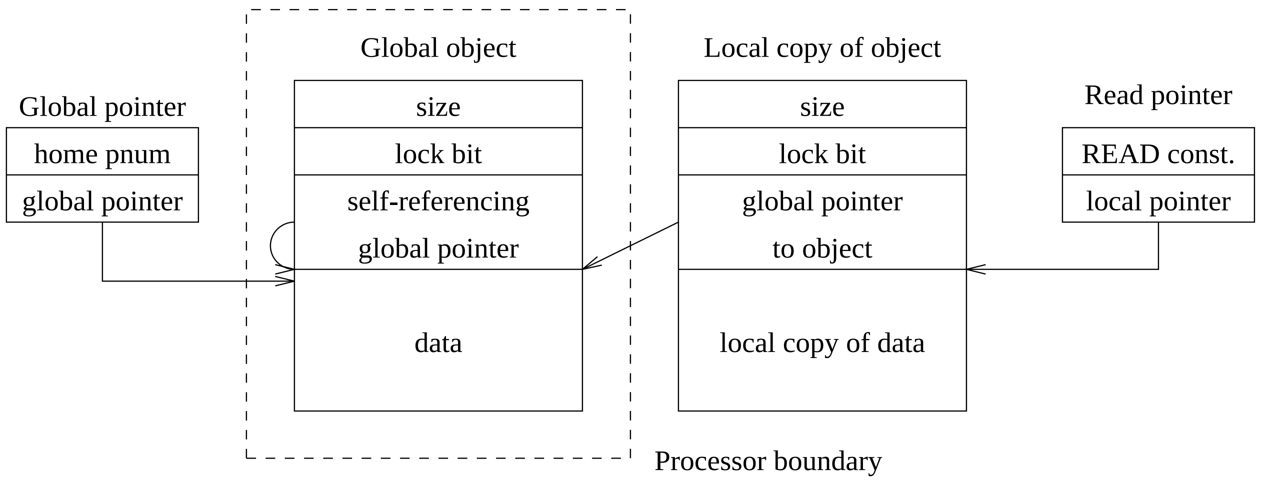

Figure 4: Global and local data objects

global object and is used to efficiently prevent more than one gptr on the same processor from entering the read/write state (see caching below). The third and fourth fields of the gheader form a global pointer which points to the global data object. When a routine changes the state of a gptr from global pointer to either read or read/write, it must store the global pointer for later use. The fields in the global object are self-referencing, and allow the system to change a global pointer into a local pointer for objects on the same processor by merely changing the pnum field to READ or READ_WRITE and to avoid relatively high cost of copying the data.

The other internal device used by the system is a hash table mapping global pointers to local copies. Because of the possibility of several gptrs on a single processor referencing the same data object, a mapping with reference counts is maintained, ensuring that only a single local copy of any object will ever exist. Any time a read or read/write pointer is requested, the routines first search the hash table to determine if a local copy already exists. If a copy is present, it is used, preventing costly communication with the data's home processor. Requests to obtain a read/write pointer must still contact the home processor for the lock, of course, but may not need to receive the data, just an approval. Again, if another gptr on the same processor already possesses the lock, refusal is automatic and requires no communication because the lock bit in the local copy will be set.

3.4 Example of Use

To demonstrate how the object system simplifies the code in the program, this section presents a portion of the code used to compute demand at the leaves. Figure 5 shows the code executed for each leaf owned by a processor. The leaves are linked in a list by gptrs, and the processor traverses the list and calculates the demand for each node, adding it to the downstream demand of the parent. If all of the children for a parent node have been processed, the parent node is added to the queue on its home processor (not shown).

Note that the only differences apparent between this code and code one would write for a uniprocessor are the calls to get and write back local copies of data, and the duplication of the parent's gptr. These calls can be likened to declaration of variables—the programmer declares which data he intends to read and which he intends to write, performs the actions, and then declares that he has finished with the data. Because it is modified directly, the local gptr is fully equivalent to a uniprocessor pointer.

4 Implementation and Results

After designing and implementing the object system to help break apart the electrical power network problem, we began to attack the problem itself. This section discusses the various stages of development, first with the structure of the program and the optimal solution for lossless power lines, and then with the development of the algorithm to solve the problem with typical loss rates.

4.1 Algorithm Implementation

The first part to be written was the code to distribute the network across the processors. The net-

```
        /* Obtain a writeable copy of the leaf node */
        /* and read the gptr to the next leaf node.   */

while (get_writable (&current));
next=LEAF_NODE (current)->next_leaf;

        /* Calculate the demand at the leaf */

optimize_node (&NODE_P (current), &NODE_Q (current),
                LEAF_NODE (current)->pi_R,
                LEAF_NODE (current)->pi_I);
if (NODE_P (current) < 0)
    NODE_P (current)=NODE_Q (current)=0;

        /* Duplicate the gptr to the parent of the */
        /* leaf, then modify the parent node to    */
        /* indicate that another child's           */
        /* calculations have been completed.       */

parent=LEAF_NODE (current)->parent;
while (get_writable (&parent));
NODE_P (parent)+=NODE_P (current);
NODE_Q (parent)+=NODE_Q (current);
done=++CHILDREN_DONE (parent);

        /* Write the modified data back */
        /* and free the local copies.   */

write_back_and_free (&parent);
write_back_and_free (&current);
```

Figure 5: Code segment for calculation of leaf demand

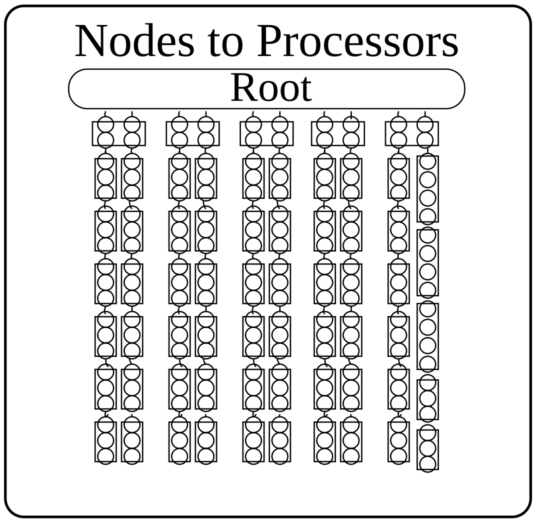

Figure 6: Distribution of the tree across processors

work used in the problem is fixed, so we avoided the issues of dynamic load-balancing and simply allocated the nodes to processors to roughly balance the load. The division of the tree is shown in Figure 6—the small boxes represent processors and each circle represents a lateral node and associated branch and leaf nodes. The root node is owned by one of the processors along the top row. After building the lateral nodes and passing a gptr to the next processor, each processor builds the nodes below each lateral node it owns. Node initialization is performed as each node is created.

Once the tree structure had been set up, other sections of code could treat the tree as if it were completely local, with the object system handling any implicit communication. For example, see Figure 5 for a sample segment of code for computation of customer demand.

The iteration of the algorithm became a simple loop relying on two procedures to perform the work. The first procedure corresponds to the 'Pass Down' frame of Figure 2 and computes the path-dependent price information for each customer. The second corresponds to the 'Compute Customer Demands' and 'Pass Up' frames, which find customer demand and calculate line losses, passing information upward to determine the power demand at the root. The 'Converged?' and 'Update' frames were coded directly into the main loop since only the owner of the root node need perform these actions.

4.2 Lossless Solution

Working with the simpler problem of a lossless system, we debugged the code and found that the basic algorithm did work as expected. Although we can not directly verify that our solution to the lossless problem is indeed correct, predictions based on smaller systems agree with our results, and scaling is reliable with lossless systems.

Recall now the real-time constraints on the problem: the substation will perform this algorithm with a period of between 10 and 30 minutes. Prices must not be allowed to vary for more than a small fraction of this period to ensure reliable costs to the customers. With these limits in mind, we proceeded to time our first results.

Timing with the 33MHz processor clock on one of the processors (after processor synchronization), the solution of the lossless network takes a total of 5.436 seconds, most of which is spent building the tree. The iterations take between 107 and 691 milliseconds, depending on how close the leaf nodes start to the solution of the demand optimization. The previous solution of demand from a leaf node is used as initial values in the subsequent solution, and computation time decreases monotonically as the algorithm progresses. The iterations are computationally intensive, with only about 10 milliseconds needed to pass prices down the tree and, presumably, a similar amount to pass information back up. Clearly the parallel machine meets the constraints of the problem, demonstrating that the algorithm can meet the needs of actual power networks.

Initial timing results on a workstation indicated that solution of a single customer demand problem required about 6 milliseconds. Scaling the problem to 10,000 nodes, we expect that the problem will take about one minute per iteration. Even the lossless problem requires six iterations, so a workstation could well require one-half of the pricing period just to settle the prices. To verify our beliefs, we built the tree on a single processor and solved the lossless problem. Finding the solution required an average of 48 times longer than did 64 processors, indicating approximately 75% efficiency. Part of this speedup can be attributed to the cache size—1/64th of the tree fits into the cache, while the entire tree does not. Running the code on the workstation mentioned above, we found that the solution in fact took twice as long as predicted—a total of 12 minutes for the lossless problem. Neither the single CM-5 Sparc processor nor the workstation is capable of meeting the real-time constraints of the problem. An adequately fast solution requires the computational power of a supercomputer.

4.3 Code Development

After our initial success with the lossless problem, we set the line impedances to what we believed to be typical values (ten times the values given in Appendix B). The algorithm broke down trying to solve such a high-loss problem—demand oscillated wildly between almost nothing and about twice that of the lossless solution. Reducing the line impedances, we found that the algorithm converged for impedances of up to 1/10th of the proposed values. Table 1 shows the solutions for various impedances, scaled to actual typical impedances. Noting that the total demand at the root decreases almost linearly in impedance over the range measured, with real demand dropping by 2.4 units at $1/100^{th}$, 23.4 units at $1/10^{th}$, and 216.7 units at full impedance, we were able to guess at the solution to the high-loss problem.

Over the next few weeks we directly verified our ideas about reducing modification time as we passed hundreds of times through the execution-fix cycle. The time invested in development of the object system was more than returned in the time saved in later modification. The global memory abstraction insulated the necessary changes from the communication schemes, allowing us to concentrate on the algorithm itself.

After many failed attempts to eliminate the oscillations, we tested the system to determine the amount of power being lost in the lines. The line impedances proposed initially for the system proved far too high—over 16% of the power was lost near equilibrium. Since no actual system would allow the total losses to exceed 2-3% of demand at the root, we probed the system with successively higher line impedances to determine the percentage of power loss for which the algorithm broke down. The results are shown in Table 2. The algorithm breaks down at about four times the typical line impedance, when the power lost in the system is roughly three times the nominal value of 2-3%.

5 Conclusions and Future Work

Using the CM-5 as a development tool, we have successfully implemented a new algorithm to solve the problem of optimal pricing for electrical power networks. Because of the problem's size, solution by a workstation would be unproductive. Because of the real-time constraints imposed on the problem for use with actual power networks, such a solution would also

Fraction of Typical Impedance	Number of Iterations	θ_R	θ_X	P_0	Q_0
0	6	0.72527	0.14505	7252.7	1450.5
$1/100^{th}$	6	0.72504	0.14502	7250.3	1450.2
$1/10^{th}$	12	0.72293	0.14472	7229.3	1447.2
Typical	19	0.70360	0.14192	7036.0	1419.2

Table 1: Results for Various Loss Rates

Multiple of Typical Impedance	Number of Iterations	Demand at Root	Demand at Leaves	% Loss
Typical	18	7034.7	6862.9	2.44
2	21	6848.9	6535.1	4.58
3	48	6686.8	6252.5	6.49
3.5	107	6612.5	6124.4	7.38

Table 2: Power Lost for Various Loss Rates

fail to be worthwhile. An adequate solution to the problem requires the computational power of a parallel machine.

We approached the problem by first separating out the requirements of communication and layout from the algorithm. Through the abstraction provided by our object system, we removed the complexity of integrating communication patterns into the main code. As a result, the code became more flexible and easier to modify. The overhead cost of the simplification was not unreasonable, with processors operating at approximately 75% efficiency. We feel that our approach could be profitably extended to many problems.

Although we have made progress in understanding optimal power pricing in distribution networks, many avenues remain for exploration. Realistic power systems might reconfigure the network, for example, so a simulation must be capable of redistributing nodes to processors in a short time. The addition of direct power flow constraints on internal lines would also increase realism, perhaps requiring iteration over each subtree to arrive at a valid solution. Also, some of the equations used in our solution were simplified versions of the actual optimization equations—the precise equations may introduce further complications in the algorithm. Furthermore, customers in realistic systems would have distinct benefit and inequality functions, whereas each of the customers in our example used the same functions.

We also hope to further develop the object system by integrating it into the Split-C library[2]. Split-C is an extension to the C language which provides efficient support for the shared memory, message passing, and data parallel programming paradigms. Making the system available for use by others at Berkeley will provide valuable feedback about its features, drawbacks, and general usefulness.

Acknowledgements

The authors would like to thank Professor Felix Wu for helpful discussions about the material presented here.

A Mathematical Formulation

	<u>**List of Variable Names**</u>
n	Number of customers (leaf nodes)
P_i	Real power demanded by i^{th} customer
Q_i	Reactive power demanded by i^{th} customer
$b_i(P_i, Q_i)$	i^{th} customer benefit as a function of P_i and Q_i
$g_i(P_i, Q_i)$	Maximum demand (inequality) constraint on customer i
$h_i(P_i, Q_i)$	Plant (equality) constraint on customer i
π_{iR}	Price for real power demand P_i
π_{iX}	Price for reactive power demand Q_i
R_r	Resistance in lossy connection line r
X_r	Inductance in lossy connection line r
P_r^{out}	Real power flow out of line r (sum of flows to children)
Q_r^{out}	Reactive power flow out of line r (sum of flows to children)
P_r^{loss}	Real power loss in line r
Q_r^{loss}	Reactive power loss in line r
P_r^{in}	Real power flow into line r (before loss)
Q_r^{in}	Reactive power flow into line r (before loss)

Iteration	θ_R	θ_X	P_0	Q_0
Seed	0.700000000	0.140000000	8239.801156694	1664.608805105
1	0.700000000	0.140000000	6959.868551160	1403.411689330
2	0.699019554	0.140084472	7188.664606141	1450.045276479
3	0.703868325	0.141302650	7004.110193816	1412.489213193
4	0.703001600	0.141289347	7056.467032462	1423.147533006
5	0.703664711	0.141543232	7027.919801036	1417.338490767
6	0.703445923	0.141590428	7038.171058708	1419.424940720
7	0.703538976	0.141677597	7033.458500768	1418.465918893
8	0.703490560	0.141719440	7035.284828317	1418.837616770
9	0.703500067	0.141760125	7034.529297524	1418.683852235
10	0.703488250	0.141786929	7034.794122864	1418.737754710
11	0.703486035	0.141808432	7034.724649824	1418.723604511
12	0.703482633	0.141824260	7034.725501200	1418.723781255
13	0.703480105	0.141836174	7034.727268342	1418.724142099
14	0.703478255	0.141845147	7034.728379528	1418.724368484
15	0.703476897	0.141851904	7034.728625597	1418.724418497
16	0.703475886	0.141856989	7034.728481712	1418.724389115
17	0.703475124	0.141860814	7034.728188545	1418.724329308
18	0.703474546	0.141863691	7034.727974218	1418.724285567

Table 3: Results for Typical Loss Rates

P_0	Total real power supplied by substation
Q_0	Total reactive power supplied by substation
$c(P_0, Q_0)$	Cost to substation for supplying P_0 and Q_0
θ_R	Lagrange multiplier for the global real power equality constraint
θ_X	Lagrange multiplier for the global reactive power equality constraint

In practice there would be some variation in customer behavior. In this application, however, we assume that the customers are identical. Therefore the same model applies to all n customers. The i^{th} customer's problem is

$$\max_{P_i, Q_i} \{b_i(P_i, Q_i) - (\pi_{iR} \cdot P_i + \pi_{iX} \cdot Q_i)\} \qquad (1)$$

$$\begin{array}{lrcll} \text{such that} & b_i(P_i,Q_i) & = & \log(1+P_i)+\log(1+Q_i) & \\ & h_i(P_i,Q_i) & = & P_i - 5Q_i & = \quad 0 \\ & g_i(P_i,Q_i) & = & P_i^2 + Q_i^2 - 0.8 & \leq \quad 0 \end{array}$$

Assume that all lines in Figure 1 except those from branch nodes to leaves have impedance of the form $Z_r = R_r + jX_r$ (see Figure 7). The power flows in line r are then related by

$$\begin{aligned} P_r^{in} &= P_r^{out} + P_r^{loss} \\ Q_r^{in} &= Q_r^{out} + Q_r^{loss} \end{aligned}$$

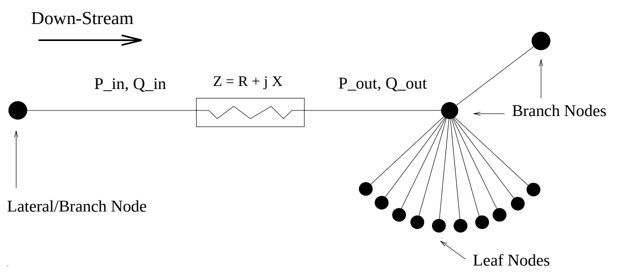

Figure 7: Typical lossy line

The power losses in line r depend directly on the flows into the line:

$$P_r^{loss} = R_r \cdot ((P_r^{in})^2 + (Q_r^{in})^2) \qquad (2)$$

$$Q_r^{loss} = X_r \cdot ((P_r^{in})^2 + (Q_r^{in})^2)$$

Given P_r^{out} and Q_r^{out}, the above coupled quadratic equations are solved to find the power flow into each line. This process propagates up the tree until the substation demand P_0 and Q_0 (at the root) is found.

The global problem is then

$$\max_{P_0, Q_0, \{P_i\}, \{Q_i\}} \left\{ \left[\sum_i b_i(P_i, Q_i) \right] - c(P_0, Q_0) \right\} \qquad (3)$$

such that

$$\begin{array}{rcl} P_0 - \sum_i P_i - \sum_r P_r^{loss} & = & 0 \\ Q_0 - \sum_i Q_i - \sum_r Q_r^{loss} & = & 0 \\ \forall i, \quad h_i(P_i, Q_i) \; = \; P_i - 5Q_i & = & 0 \\ \forall i, \quad g_i(P_i, Q_i) \; = \; P_i^2 + Q_i^2 - 0.8 & \leq & 0 \end{array}$$

where, in this application,

$$c(P_0, Q_0) \; = \; \frac{1}{2 \times 10^4} \, (P_0^2 + Q_0^2)$$

Note that the constraints h_i and g_i are satisfied locally by leaf i, based on the i^{th} prices.

Assume (3) has a solution $\hat{P}_0$, $\hat{Q}_0$, $\{\hat{P}_i\}$, $\{\hat{Q}_i\}$, $\hat{\theta}_R$ and $\hat{\theta}_X$. Then from the Kuhn-Tucker optimality conditions [6], we get

$$\pi_{iR} = \hat{\theta}_R \left(1 + \sum_{r \in U(i)} \alpha_r \right) + \hat{\theta}_X \left(\sum_{r \in U(i)} \frac{X_r}{R_r} \alpha_r \right)$$

$$\pi_{iX} = \hat{\theta}_X \left(1 + \sum_{r \in U(i)} \beta_r \right) + \hat{\theta}_R \left(\sum_{r \in U(i)} \frac{R_r}{X_r} \beta_r \right)$$

$$\text{where} \qquad \begin{array}{rl} \alpha_r & = 2R_r \dfrac{P_r^{in}}{1 - 2R_r P_r^{in} - 2X_r Q_r^{in}} \\ \beta_r & = 2X_r \dfrac{Q_r^{in}}{1 - 2R_r P_r^{in} - 2X_r Q_r^{in}} \end{array}$$

and $U(i)$ is the set of lines on the path from the root to i.

Since the optimal values of the multipliers, $\hat{\theta}_R$ and $\hat{\theta}_X$, are not known in advance, we iteratively send prices down the tree and propagate demand back up, each time adjusting the prices towards their optimal values. We begin by guessing values for θ_R and θ_X, and at each iteration we check the optimality conditions

$$\begin{array}{rcl} \dfrac{\partial c(P_0, Q_0)}{\partial P_0} & = & \theta_R \\ \dfrac{\partial c(P_0, Q_0)}{\partial Q_0} & = & \theta_X \end{array}$$

Hence after receiving the total demand P_0 and Q_0 at the root, we check if the condition holds to the desired precision. If it does not, we update θ_R and θ_X using the following rule:

$$\theta_R \; \leftarrow \; \theta_R + \delta\theta_R \;\; , \;\; \theta_X \; \leftarrow \; \theta_X + \delta\theta_X$$

where the corrections are defined by

$$\begin{array}{rcl} \delta\theta_R & = & -\dfrac{\theta_R - \frac{\partial c}{\partial P_0}}{1 - \frac{\partial^2 c}{\partial P_0^2} \frac{\partial P_0}{\partial \theta_R}} \\ \delta\theta_X & = & -\dfrac{\theta_X - \frac{\partial c}{\partial Q_0}}{1 - \frac{\partial^2 c}{\partial Q_0^2} \frac{\partial Q_0}{\partial \theta_X}} \end{array}$$

B Convergence for Typical Loss Rates

Table 3 shows the convergence to optimality for a system with typical losses (around 2.5% of total power is lost in the lines): $R_r = 1/(3 \times 10^5)$ and $X_r = 1/10^6$ for lines leading to lateral nodes and $R_r = 1/10^4$ and $X_r = 1/(5 \times 10^4)$ for lines leading to branch nodes.

Recall that the convergence criteria here are given by:

$$\begin{array}{rcccl} \dfrac{\partial c(P_0, Q_0)}{\partial P_0} & = & \frac{P_0}{10000} & = & \theta_R \\ \dfrac{\partial c(P_0, Q_0)}{\partial Q_0} & = & \frac{Q_0}{10000} & = & \theta_X \end{array}$$

Notice that the convergence towards the final solution in iteration 18 is initially underdamped, with prices oscillating around the solution, but in later iterations becomes overdamped, approaching the solution from above. We attribute this phenomena to the interaction between the root price multipliers θ_R and θ_X and the intermediate node multipliers α and β—the same interaction which causes the algorithm to break down at high loss rates.

References

[1] D. F. Bacon and S. E. Lucco, "Tarmac: A Mobile Memory System for the Connection Machine CM-5," Draft

[2] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, K. Yelick, "Introduction to Split-C," to be published in *Proceedings of Supercomputing*, 1993.

[3] D. Gelernter, "Parallel Programming in Linda," *Proceedings of the International Conference on Parallel Processing*, pp. 255-263, Aug. 1985.

[4] K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," *Proceedings of the 5th Annual ACM Symp. on ACM Conf. on Principles on Distributed Computing*, pp. 229-239, 1986.

[5] S. E. Lucco and D. P. Anderson, "Tarmac: a Language System Substrate Based on Mobile Memory," UCB Report CSD 89/#525, November 1989.

[6] D. G. Luenberger, *Linear and Nonlinear Programming*, 2nd. Ed., Addison-Wesley, 1989.

[7] L. Murphy, R. J. Kaye and F. F. Wu, "Distributed Spot Pricing in Radial Distribution Systems," Paper 93 WM 148-7 PWRS, presented at the IEEE Power Engineering Society 1993 Winter Meeting, Columbus, OH, Jan 31 - Feb 5, 1993.

[8] P. Stenström, "A Survey of Cache Coherence Scheme for Multiprocessors," IEEE Computer, pp. 12-24, June 1990.

[9] T. von Eicken, D. E. Culler, S. C. Goldstein, K. E. Schauser, "Active Messages: a Mechanism for Integrated Communication and Computation," UCB Report CSD 92/#675, March 1992.