UC-405

SAND92-8467J
Unlimited Release
Printed March 1992

A Parallel Network Computer Approach for 3D Geophysical Modeling

J.C. Meza
Center for Computational Engineering
Sandia National Laboratories
Livermore, California 94551-0969

M. Koshy and V. Pereyra
Weidlinger Associates
Los Altos, California

ABSTRACT

Modeling elastic wave propagation in complex three-dimensional geologic structures is an important problem in exploration geophysics. Even today these problems can task most supercomputers to their limit. Fortunately, the approximate solution of these problems by seismic ray tracing can be easily parallelized. We propose here an approach based on using a local area network of workstations for parallelizing a seismic ray tracing code. We use two public-domain libraries, PICL and TCGMSG, to implement a simple message passing system for communicating between different processors over an Ethernet network. We also test this approach on a set of realistic problems showing an almost linear speedup for all sufficiently large cases.

1. Introduction

Seismic methods are one of the main tools used in modern oil exploration, reservoir delineation, and mapping. A seismic survey consists of setting up sources of elastic energy (explosives, vibrators, air guns) and recording the back-scattered signals at multiple points on the earth's surface or in wells, with the purpose of obtaining information about the hidden structure of the earth.

For many years, due to computational constraints, a two-dimensional approach was used, which assumed that the earth was laterally homogeneous in one direction. In more recent years, with increasing and more readily available computational power, truly three-dimensional surveys have become more common. These three-dimensional surveys not only provide increased resolution and reliability, especially in areas of complex geology, but they have also proven to be cost efficient.

A three-dimensional survey usually involves hundreds of shots, and digital recordings of a few seconds at hundreds of locations on a two-dimensional grid. The elastic energy from each shot propagates into the earth, refracts, diffracts, and reflects at interfaces between sharply different rock types, and finally, some of the energy returns to the earth's surface (see figure 1). The paths that the energy follows between a source and a receiver is called a ray. Rays, in isotropic media, are orthogonal trajectories to wave fronts, and they are defined by Fermat's principle of minimal (stationary) time.

A complete mathematical simulation of a seismic survey requires the following two steps: 1) a parameterized earth model, and 2) the solution of the elastic wave equation for the earth model. In the first step, we need to specify the material properties that affect wave propagation for each point in the volume of interest. In the isotropic case, three quantities suffice: velocities of propagation of the pressure and shear waves and densities. We also need to specify boundary conditions and model the shot, that is the initial perturbation.

Since the purpose of the seismic experiment is to determine the inner structure of the earth, the first step requires an initial guess, where all the available information about the site should be incorporated. The second step then simulates the shot and its propagation through the earth volume. Finally, a synthetic, mathematically calculated survey is produced and compared with the measured data. From the differences between these two sets one must then infer a change in the model parameters. By iterating this process within an optimization algorithm one expects to extract the information about the earth structure that the data contains.

We remark that step 2 has to be done independently for each shot. The approximate solution of the elastic wave equation (two coupled partial differential equations of hyperbolic type) in three dimensional general media can be accomplished by numerical methods such

as finite differences or finite elements. This type of approach is highly computer intensive. For each shot it will provide the detailed time evolution of the displacement field over the whole three-dimensional volume. Of course, since we observe only at a finite number of locations on the free surface or in wells, there is considerable waste in this calculation. Since the imaging or inverse modeling process in step 2 has to be iterated a number of times, this task has the potential of taxing the largest supercomputers. Thus, we need to improve the efficiency of the numerical process and also choose an approach that can be parallelized.

Instead of solving the full elastic wave equation, we can calculate the ray paths that join sources with receivers, after interacting with the media, say by reflecting at an interface. Having calculated the trajectory we can also obtain the time of travel. By processing and displaying the data in appropriate ways, we can identify coherent signals corresponding to such reflections, and therefore pick corresponding measured travel times accurately. Thus, we can do inverse modeling as described above by matching calculated and measured travel times of picked, interpreted arrivals, using the more inexpensive ray tracing instead of the full elastic wave calculation.

Although less expensive, ray tracing can still be quite time consuming for realistic problems. For instance, a typical data set corresponding to an inverted vertical seismic profiling experiment, consisting of a number of source positions on a well, and data recorded on a rectangular array of geophones on the free surface contains 55,000 picked direct arrivals. Ray tracing in a preliminary model with 12 homogeneous horizontal layers produces about two arrivals per second on a workstation, and thus a complete simulation of this data set would take approximately nine hours. If we want to apply travel time inversion to such a data set and model, then each iteration of a nonlinear optimization algorithm will require the time indicated above, and approximately 30 iterations would be necessary to obtain a good fit.

Our ray tracing module is organized in such a way that independent tasks are apparent, and only a moderate amount of development is required to create a parallel version. The subtasks correspond to calculating primary reflections from a collection of r different reflectors, corresponding to a set of s shots. These two loops break out naturally into independent tasks, where each shot and each reflector is treated separately. Each one of these $(r \times s)$ independent tasks is given to a different processor. We should note that not all of the subtasks are of equal difficulty; in fact deeper reflectors will require more computing time than shallower ones because they have longer paths. In addition, in a distributed network of computers we cannot assume that each one of the processors has the same computational power. Both of these factors have implications on load balancing that must be addressed in any distributed computer strategy.

In the next section we describe the seismic ray tracing equations in more detail. Section 3 gives a short overview of network computing and a description of our parallel version of a

ray tracing code. In section 4, we give some numerical results using the parallel code on a local area network of workstations. We then give some concluding remarks in section 5.

2. Seismic Ray Tracing

The equations of seismic ray tracing can be derived either from the eikonal equation or directly from Fermat's principle of least time. One form of these ordinary differential equations in three dimensions is:

$$\dot{\eta}(s) = v(\eta)w, \tag{2.1}$$
$$\dot{w}(s) = \nabla u(\eta), \tag{2.2}$$

where $\eta = (x(s), y(s), z(s))$, $w(s) = u(\eta)\eta$, v is the velocity of propagation, $u = 1/v$ is the slowness, and s is the arc length.

The simplest form of ray tracing is shooting, in which the initial position, and the initial direction of the ray are prescribed by:

$$\eta(0) = \eta_0, \tag{2.3}$$
$$w(0) = w_0. \tag{2.4}$$

We use shooting only as a means of initializing a source-receiver iterative calculation or to check if a two-point ray has changed signature (a ray signature is an ordered sequence of reflecting interfaces). For smooth velocity fields, equations (2.1-2.4) constitute an initial value problem that can be solved numerically by any standard technique. The case of complex geological models where the velocity field is usually discontinuous is more difficult, since then precise intersections with the discontinuity surfaces have to be calculated.

The software package RAY3D [1] computes rays that originate from a number of sources, travel through an inhomogeneous medium and arrive at an array of receivers. The three-dimensional regions we consider have material properties which are inhomogeneous with material discontinuities. The boundary surface between different smoothly varying regions are called reflecting interfaces or simply reflectors. RAY3D computes the rays which reflect from each of these interfaces and reports them to the user in separate data files. Furthermore, the rays generated by each shot are computed sequentially for each reflecting horizon.

The program uses a general two-point boundary value approach for tracing rays from a source to a receiver array. A multi-point boundary value finite difference solver for nonlinear systems of first order ordinary differential equations is used. This solver has variable order, variable mesh, and global error estimation capabilities. The ray equations (2.1-2.2) are

discretized on a mesh by a second order trapezoidal rule. An efficient Newton-type nonlinear equation solver with a sparse linear solver is used on the resulting nonlinear difference system. As mentioned above the shooting method is used to provide an initial guess for the nonlinear solver.

To generate a synthetic survey then, one must solve a boundary value problem for each combination of source and receiver. For a given shot, however, the computed ray for a given receiver can be used to start the calculation for a nearby receiver (receiver continuation), so we want to keep all these two-point solves together. Each computation involving a specific shot and reflector is independent of all the others. In addition, if the receiver array is large and the model is complex each job is computationally intensive requiring several minutes of CPU time on a typical workstation.

3. Network Computing

There are many approaches to parallelization. Here, we only touch on ideas suitable for distributed memory multiprocessors. Two examples of this type of architecture are the hypercube machines (NCUBE, Intel) and networks of computers. The idea of using a local area network as a computing resource is hardly new. One of the earliest attempts at using a local area network in this manner was the work by Whiteside and Leichter [2]. They showed that they could use a programming model based on LINDA [3] over a local area network to achieve supercomputer performance on various problems. Recently another approach based on PVM [4] (Parallel Virtual Machine) from Oakridge National Laboratories was proposed. Yet another model, TCGMSG [5] also based on using a heterogeneous network of computers, has been used for quantum chemistry applications.

One of the frustrating problems in developing parallel codes is the rather large number of programming models one can choose from. Yet it is clear that at least at the lowest level all of these models have certain characteristics in common. In order to save development time, we have chosen to pick one set of basic primitives and to write all of our code in these primitives. The interface to the various parallel machines is then accomplished by linking with appropriate libraries for specific machines. This approach is closely modeled after the work on PICL (Portable Instrumented Communication Library) described in [6]. PICL provides a small set of low-level communication primitives as well as a set of higher level functions for global operations and synchronization. The low-level routines consist of 12 functions: *check0, clock0, close0, load0, message0, open0, probe0, recv0, recvinfo0, send0, sync0*, and *who0*. In many cases these set of primitives are enough to write a parallel program.

Having chosen PICL as our basic set of primitives, we still needed a library to imple-

ment message passing on a heterogeneous network. While LINDA is very powerful, it does not currently work well over heterogeneous networks. Another difficulty is that current implementations of LINDA do not take advantage of multiprocessors on the network. PVM appears to work in a heterogeneous environment but at the time this project was started it was still in a testing phase. Additionally, PVM makes certain assumptions on the location of the user programs that are too restrictive for our applications.

Of the approaches mentioned above we chose the TCGMSG toolkit because of its ease of use. The system worked well over a wide range of computers on our network and it interfaced well with PICL. One feature which proved very useful was the idea of the PROCGRP file which is used to configure the computers that the user wishes to use in the network. The only disadvantage we found is that the *send's* are blocking. In our particular application this did not prove to be overly restrictive.

3.1 Parallel RAY3D implementation

Using the low level routines from PICL and the message passing capabilities of TCGMSG it is easy to implement a parallel version of RAY3D, extending to a heterogeneous network a previously implemented LINDA version [7]. Our parallel approach is based on a master-slave relationship. The master or host node is distinguished from the other nodes in that it handles all the communication necessary between the nodes. The main job of the master node is to allocate the jobs to the nodes and distribute new jobs as nodes finish their tasks. When all jobs are allocated the master node waits for a message from each of the nodes that is still active and terminates when all jobs are finished. A high level overview of the main routine is given in Figure 3.

The worker routine is merely an interface between the master and the serial version of the RAY3D program. When the worker routine first starts it waits for a message to start a job. Upon the receipt of a start message the worker calls the RAY3D program to ray trace for one source, one reflector, and all the receivers. If the program terminates normally, the worker routine then sends a message back to the master node telling it that it is done and waits for another start message. At some point all of the jobs will be either done or allocated to a node. When this happens the master node sends the worker node a quit message. Any other message is considered an unknown message and terminates the worker routine with an error. Figure 4 gives some of the details of this routine.

The subroutines that handle the allocation of jobs and checking that all jobs are done are straightforward except for one detail that will become important in a later section. It is clear that the jobs can be allocated to the nodes in any manner. At the beginning of a simulation the total number of jobs, as well as all information required to start them, is known. In the

numerical example described in the next section, it is also known that jobs corresponding to deeper shots take longer to compute than shallower ones. Using a bin packing argument it is not hard to see that allocating the longer jobs first is therefore a better allocation strategy. In fact, this initial distribution of jobs does turn out to be better than a previous strategy in which the jobs corresponding to the shallower shots were allocated first. The allocation scheme whereby a node is given a new job immediately after finishing with its current job also helps to automatically load balance the system, as the faster nodes tend to receive a greater number of jobs.

4. Numerical Results

We tested our parallel version of RAY3D on a network of computers comprised of a combination of SGI workstations, SGI Power Series multiprocessors, and SUN workstations at Sandia National Laboratories. A schematic of the local area network is given in Figure 5. A more complete description of the workstations on this local area network is given in Table 1.

Table 1: Computers on the network.

Name	Type	CPU Speed (MHz)	Memory (MBytes)	# CPUS
ransgi	SGI 4D/380S	33	128	8
trantor	SGI 4D/240S	25	64	4
prufrock	SGI 4D35	36	16	1
ole	SGI 4D35	36	16	1
mom	SGI 4D25	20	8	1
baley	SGI 4D25	20	16	1
smyrno	SGI 4D20	12	16	1
matterhorn	SGI 4D20	12	16	1
solaria	SUN Sparc 1	20	40	1
hobbs	SUN Sparc 1	20	40	1
slice	SUN Sparc SLC	20	8	1
splash	SUN Sparc 1	14	16	1

Because of the wide range in performance within this network, it is difficult to measure speedup in a straightforward way. Instead, to determine an appropriate measure of performance we benchmarked each one of the computers in the network on a representative problem and used this as a measure of its computing capability. As the base problem we chose a representative number of shots spread evenly along a wellbore for an inverse seismic

profiling case described below. This problem took roughly 42 minutes to run on 1 processor of the SGI 4D/380S. The timings for the rest of the workstations are given in Table 2. All of the timings are for single precision computations using the UNIX *gettimeofday()* function. The ratio in the last column of Table 2 is an indication of the relative power of a particular workstation to 1 cpu of the 4D/380S machine.

Table 2: Relative computing power of workstations on the network.

Name	Time	Equivalent CPU
ransgi(1)	2515.98	1.00
trantor(1)	3268.61	0.77
prufrock	2326.39	1.08
ole	2326.39	1.08
baley	4545.49	0.55
matterhorn	8056.96	0.31
solaria	7240.42	0.35
splash	19915.48	0.13
slice	7394.15	0.35

To demonstrate the effects of problem size we picked a set of test problems based on a model with 2 reflecting surfaces and 3 regions with inhomogeneous velocities described by tri-cubic tensor products of B-splines. The receiver array on the free surface has 450 geophones in a rectangular grid. The number of shots and therefore the total number of tasks varied for each test problem from the smallest case with 5 shots to the largest problem with 80 shots. Each of the test problems is designated by the number of shots calculated, for example the test problem with 20 shots is designated as WELL20. Altogether we ran five test cases, WELL10, WELL20, WELL40, WELL60, and WELL80. To time the serial version of the code we ran each of these problems on 1 cpu of the 8 processor SGI.

We then tried various combinations of workstations on the network to test out the parallel version of the code. In the 5 CPU case, we used *baley* as the host and 4 cpus from the SGI 4D/380S (*ransgi*). In the 10 cpu case, we used *baley* as the host, 6 cpus on *ransgi*, and 3 cpus on the SGI 4D/240S (*trantor*). In the 15 cpu case, we used *baley* as the host, all 8 cpus of *ransgi*, all 4 cpus of *trantor*, as well as 2 SGI 4D35's, *ole* and *prufrock*. Table 3 contains the results from these timings.

The speedups for these tests are displayed in Table 4. We should note that for the 5 cpu case the maximum achievable speedup (indicated in parentheses) would be 4.7 due to the slower cpu *baley*. Similarly, the maximum speedups for the 10 and 15 cpu cases are 8.9 and 13.8 respectively.

Table 3: Timings in seconds for Parallel RAY3D on WELL test cases.

	Total CPUS			
WELL	1	5	10	15
10	793.0	273.4	127.4	
20	2108.1	505.3	304.2	217.0
40	5605.3	1143.6	649.3	461.1
60	10910.9	2529.7	1527.5	1063.7
80	24254.6	4908.6	2921.6	2089.5

Table 4: Speedups for WELL test cases.

	Total CPUS		
WELL	5(4.7)	10(8.9)	15(13.8)
10	2.5	6.2	
20	2.9	6.9	9.7
40	4.2	8.6	12.2
60	4.3	7.1	10.3
80	4.9	8.3	11.6

It is interesting to note that these test problems show an almost perfect linear speedup as the problem size grows. In certain cases the timings actually yield speedups greater than might be expected from using the equivalent cpu numbers. These differences are undoubtedly due to variations in the timing benchmarks though. Another point that is worth mentioning is that the total turnaround time can be substantially reduced by using this type of network. In the most realistic problem, the WELL80 test case, the serial version takes almost 7 hours to complete, while the parallel network version only takes about 35 minutes using 15 cpus.

Another measure of performance pertains to load balancing. Since the task of computing ray paths is more computationally intensive the deeper in the wellbore the source is, we allocated the jobs corresponding to the deepest locations first and worked our way up the wellbore to the easier jobs. Using this allocation scheme it was possible to achieve a very good load balance. In the WELL80 test case, for example, each one of the cpus used between 1964 and 2089 seconds, with a mean of 2010 seconds and a standard deviation of 42 seconds (a 2% deviation).

4.1 Heterogeneous network

We also ran some test cases using a heterogeneous network made up of SGI's and SUN's. We show the results for one of the test cases using a configuration with 20 cpus (*ransgi*, *trantor*, *ole*, *prufrock*, *baley*, *mom*, *matterhorn*, *solaria*, *hobbs*, *slice*) in Table 5. In this table we give the time to run these test problems on 1 cpu of the SGI 4D/380S as a comparison. The parallel version of this problem used a configuration with an equivalent cpu power of 15.7. Once again we see that as the size of the problem increases we get an almost linear speedup.

Table 5: Speedups for Heterogeneous network. Equivalent CPUS = 15.7.

WELL	ransgi	parallel	speedup
20	2108.06	344.26	6.1
40	5605.33	463.32	12.1
80	24254.6	1658.39	14.6

5. Conclusions

The process of parallelizing the RAY3D code was fairly simple using the combination of PICL calls and the TCGMSG toolkit. In addition, the use of this network computer approach takes advantage of computing resources that are readily available to most engineers and scientists. The next step of porting the code to a massively parallel machine should be straightforward given that the only parallel primitives used are the PICL calls. A side benefit of this approach is that most of the development and debugging of the parallel code can be done on the distributed network. Finally, we note that we achieved almost linear speedups for all of the larger problems, with the largest problem solved in a little over 30 minutes versus close to 7 hours with the serial version. Since the solution of one of these problems can be viewed as a single function evaluation in a three-dimensional inverse modeling problem, the use of parallel programming techniques appears to be promising for the efficient solution of this problem.

REFERENCES

[1] V. Pereyra, *Two-Point Ray Tracing in General 3D Media*, Geophysical Prospecting, 1992.

[2] R.A. Whiteside and J.S. Leichter, *Using LINDA for Supercomputing on a Local Area Network*, Proceedings of the IEEE Supercomputing Conference, Orlando, FL, November 14-18, 1988.

[3] A.H. Sherman, *C-Linda Reference Manual*, New Haven, CT, 1990.

[4] A. Beguelin, J. Dongarra, A. Geist, R. Mancher, V. Sunderam, *A USer's Guide to PVM: Parallel Virtual Machine*, Oakridge National Laboratories Technical Report ORNL/TM-11826, 1991.

[5] R.A. Harrison, *Portable Tools and Applications for Parallel Computers*, Intl. Journal of Quantum Chemistry, Vol. 40, 847-863, 1991.

[6] G.A. Geist, M.T. Heath, B.W. Peyton, and P.H. Worley, *A User's Guide to PICL: A Portable Instrumented Communication Library*, Oakridge National Laboratories Technical Report ORNL/TM-11616, 1990.

[7] M. Koshy, V. Pereyra, and J. Meza, *Distributed Computing Applications in Forward and Inverse Geophysical Modeling.*, presented at Society of Exploration Geophysicists 61st Annual Meeting, Houston, TX., Expanded Absttracts, pp 349-352 (1991).

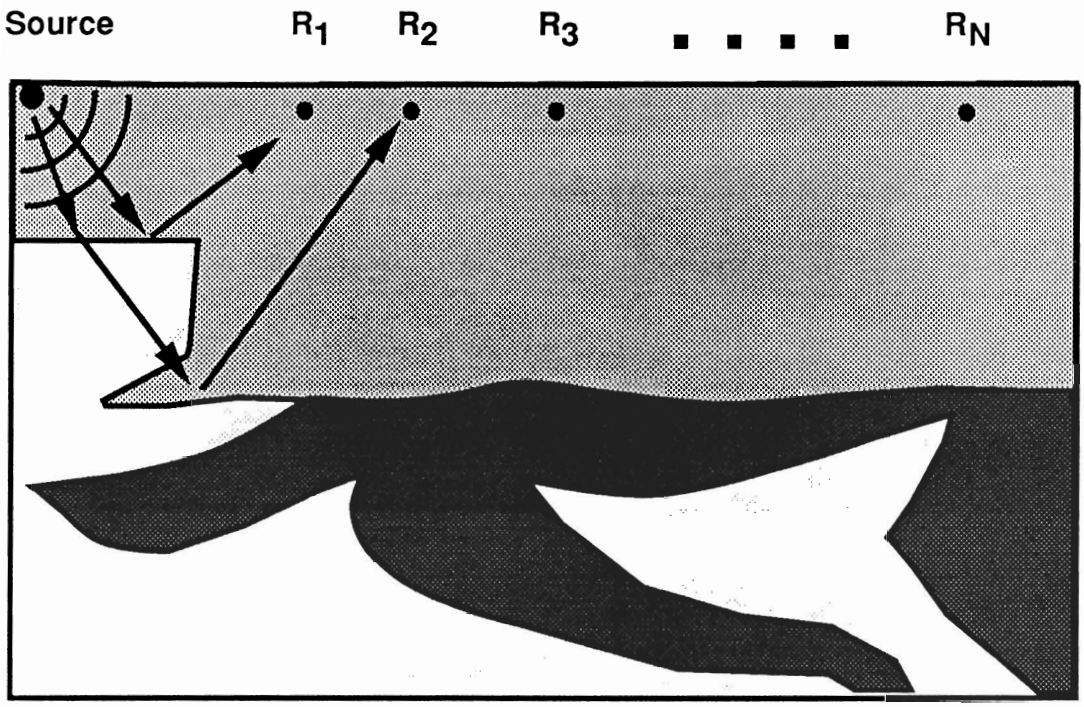

Figure 1: Vertical slice of a 3D overthrust model

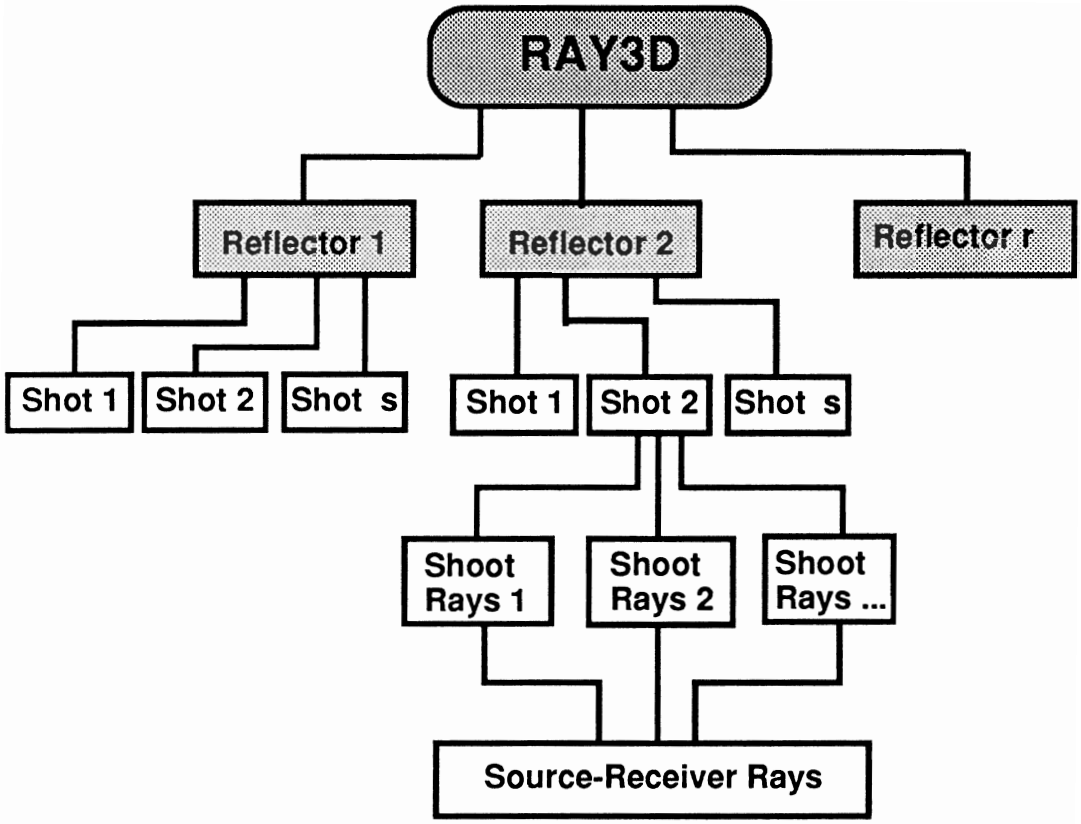

Figure 2: Reflection Seismic Overview

```
main(argc,argv)
int argc;
char **argv;
{

/*    Open communications */

  pbegin(argc,argv);
  open0( &nnodes, &me, &host);
  master = host;

  if ( me == master ) { /* Master node  */
    auto3dpar_(model, &lnmod, &nrefl, &nsrc, modelpath, &integ, ipref);

    alloc_jobs (model, nnodes, nrefl, nsrc, ipref, done,
                      fp, job_stats, node_stats);

    check_done (nrefl, nsrc, done, fp, job_stats, node_stats);
  }

  else {   /* Just a lowly worker   */

    ierr = worker(me, master);
    if (ierr != 0) fprintf(fp,"ray3main:  Error in Node %d.  \n", me);
  }

  close0(1);
}
```

Figure 3: Main routine

```
int worker(me, master)
int    me, master;
{

 start_parms inbuf;  /* structure containing inputs for ray3d */
 auto_parms  outbuf; /* structure containing output from ray3d  */

 while (1) {  /*  Hang around and wait for a start message  */

   msg0        = STARTMSG; inbuf_bytes   = sizeof(inbuf);
   recv0(&inbuf, inbuf_bytes, msg0);

   if (inbuf.istatus == START_UP) {

/*    Get reflector and shot number from inbuf and call the RAY3D program  */

     unpack_message(inbuf, &job_id, &irefl, &horiz, &ishot, &lnmod, model);

     ray3d(model, &lnmod, &irefl, &horiz, &ishot, &ntot,
                  &nshots, &nshotf, &n2ps, &n2pf, source, &cputime);

/*    Save results in output buffer and send done message to master */

     create_message(&outbuf, job_id, me, irefl, horiz, ishot, not,
                       nshots, nshotf, n2ps, n2pf, source, cputime);

     msg0  = DONEMSG;  outbuf_bytes = sizeof(outbuf);
     send0(&outbuf, outbuf_bytes, msg0, master);
   }

   else if (inbuf.istatus == QUIT) return(0);

   else return(-1);        /* Unknown message type */

   }
 }
}
```

Figure 4: Generic worker routine

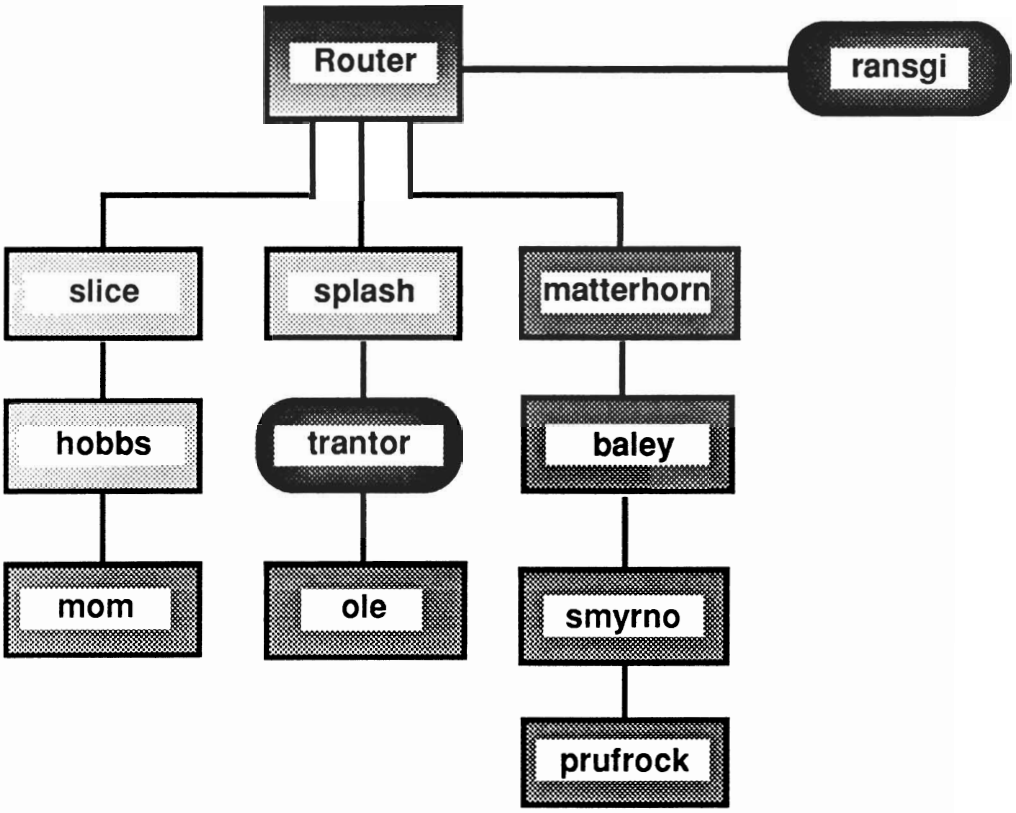

Figure 5: Configuration of Local Area Network