User guide to a simplified parallel code for Laplacian-fractal growth

Howland A. Fowler[*], Judith E. Devaney and John G. Hagedorn
High performance Systems and Services Division, ITL
National Institute of Standards and Technology
Gaithersburg, MD 20899

ABSTRACT

High-voltage breakdown in liquid dielectric is simulated as growth of a stochastic Laplacian fractal. The model is contained in a software package that is written in Fortran 90 with data parallel extensions for distributed execution. These extensions encapsulate an underlying, invisible message-passing environment (MPI), thus enabling the solution of memory intensive problems on a group of limited-memory processors. Block-partitioning creates processes of reasonable size, which operate in parallel like small copies of the original code. The user needs only to express his model in transparent array-directed commands; parallel interfacing between blocks is handled invisibly. Breakdown is performed in parallel, in each of the local blocks.

[*]Guest researcher

1 Introduction

This model simulates the growth of high-speed filamentary streamers, during high-voltage breakdown in liquid dielectrics.

Fortran 90 is used as a high-level parallel language for the code, supplemented by NIST's DPARLIB, a set of subroutines which extend F90 across block-process boundaries, providing an invisible interface to the Message Passing Interface (MPI).

Fortran 90's advantages are

- addressing huge arrays directly, so as to take advantage of the large individual-processor memory which is currently available on workstations and multiprocessors.
- allowing the assembly of program logic with combinations of real and logical arrays.
- it contains the powerful WHERE mask, CSHIFT, PACK and UNPACK, distributing and collecting operations.
- user needs to loop over arrays only when reading in and out.
- organization of the program into modules makes for easy substitution, compiling, and testing of code.

DPARLIB's extensions permit

- carrying F90 parallelism across block partitioning.
- taking advantage of MPI's ability to facilitate and extend C-shifts based upon a Cartesian-grid topology.
- The underlying MPI code is completely hidden; the resulting program is very similar to serial code; indeed, it is so executed by the individual processor.
- All processes start the same instructions together.
- A program may be built and tested on one processor, then scaled upward to execute over a 2-D array of process blocks.

Thus, we have a high-level scalable language: ***** Fortran 90 / DPARLIB / MPI ***** in which it is easy to model physics problems, test the (serial) code at small scale, then enlarge to full scale on a multiprocessor (for example IBM SP2, SGI Onyx, SGI Origin [1]). The language is particularly well-suited to problems which are easily expressed on a large Cartesian grid, since MPI is adapted for communicating data across rectangular block boundaries.

The model is a routine for the stochastic growth of Laplacian fractals on large 3-D Cartesian grids [1]. While the intended application has been to simulate global features of the growth of fast streamers in liquid dielectrics, the family of possible application areas is larger than this (water-treeing in solid dielectrics, growth of metallic structures in electrodeposition, surface streamer-spreading at dielectric interfaces are examples). Each would require some specialization of parameters and boundary conditions. These routines are presented as expository examples.

The elements of the algorithm are:

1. Assume streamer tree either fully conductive or with a retained voltage gradient, and attached to the anode electrically.

2. Solve Laplace's equation throughout the interior region, using the anode and streamer tree as one boundary, and the cathode as the counter-electrode boundary.

3. Examine neighbor sites to the tree. If phi is above threshold (cutoff) level, then compare against weighted random numbers. If they exceed, attach to tree.

4. Cycle until counter-electrode is reached.

Adjustable parameters of the calculation (grid bounds, needle position, threshold (cutoff) voltage level, and power-law exponent) allow a broad range of fractal behavior to be approximated.

Novel features of the realization include:

- Concurrent (simultaneous) growth is distributed over the entire tree at any instant

- Simulated time progression (Monte-Carlo time ticks) is recorded. When time-compression is used to avoid empty statistical trials, the Monte-Carlo time is estimated.

[1]Certain commercial equipment, instruments, or materials are identified in the paper to foster understanding. Such identification does not imply recommendation of endorsement by the National Institute of Standards and Technology, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

- Full three-dimensional display of the growth process (animated or color-banded) is possible.
- Compact procedures are used for statistical testing and for time compression.
- Face- and body-diagonal links may be included in the fractal tree.

The code is presented as a directory of modules (cadmus/source), see Figure 1, which are compiled by Makefile under four separate main programs (**diagone, diagtwo, diagthree, diagfour**), each of which performs a different task. **diagone** makes a comparison against a linear-weighted uniform random-number distribution, and counts the number of statistical tries (Monte Carlo time ticks). **diagtwo** uses a square-law-weighted random-number distribution, and counts statistical tries. **diagthree** and **diagfour** start with very low probabilities. In these latter two programs, a normalization is used which corresponds to time compression for low probabilities, PACKing and weighting the neighbor-voltage distribution, and then comparing against a uniform random number distribution, compressed when the voltage values are low (as at the outset). In such cases we estimate the progression of statistical tries, instead of counting. This permits cube-law and higher powers to be tested, without a very large number of "empty" statistical tries. It has proved useful for tracing the fractal shape of cube-law and fourth-power trees. **diagthree** and **diagfour**contain an estimating formula for the value of Monte-Carlo waiting time between events (important when the field strengths and discharge probabilities are low) as presented by I. Beichl and F.E. Sullivan [2].

The routines have been run on both multiprocessors (IBM SP2, SGI Onyx, SGI Origin) and open clusters of SGI workstations – sometimes referred to as networks of workstations (NOWs) – linked by LAM, with access to a common memory-server.

For grid sizes up to 64 X 64 X 64, workstation clusters or running on a time-shared Onyx are satisfactory for development work, with interactive runs. For large grid sizes, namely 100 X 100 X 100 or 128 X 128 X 128, the limitations are set by the burden of repetitive computation of Laplacian convergence. The number of available dedicated processors will determine the duration of the run, and will have to be determined by the local batch queueing protocol.

Program Requirements:

These routines require both MPI and the DPARLIB library. On an SP2, MPI can be the IBM Parallel Operating Environment (POE) MPL or a public-domain MPI like LAM [3] or MPICH [4]. On an SGI, MPI can be SGI's MPI or a public-domain MPI. DPARLIB can be obtained from our web site:

`http://www.itl.nist.gov/div895/sasg`

2 Details of the Routines

The main program (typ. diagone.f90) calls routines from the other modules. These are compiled in the Makefile, in the order in which the main requires them. Note that the Makefile needs object code for DPARLIB and MPI to be in place, in order to compile, and may need to be modified to state the correct DPARLIB location on the local system. In the listings below, modules also carry the suffix [underscore plus mod].

MAIN programs:
diagone — calls **diagcomp**;
diagtwo — calls **diag2comp**;
diagthree — calls **diag3comp**;
diagfour — calls **diag4comp**.

The Makefile is arranged to compile and link individual modules in the proper order. It is only necessary to call "make diagone", for example, to compile from scratch or to recompile. When recompiling, to ensure a "clean" result, it is often advisable to remove all object code and interface files (the suffix for these will vary with the compiler used), and to restart the whole compile from only the source code. Because the code is straightforward, it requires only a few seconds to recompile.

Setup, initializing modules:
globals; **breakup**; **growinit**.

Display modules, printing to screen:
treedisp; **nbrdisp**.

Program logic and arithmetic subroutines, in their modules:
diagnbrset in **diagnbr**;
laplace, lplinit in **lpl**;
dgcomp in **diagcomp**;
dg2comp in **diag2comp**;

dg3comp in **diag3comp**;
dg4comp in **diag4comp**.

The modules contain data, assignments, and subroutines as follows:

globals reads from the input file 'breakdown.in' the integers decribing the desired array of process blocks, and the dimensions of the desired rectangular grid (the latter must be an even number in the c- or z-direction, along which the blocks are partitioned. The global z-dimension, divided by the number of process blocks, must also be an even number, in order for the red and black subgrids to match up at the boundaries.) It also contains a list of 32 seed integers for the random number generator dp-uni. In order to create a new run, alter the lead integers in the list and recompile. Only as many seed integers will be used, as number of processes.

breakup allocates major arrays within the bounds of the individual process blocks. It also contains a routine to assign the red and black subgrids.

growinit starts with the subroutine **gradient**, which fills the interior volume with a linear-gradient voltage field 'phi'. This is a zeroth approximation which reduces later convergence time. It then allocates and initializes the 'tree' array, including the anode plate and the concentrating needle. It allocates and initializes the cathode plate. It runs a test round-robin access sequence to the output file streamer (if the empty streamer file is not present, the program will stop). The subroutine writestream will be used for output.

treedisp contains a test routine for displaying to screen one plane of the 'tree' array. When used, it displays, successively, all the planes in one of the center-most blocks of the calculation, with +'s and .'s to mark the disharged and empty sites. (Check to make sure that the starting needle is located within the appropriate block.)

diagnbr finds all neighbor positions which are one grid step away from the tree array as then constituted, identifying possible candidates for breakdown in the next statistical try. Outputs from the **diagnbrset** routine are the position (logical) array nbrs, the (directional index) array intnbrs, on those positions, and the real arrays vdiff (voltage difference along candidate link, down-weighted by factors 1/2 and 1/3 in case of diagonal paths), and vold (voltage at tree end of candidate link). All these arrays will be needed for the statistical filter stage of growth.

Neighbors are listed in three groups: cube diagonals, face diagonals, and edge neighbors, with the shorter paths taking precedence in the final list. Thus we avoid more than one path to the same new candidate site.

nbrdisp contains a test routine for displaying one plane of the nbrs array. As above, it may need adjustment before use.

Method: At each growth stage, the neighbors are chosen by combinations of dpcshifts on the existing tree. The path direction to each individual site is recorded by an integer on the corresponding site in the array 'intnbrs'. Integers 1-6 are used for the edge-neighbors; 7-18 for the face-diagonal neighbors; and 19-26 for the body-diagonal neighbors. The order of selection, with shorter paths overwriting longer ones, eliminates ambiguities.

The array 'intnbrs' is passed to the statistical comparison routine, which receives the weighted 'vdiff' values as described above, before making the statistical tests.

lpl contains the subroutine **laplace** which does the major repetitive calculation of the voltage field, by Gauss-Seidel over-relaxation. Epsilon is set at 0.0050; but this may be changed. Setting epsilon to 0.0010 will roughly double the number of convergence loops to reach completion. Also here is subroutine **lplinit** for initialization; this is set to run for 200 loops instead of using the epsilon.

The Dparlib random number generator is used; it starts from a separate seed within each process and creates independent streams of random numbers.

Both **laplace** and **dgcomp** terminate with global-count if loops which ensure that the step into the alternate subroutine is performed simultaneously in all processes.

Output into the 'dgstreamer' file is a six-column list; three position integers followed by the breakdown voltage at that site, the statistical trial number (Monte Carlo time tick), and the directional index (1-26).

dgcomp is the statistical-choice routine for the **diagone** main program. It differs from **dg3comp** in counting all statistical trials, instead of just estimating the ones with low probability. No PACK routine is called. The neighbor phis are compared against a uniform distribution of random numbers, at their sites.

dg2comp is the corresponding routine for the **diagtwo** main program. Instead of exponentiating the field to power 2, as in **dg3comp**, it finds a MAX array from two linear arrays, each filled with a uniform distribution of random numbers. This is equivalent to conditional(or product) combination, passing the distribution of voltages at the neighbor positions over a "double hurdle" of filters, and leaving a square-weighted distribution of survivors.

dg3comp is the statistical-choice routine for the **diagthree** main. (The main action alternates between **dg3comp** and **laplace**.) **dg3comp** identifies neighbor candidates whose voltages exceed the 'cutoff' threshold, PACKs their values on ‘philin’ array, exponentiates that to ‘pwr’ onto ‘phisq’ 1-dimensional array, and sums the elements of ‘phisq’ to obtain the a-priori expected probability of any event occurring. This sum is used to weight a uniform random distribution, for test comparison against ‘phisq’. Survivors are UNPACKed to their original positions, forming the array ‘nutree’, which is combined with ‘tree’ as the growth step.

dg3comp dnd **dg4comp** utilize the direct BKL approximations for estimating the number of “empty trials” to reach a composite probability of one [2]. Thus, time compression is accompanied by an estimate of the “waiting time” which was avoided.

The main routines are set up to call **nbrdisp** and **treedisp** at convenient intervals during an interactive run. We print a lot of numerical information at the steps of the statistical routine, which can be read to tell the status of the run. When not desired, these printout instructions may be commented out; or, their output may be led to a null file.

3 Input required from user

The program looks for an input file ‘breakdown.in’ which must contain six lines of integers:

Line 1 gives the global upper bounds of the grid, in three columns

Line 2 gives the 2-D array configuration of process blocks

Lines 3, 4 and 5 give the global bounds of the anode needle: X or a direction in line 3 is direction of growth,anode to cathode. Y or b direction, line 4, is orthogonal to this, across the growth axis. Z or c direction, line 5, is also orthogonal; this is the direction in which the process blocks are stacked.

4 Guide to Operating

Before starting each run, make sure that an empty file ‘dgstreamer’ exists, to catch the listing of breakdown sites from the individual processes. The unix commands “rm dgstreamer” followed by “touch dgstreamer” will do this job. In its present realization, the programs assume that all processes will have access (round-robin) into a common file; this is especially convenient if the run happens to be interrupted, because it will contain the full growth history of the streamer, up to that moment.

If the processes cannot communicate to a common file, then the output instructions should be adjusted to form a streamer file for each process, which will dump at the end into a common 'dgstreamer'. This should then be sorted by statistical trial number (Monte Carlo time tick) to arrive at the correct sequence.

If a new selection of random-number seeds is desired, change the leading integers in the data list of 32 seeds, which is supplied in globals, and recompile. Be sure that there are 32 in the list after you modify it.

If a different value of cutoff (threshold phi) is desired, change the assigned value of this parameter at the beginning of **dgcomp** (or **dg2comp, dg3comp, dg4comp**) and recompile. Likewise,the value of gradient multiplier (fraction of the neighbor voltage causing discharge, to be retained as voltage gradient in the streamer tree) should be reset to desired value where it occurs (twice) near the end of the statistical comparison routine. If no net retained gradient is desired, set this to 0.0. Recompile. If a value of convergence epsilon in the Laplacian calculation, other than 0.0050 is desired, change this assigned value at the start of subroutine **laplace**, in **lpl**, and recompile.

Subroutines **showtree** and **shownbrs** are useful in early testing of the program on small grids, say $48 \times 48 \times 48$. To use **shownbrs** the integer k is set to the z-plane in which the anode needle is located (usually identified with the central-most process block), and the routines produce a screen display of +'s and .'s, to indicate the subarray of tree or nbr sites in that plane only. As a general rule, they are not needed in larger runs – the system will print (to screen) enough data to show the progress of the run.

5 Graphics for display of the breakdown tree

The 6-column 'streamer' list file generated by Cadmus runs must be post-processed to render it in 3-D graphical format. The information reads as follows: 3 columns for position of new site (first column is the coordinate in the growth direction, third column in the direction normal to the separation planes between processor blocks); fourth column, voltage difference from the tree to the new site; fifth column, cumulative statistical try number; sixth column, directional index for the link to the new site.

Three programs are used: **dgldrread** prepares two files, 'newtree' and 'ldr', which can be read in black and white line display (fixed angle) by PVWave graphics routines, several examples of which are provided, having the suffix .pvw. For a system provided with PVWave support, the calls are then "wave", and ".RUN xxx.pvw", where xxx stands for the name of the specific PVwave program. 'newtree' is six-column, containing the 'move' and ‘draw’ coordinates for the ends of each link. The specific PVWave routines call up cubes of various sizes, with different needle lengths.

dgldrread makes a backward search to find the link connections implied by the ‘dgstreamer’ listing, and then lists all of these links. Thereupon, it traces back through the array of the links to find the continuous path from cathode to anode, which it lists in ‘ldr’file, to be called in a heavy-weight line display. The PVWave routines read the links and the continous path from the newtree and ldr files.

NOTE: In order for this leader-trace function to proceed correctly, the discharge of the site in the plane next to the cathode must be in the last line of the ‘dgstreamer’ file. (In cases where many sites discharge at once in the last statistical try, this may have to be adjusted manually.)

streamer2avs calls a routine which resets the coordinates of the cadmus cubic coordinates to those used by the AVS graphics system. It outputs a ‘newtree_avs’ file which must then be processed by “de2ucd newtree_avs”. The latter performs a sort on the statistical try variable, and restates the data in the ucd format, which enables AVS to pick it up rapidly. Output is in the file ‘newtree_avs.inp’, which can be retitied ‘xxx.inp’ to suit the occasion, for retention.

The sample AVS network, ‘dbdiag.net’, which is included, will then display the streamer in its cube of coordinates. (The outlines of the cube and needle are read in separately, and must be adjusted for the problem at hand). Check the submenu boxes in the ”read ucd” subroutines to see which files are being read.

6 Preparing graphics

Copy the ‘dgstreamer’ file from the current run (and make a permanent backup for later use.)

Examine the tail of the ‘dgstreamer’ file. There will be one (or sometimes more) discharge sites in the plane adjacent to the cathode. These will show a voltage-difference reading considerably higher than the others because of the short remaining gap. One of these should be made the last line in the file (either move it, or erase the lines which follow it), in order for **dgldrread** to trace the leader path back correctly.

Call “dgldrread”. Call “wave”, “.RUN xxx.pvw”. PVWave will display a perspective white-on-black view of the tree. On Silicon Graphics machines, use snapshot and swapbw to obtain black-on-white. If Showcase is available, it is convenient for reshaping and printing.

Call "streamer2avs". Call "de2ucd streamer_avs" to form a similar-named file with .inp suffix. AVS can call the sample.net to display in its 3-D geometry viewer the cube and streamer, which will be color-banded to represent Monte-Carlo timing. Rotation and animated growth may be demonstrated. In the color-banded version, we do not display the heavy leader path for fixed printout. However, if one is doing an animated demonstration, it may be called from the 'ldr' file, above, as a separate display element in the last few frames.

For each discharged link, the directional index integer is recorded in a separate column of the 'dgstreamer' file (note the change from 'streamer' label). The post-processing routine **dgldrread** translates this into the appropriate 'newtree' and 'ldr' files for graphical display as before.

References

[1] H. A. Fowler, J.E. Devaney, J.G. Hagedorn, and F.E. Sullivan, "Dielectric Breakdown in a Simplified Parallel Model", NISTIR No. 6174, June 1998.

[2] I. Beichl and F.E. Sullivan, "(Monte Carlo) Time after Time", Computational Science and Engineering 4, no. 3 (July-Sept. 1998), pp.91-94.

[3] D. Burns and R.B. Daoud, "Lam; an open cluster environment for mpi", in Supercomputing Symposium '94, June 1994, Toronto, Canada. Code available at http//www.osc.edu/lam.html.

[4] See http://www.mcs.anl.gov/Projects/mpi/mpich/index.html

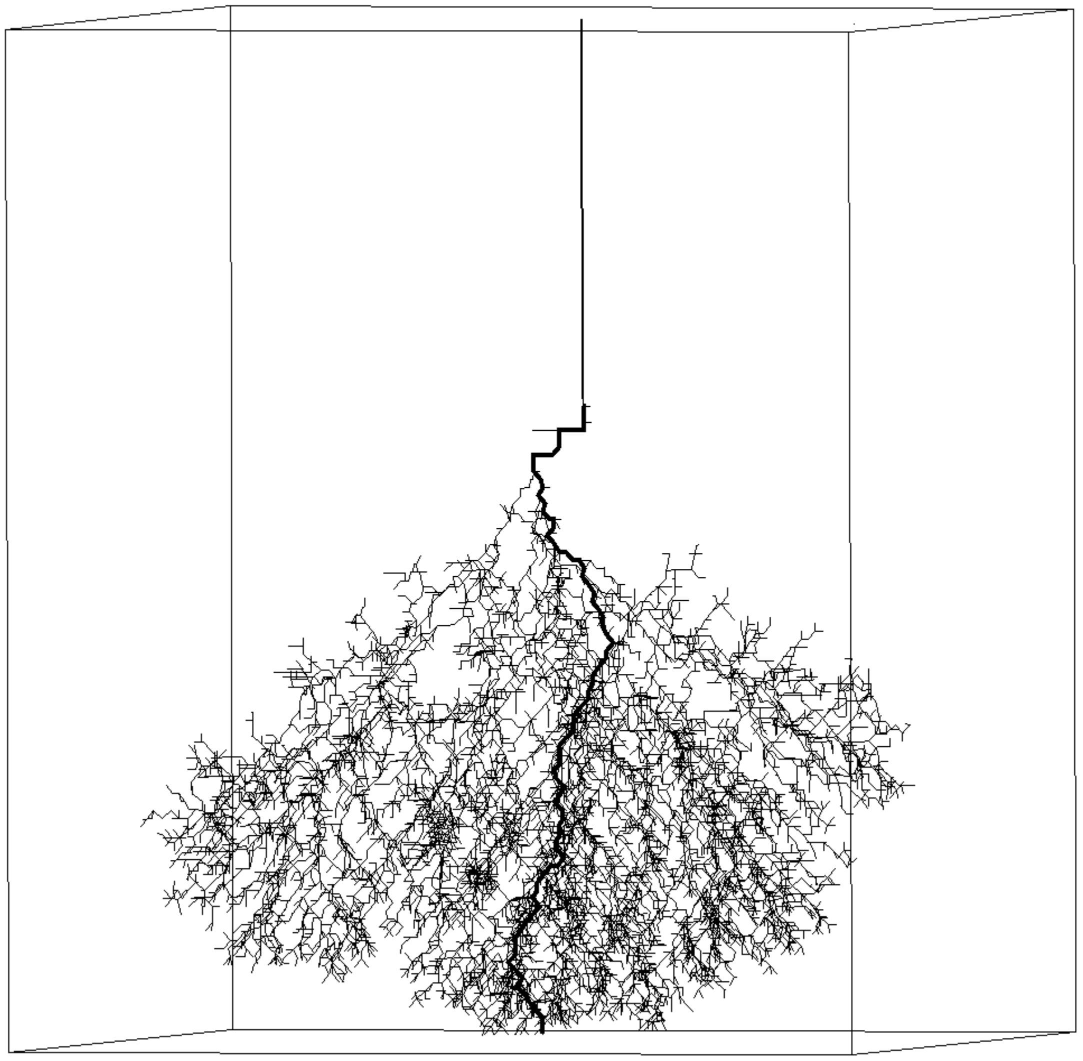

Figure 1: Sample of PVwave black-and-white graphics