propag-4

STATUS — Saving and reloading status dumps

Version 4.67 – December 23, 2008

How many it had cost in the amassing, what blood and sorrow, . . .
what shame and lies and cruelty . . .

—Robert L. Stevenson, *Treasure Island*

1. Introduction. This is one of the documents describing propag-4, a program for large-scale cardiac simulation, intended to run a model of the human heart. See the document entitled PROPAG for a description of its purpose and method of documentation, or our published papers for a more general introduction [trudel:prop,potse:bidofex] and applications [potse:isch].

The functions described in this document write and read status dumps for `propag`. Status dumps or "checkpoint files" can be used to recover from an interruption, or to start a simulation with a given non-resting state. The status dumps can also be read by the `prundump` program, which translates them into a set of IGB files for (post-mortem) analysis.

There are several files associated with the status dumps. Two sets of dump files are written alternatingly, so that there will always be a complete set even if the program is interrupted while writing a dump. After successful completion of a dump, the other dump is deleted. A single *pointer file* indicates which of the two dumps is the most recently completed. Each dump consists of a small *master file*, written by the master thread, and several *part files*, each written by its own thread. The master file describes how many part files there are and what each of them contains.

The save function was parallelized, because it runs several times during a simulation and can take 10–20 minutes to do its work in serial mode, due to the large amount of data to be compressed by a single thread, and to the transport of data from remote memory. To facilitate parallel work, each thread dumps to its own file. The load function is parallelized as well. The number of threads may be different in the writing and reading runs.

The following program parameters (global variables) influence the code in this document.

- *p_interrupted* determines if the program should start from a status dump file; it is used here indirectly to set the *required* argument to *load_status*
- *status_dump_interval* determines the frequency of dumps; it is used by the main program
- *fich_var_w* determines the name of the dump files
- Either *fich_var_w* or *boostfile* is passed to *load_status* in the *dump_name* argument
- *fich_var_ext* gives the filename extension for the dump files
- *compress_status_files* determines the compression level for the dump files.

```
⟨ status.h 1 ⟩ ≡
  //$Id: status.web,v 4.67 2008/12/15 01:48:41 potse Exp $
  ⟨ Preprocessor definitions ⟩
    void save_status(double stime, float ecoule_o);
    int load_status(char *dump_name, double *stime, float *t_ecoule, int required);
```

2. Zlib functions are used to compress and decompress the dump files.

```
⟨ status.c 2 ⟩ ≡
  //$Id: status.web,v 4.67 2008/12/15 01:48:41 potse Exp $
  ⟨ Preprocessor definitions ⟩
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <limits.h>
#include <time.h>     /* time, difftime */
#include <errno.h>
#include <zlib.h>     /* gzopen, gzwrite, documentation of zlib */
#include "propag.h"
    ⟨ local declarations 3 ⟩
    static int Ndumps = 0;     /* shared between the two functions */
    char *l_ext = Λ;     /* file extension of lastly loaded dump */
  ⟨ private functions 5 ⟩
  ⟨ implementation of save_status( ) 4 ⟩
  ⟨ implementation of load_status( ) 14 ⟩
```

3. File format. The documentation of the file format consists only of the implementation here, and we take no care to keep it standardized. The format has changed often, and may change again. To reduce the probability of errors, all small fixed-size things are stored in a struct that is written and loaded at once, in/from the master file. Variable-size arrays are implemented separately.

⟨ local declarations 3 ⟩ ≡

```
typedef struct master_info {
    int Nverts;    /* nr of vertices (Nverts in propag) */
    int Nnodes;    /* nr of nodes (Nnodes in propag) */
    double stime;    /* simulated time (simtime in propag) */
    int compteur;    /* step counter (compteur in propag) */
    int temps;    /* lap counter (temps in propag) */
    int stade;    /* depolarization/repolarization (stade in propag) */
    float ecoule;    /* wallclock time passed (ecoule_o in propag) */
    int Ndumps;    /* nr of dumps already made */
    int Nparts;    /* nr of parts (= nr of threads in writing run) */
    int domi;    /* mono/bi/tridomain (domi in propag) */
    int ve_valid;    /* true if computation of Vex completed */
} master_info;
```

This code is used in section 2.

4. Writing. The function first writes a "master output file" containing some scalars, including the number of "part files." Then, in parallel mode, each thread writes a part. If anything fails, the whole dump will be removed and the function returns. This may save the day if the filesystem is nearly full. If the dump succeeded, the pointer file is updated. Finally, if that succeeded too, the previous dump is removed.

The *Ndumps* variable counts the number of dump attempts and is used to implement alternation between two dump areas. We choose to update it also after a failed attempt. This will cause the next attempt to destroy the only remaining dump. Thus we take the risk of having no valid dump, rather than leaving an old dump around. If the dump failed because of a full disk, overwriting the previous dump may still succeed.

```
⟨ implementation of save_status( ) 4 ⟩ ≡
  void save_status(double stime, float ecoule_o)
  {
    FILE *fm;   /* Master output file; either fm or zm is used */
    gzFile zm;
    int full = 0;   /* flag */
    int cmpr = compress_status_files;   /* compression level (parameter) */
    char *pfx = fich_var_w;   /* output file prefix (program parameter) */
    char *fn_ext;   /* output file extension */
    static char fn_pointer[256], fn_master[256], fn_prefix[256];
    logline("DUMP", "save_status␣$Revision:␣4.67␣$");
    if (cmpr ≡ 0) fn_ext = fich_var_ext[0];
    else fn_ext = fich_var_ext[1];
    if (¬l_ext) l_ext = strdup(fn_ext);
    snprintf(fn_pointer, 256, "%s/%s.ptr", dirname, pfx);
    snprintf(fn_master, 256, "%s/%s%d%s", dirname, pfx, Ndumps % 2, fn_ext);
    snprintf(fn_prefix, 256, "%s/%s%d", dirname, pfx, Ndumps % 2);   /* for parts */
    logline("DUMP", "saving␣status␣in␣`%s',␣compression␣level␣%d", fn_master, cmpr);
    if (open_file(&fm, &zm, fn_master, cmpr)) full = 1;
    ⟨ write header 8 ⟩
    ⟨ write master info 9 ⟩
    close_file(fm, zm, cmpr);
    if (¬full) ⟨ write part files 10 ⟩
    if (full) {
      Warning(1, "cannot␣write␣status␣in␣'%s';␣deleting␣it.␣Disk␣full?", fn_master);
      remove_dump(fn_prefix, fn_ext);   /* remove this one */
    }
    else {
      ⟨ write pointer file; return if it fails 12 ⟩
      snprintf(fn_prefix, 256, "%s%d", pfx, (Ndumps + 1) % 2);
      remove_dump(fn_prefix, l_ext);   /* remove the other */
    }
    Ndumps++;
  }
```

This code is used in section 2.

5. The *cmpr* argument indicates the compression method/level to be used for status dumps. Zero means no compression; a value between 1 and 9 (inclusive) means using gzip with compression level *cmpr*. Other methods can be added. For all values from 1 to 9, we use the zlib interface (documented in `/usr/include/zlib.h` on most systems). Given a value 0 for the compression level, the zlib functions will create an uncompressed file, but *with a gzip header*; the gzip format includes a non-compression. Since it may be confusing to have files with `.gz` extension but no compression around, we handle the case of $cmpr \equiv 0$ with ordinary stream output.

We use zlib rather than opening a pipe to `gzip` because a status dump may be triggered by a signal, e.g. `SIGTERM`, and we have no control over signal handling in `gzip`, which means in practice that it terminates if it receives the signal. Termination on `SIGTERM` is quite common when working with a queuing system like PBS, which may send such signals when a job runs too long or uses too much memory.

⟨ private functions 5 ⟩ ≡

```
static int open_file(FILE **fp, gzFile *zf, char *name, int cmpr)
{
  int c = 0;
  if (cmpr ≡ 0) {
    if (¬(*fp = fopen(name, "wb"))) {
      Warning(2, "Cannot␣open␣file␣<%s>", name); return 1;
    }
  }
  else if (cmpr > 0 ∧ cmpr < 10) {
    if (¬(*zf = gzopen(name, "wb"))) {
      Warning(2, "Cannot␣gzopen␣file␣%s:␣%s", name, gzerror(*zf, &c)); return 1;
    }
    c = gzsetparams(*zf, cmpr, Z_DEFAULT_STRATEGY);
    if (c ≠ Z_OK) {
      Warning(1, "zlib␣error:␣%s", gzerror(*zf, &c)); return 1;
    }
  }
  else {
    Warning(1, "unknown␣compression␣code␣%d", cmpr); return 1;
  }
  return 0;
}
```

See also sections 6 and 11.

This code is used in section 2.

6. ⟨ private functions 5 ⟩ +≡

```
static int close_file(FILE *fp, gzFile zf, int cmpr)
{
  int c;
  if (cmpr ≡ 0) {
    c = fclose(fp);
    if (c) {
      Warning(1, "closing␣status␣output␣file␣failed"); return 1;
    }
  }
  else {
    c = gzclose(zf);
    if (c ≠ Z_OK) {
      Warning(1, "zlib␣error:␣%s", gzerror(zf, &c)); return 1;
    }
  }
  return 0;
}
```

7. This macro is used for all output. We cannot use goto or jump in case of errors because it's used in a parallel section, so we'll just raise the *failure* flag. This flag is passed as a parameter because it differs between serial and parallel code.

```
#define test_write(fp, zf, failure, src, Nbytes)
        {
          if (¬failure) {
            if (cmpr ≡ 0) {
              if (fwrite((src), 1, Nbytes, fp) ≠ Nbytes) failure = 1;
            }
            else {
              if (gzwrite((zf), (src), (Nbytes)) ≠ (Nbytes)) failure = 1;
            }
          }
        }
```

8. The master file has a 1024-byte header. The first line of this header allows identification. It must be changed when the format of the dump file changes incompatibly.

```
#define HEADER_CHARS 1024
#define ID_STRING "PROPAG␣status␣dump␣version␣4.3"
⟨ write header 8 ⟩ ≡
  {
    int i;
    char s[HEADER_CHARS];
    for (i = 0; i < HEADER_CHARS; i++) s[i] = '␣';
    s[HEADER_CHARS − 1] = '\n';
    sprintf(s, "%s\n", ID_STRING);
    sprintf(s + strlen(s), "propag␣version␣%s\n", version());
    sprintf(s + strlen(s), "Nverts␣=␣%d\n", Nverts);
    sprintf(s + strlen(s), "Nnodes␣=␣%d\n", Nnodes);
    sprintf(s + strlen(s), "t␣=␣%f\n", stime);
    sprintf(s + strlen(s), "anatomy␣from␣%s\n", fich_cell);
    test_write(fm, zm, full, s, HEADER_CHARS ∗ sizeof(char));
  }
```

This code is used in section 4.

9. The master information includes the number of parts and the part limits, so that the reader knows how many files to read and how much to read from them. We also write *Ndumps*, so that the alternation between the two dump areas can continue in the next run. The *domi* variable determines if *Vex* is written; the reader will have to know this too.

The *stade* parameter replaces *dt*; it allows for an unambiguous identification of the stage, which is necessary because for example *inc_e1* may be different from *inc_e2* while $dt1 \equiv dt2$.

⟨ write master info 9 ⟩ ≡

```
{
  master_info minfo;
  minfo.Nverts = Nverts;
  minfo.Nnodes = Nnodes;
  minfo.stime = stime;
  minfo.compteur = compteur;
  minfo.temps = temps;
  minfo.stade = stade;
  minfo.ecoule = ecoule_o;
  minfo.Ndumps = Ndumps + 1;   /* Ndumps is not yet updated */
  minfo.Nparts = Nthreads;   /* each thread (c.q. domain) writes a part */
  minfo.domi = domi;
  minfo.ve_valid = ve_valid;
  test_write(fm, zm, full, &minfo, sizeof (minfo));
  test_write(fm, zm, full, domain_begin_vertex, Nthreads * sizeof(elem_t));
  test_write(fm, zm, full, domain_end_vertex, Nthreads * sizeof(elem_t));
  test_write(fm, zm, full, domain_begin_node, Nthreads * sizeof(elem_t));
  test_write(fm, zm, full, domain_end_node, Nthreads * sizeof(elem_t));
}
```

This code is used in section 4.

10. The *yyy* array must be written in parts, because it can be larger than 2 GB and the size argument to *gzwrite*( ) is **int**.

Test: make the bites quite large (rather than one cell at a time) to see if this is faster on the MP cluster. This should help if it is the latency of the write operation that makes things slow.

⟨ write part files 10 ⟩ ≡

```
{
#pragma omp parallel
  {
    long bite, c, n0, Nn, v0, Nv, tr = omp_get_thread_num();
    char fname[200];
    FILE *fp;
    gzFile zp;
    int pfail = 0;    /* local failure flag, set nonzero if part fails */
    snprintf(fname, 200, "%s.%04d%s", fn_prefix, (int) tr, fn_ext);
    if (open_file(&fp, &zp, fname, cmpr)) pfail = 1;
    n0 = domain_begin_node[tr];
    Nn = domain_end_node[tr] − domain_begin_node[tr];
    v0 = domain_begin_vertex[tr];
    Nv = domain_end_vertex[tr] − domain_begin_vertex[tr];
    test_write(fp, zp, pfail, dtime + n0, Nn * sizeof(float));
    test_write(fp, zp, pfail, Vmem + v0, Nv * sizeof(vm_t));
    if (domi > 1) test_write(fp, zp, pfail, Vex + v0, Nv * sizeof(vm_t));
    if (mem_y) {    /* yyy not allocated in forward model */
      bite = 1024 * 1024;
      for (c = 0; c < Nn; c += bite) {    /* must break it for gzwrite() */
        if (bite > Nn − c) bite = Nn − c;
        if (¬pfail) test_write(fp, zp, pfail, &yyy(n0 + c, 0), bite * nsvar * sizeof(yyy_t));
      }
    }
    if (close_file(fp, zp, cmpr)) pfail = 1;
    if (pfail)
#pragma omp critical
      full = 1;    /* communicate to master */
  }
}
```

This code is used in section 4.

11. We don't know how many parts were written last time, since it may have been done with a different number of threads. So we just try to remove every possible part file, and don't complain if it does not work. It's no use complaining about that anyway. A warning is written if removal of the master file does not succeed for any but the first dump (*Ndumps* > 0).

⟨ private functions 5 ⟩ +≡

```
static void remove_dump(char *prefix, char *ext)
{
  char name[256];
  int i, c;

  snprintf(name, 256, "%s%s", prefix, ext);
  logline("DUMP", "removing old dump %s...", name);
  c = remove(name);
  if (c && Ndumps > 0) Warning(1, "cannot remove %s", name);
  for (i = 0; i < 1000; i++) {
    snprintf(name, 256, "%s.%04d%s", prefix, i, ext);
    remove(name);   /* don't care if it works */
  }
  logline("DUMP", "done.");
}
```

12. The pointer file. At the very end of *save_status*, and only when we believe that the dump really succeeded completely, the prefix of the file names is updated in the pointer file. This update is the only operation in the dumping process that should be atomic. Of course we cannot guarantee this, but we must make it as close to atomic as possible. If the update succeeded, the function goes on to remove the other dump. If the update failed, the function returns immediately, before removing anything.

The pointer file contains two lines: the first for the prefix, and the second for the extension. Concatenated, these two strings form the name of the master file. The extension must be communicated in this way because it depends on *compress_status_files*, which may be zero in one run and nonzero in another, leading to different extensions. Since this affects the name of the master file, the extension has to be stored in the pointer file.

⟨ write pointer file; **return** if it fails 12 ⟩ ≡

```
{
  FILE *fo;
  int len;

  if (!(fo = fopen(fn_pointer, "wt"))) {
    Warning(1, "cannot open dump pointer file %s for writing", fn_pointer);
    return;
  }
  len = strlen(fn_prefix) + strlen(fn_ext) + 2;
  if (fprintf(fo, "%s\n%s\n", fn_prefix, fn_ext) < len) {
    Warning(1, "failed to write in dump pointer file %s", fn_pointer);
    return;
  }
  if (fclose(fo)) {
    Warning(1, "error while closing dump pointer file %s", fn_pointer);
    return;   /* assume the write failed */
  }
}
```

This code is used in section 4.

13. Any problem in reading the file is a fatal error, except for nonexistence of the pointer file: When this occurs, it is most probably because we are starting a new simulation and propag was running with *p_interrupted* set to "auto". This warrants a special return value, to avoid scaring the user with meaningless warnings.

The *l_ext* variable will contain the extension of the loaded dump. This may be different from *fn_ext* in *save_status*. *l_ext* is used by *save_status* to remove the old dump after writing the new.

⟨ read pointer file 13 ⟩ ≡

```
{
  FILE *fi;
  if (¬(fi = fopen(fn_pointer, "rt"))) {
    if (¬required) return 2;
    Trouble((1, "cannot␣open␣dump␣pointer␣file␣%s␣for␣reading", fn_pointer));
  }
  l_ext = MALLOC(64, char, "");
  if (fscanf(fi, "%s%s", fn_prefix, l_ext) ≠ 2) {
    Trouble((1, "failed␣to␣read␣dump␣pointer␣file␣%s", fn_pointer));
  }
  fclose(fi);
}
```

This code is used in section 14.

14. Reading. When this function is called, the anatomy has already been absorbed and using that information, space for the status information should have been allocated. Strange things may happen when the anatomy of the writing and reading runs does not match, but we cannot verify everything. We just check if the crucial variables like *Nbelem* and *Ncells* agree; the user should do sensible things – or expect what may be expected.

```
⟨ implementation of load_status( ) 14 ⟩ ≡
  int load_status(char *dump_name, double *stime, float *t_ecoule, int required)
  {
    gzFile zr;
    int Nparts;    /* number of domains in dump */
    elem_t *df_begin_vertex, *df_end_vertex;    /* domain limits read from dump */
    elem_t *df_begin_node, *df_end_node;
    char fn_pointer[256], fn_master[256], fn_prefix[256];
    time_t tstart;
    master_info rinfo;

    tstart = time(Λ);
    logline("SLRP", "load_status␣$Revision:␣4.67␣$");
    snprintf(fn_pointer, 256, "%s/%s.ptr", dirname, dump_name);
    logline("SLRP", "␣␣pointer␣file␣:␣%s", fn_pointer);
    ⟨ read pointer file 13 ⟩    /* gets fn_prefix and l_ext */
    snprintf(fn_master, 256, "%s%s", fn_prefix, l_ext);
    logline("SLRP", "␣␣part␣prefix␣␣:␣%s", fn_prefix);
    logline("SLRP", "␣␣master␣file␣␣:␣%s", fn_master);
    ⟨ open master input file 16 ⟩
    ⟨ read header 18 ⟩
    ⟨ read master info 19 ⟩
    ⟨ close master input file 17 ⟩    /* reports status */
    logline("SLRP", "␣␣reading␣part␣files...");
    ⟨ read part files 20 ⟩
    logline("SLRP", "␣␣simulation␣continues␣at␣t=%.3f", rinfo.stime);
    *stime = rinfo.stime;    /* modify globals and args only if succesful */
    stade = rinfo.stade;
    compteur = rinfo.compteur;
    temps = rinfo.temps;
    *t_ecoule = rinfo.ecoule;
    Ndumps = rinfo.Ndumps;
    ve_valid = rinfo.ve_valid;
    logline("SLRP", "load_status␣took␣%.0f␣seconds", difftime(time(Λ), tstart));
    return 0;    /* happy */
  }
```

This code is used in section 2.

15. In the reader function, the *required* argument determines what to do in case of trouble: error exit or just a warning and a nonzero return. In order to allow passing an undetermined number of arguments to the *Error* and *Warning* macros, the *Trouble* macro takes double parentheses, as in *Trouble*((1, "boo")).

```
#define Trouble(args)
        {
          if (required) {
            Error args;
          }
          else {
            Warning args;
            return 1;    /* unhappy */
          }
        }
```

16. We use the `zlib` functions for reading rather than opening a pipe from `gunzip`, since we are using them anyway, and it may be useful to catch `SIGTERM` while reading, although it is not as crucial as it is for writing.

The *gzopen* function will see if the file is a gzip file or not. Mismatches in the *compress_status_files* parameter will thus be corrected. The program can be started with the *compress_status_files* parameter set to the desired value for the status output no matter what kind of status input is used.

⟨ open master input file 16 ⟩ ≡

```
{
  int c = 0;
  if (compress_status_files ≥ 0 ∧ compress_status_files < 10) {
    if (¬(zr = gzopen(fn_master, "rb")))
      Trouble((2, "Cannot␣open␣file␣<%s>:␣%s\n", fn_master, gzerror(zr, &c)));
  }
  else {
    Trouble((1, "unknown␣compression␣code"));
  }
}
```

This code is used in section 14.

17. ⟨ close master input file 17 ⟩ ≡

```
{
  int c;
  c = gzclose(zr);
  if (c ≠ Z_OK) Warning(1, "zlib␣error:␣%s", gzerror(zr, &c));
}
```

This code is used in section 14.

18. Read the header and check if the format version matches.

```
#define test_read(fz, dest, Nbytes, name)
        {
          int n, c = 0;
          n = gzread((fz), (dest), (Nbytes));
          if (n ≠ (Nbytes)) Trouble((1, "read␣from␣%s␣failed:␣%d␣of␣%d␣bytes␣for␣\"%s\";␣%s",
                fn_master, n, Nbytes, name, gzerror(fz, &c)));
        }
```

⟨ read header 18 ⟩ ≡

```
{
  char h[HEADER_CHARS];
  test_read(zr, h, HEADER_CHARS * sizeof(char), "header");
  if (strncmp(h, ID_STRING, strlen(ID_STRING)) ≠ 0)
    Trouble((1, "unrecognized␣format␣for␣%s", fn_master));
}
```

This code is used in section 14.

19. All information is read into local variables. Global variables of `propag` are only modified at the end of *load_status*, when we know that the entire load succeeded, so that we don't end up with an inconsistent status. In case of a failure to load the status dump, the main program will clean up the *Vmem* and *yyy* arrays if it decides to restart the simulation from scratch. The variables *Nverts* and *Nnodes* will never be overwritten: if they are not equal to their loaded counterparts, the anatomic setup doesn't match the status dump so the simulation cannot continue anyway.

```
⟨ read master info 19 ⟩ ≡
  {
    int i, c = 0;
    test_read(zr, &rinfo, sizeof(master_info), "master␣info");
    if (rinfo.Nverts ≠ Nverts)
      Trouble((1, "Mismatch:␣dumped␣Nverts=%d,␣current␣Nverts=%d;␣%s\n", rinfo.Nverts, Nverts,
          gzerror(zr, &c)));
    if (rinfo.Nnodes ≠ Nnodes)
      Trouble((1, "Mismatch:␣dumped␣Nnodes=%d,␣current␣Nnodes=%d;␣%s\n", rinfo.Nnodes, Nnodes,
          gzerror(zr, &c)));
    ⟨ compare domi 22 ⟩
    Nparts = rinfo.Nparts;
    logline("SLRP", "␣␣%d␣part(s)", Nparts);
    df_begin_vertex = MALLOC(Nparts, elem_t, "");
    df_end_vertex = MALLOC(Nparts, elem_t, "");
    df_begin_node = MALLOC(Nparts, elem_t, "");
    df_end_node = MALLOC(Nparts, elem_t, "");
    test_read(zr, df_begin_vertex, Nparts * sizeof(elem_t), "begin_vertex");
    test_read(zr, df_end_vertex, Nparts * sizeof(elem_t), "end_vertex");
    test_read(zr, df_begin_node, Nparts * sizeof(elem_t), "begin_node");
    test_read(zr, df_end_node, Nparts * sizeof(elem_t), "end_node");
#if 0
    for (i = 0; i < Nparts; i++) {
      logline("SLRP", "␣␣part␣%d␣:␣verts␣%d␣--␣%d␣␣nodes␣%d␣--␣%d", (int) i, df_begin_vertex[i],
          df_end_vertex[i] − 1, df_begin_node[i], df_end_node[i] − 1);
    }
#endif
  }
```

This code is used in section 14.

20. The number of parts is not necessarily equal to the number of threads in the reading process. Therefore we do this in a parallel loop rather than a parallel section. If the number of threads is the same as in the writing process, this is optimal. In other cases, it is at least much better than working in serial mode, provided that the arrays have been touched previously to have them allocated on the right processor board (on distributed-memory systems like the Altix). We use `schedule(static)` to preserve data locality as much as possible; this is more important than load balance.

The *yyy* array must be written in parts, because it can be larger than 2 GB and the size argument to *gzread*( ) is **int**.

```
⟨ read part files 20 ⟩ ≡
  {
    int part, trouble = 0;
#pragma omp parallel for schedule(static) private(part)
    for (part = 0; part < Nparts; part++) {
      gzFile zr;
      char fname[256];
      int c = 0, i, bite, trb;
      elem_t n0, Nn, v0, Nv;
      long nb;
      snprintf(fname, 256, "%s.%04d%s", fn_prefix, part, l_ext);
      if ((zr = gzopen(fname, "rb"))) {
        n0 = df_begin_node[part];
        Nn = df_end_node[part] − df_begin_node[part];
        v0 = df_begin_vertex[part];
        Nv = df_end_vertex[part] − df_begin_vertex[part];
        nb = Nn ∗ sizeof(float);
        if (gzread(zr, dtime + n0, nb) ≠ nb) trb = 2;
        nb = Nv ∗ sizeof(vm_t);
        if (gzread(zr, Vmem + v0, nb) ≠ nb) trb = 2;
        if (rinfo.domi > 1) ⟨ read or skip ve 21 ⟩
        if (mem_y) {     /∗ yyy not allocated in forward model ∗/
          bite = 1024 ∗ 1024;
          for (i = 0; i < Nn; i += bite) {     /∗ must break it for gzread( ) ∗/
            if (bite > Nn − i) bite = Nn − i;
            nb = bite ∗ nsvar ∗ sizeof(yyy_t);
            if (gzread(zr, &yyy(n0 + i, 0), nb) ≠ nb) trb = 2;
          }
        }
      }
      else {     /∗ if gzopen failed ∗/
        trb = 1;
      }
#pragma omp critical
      {
        if (trb ≡ 1) printf("\nCannot open file <%s>: %s\n", fname, gzerror(zr, &c));
        trouble = trb;     /∗ return and break not allowed in OpenMP loop ∗/
      }
#if ¬USE_OPENMP
      if (trouble) break;     /∗ if not parallel, stop in case of trouble ∗/
#endif
    }
    if (trouble ≡ 1) Trouble((2, "Cannot open part file"));
    if (trouble ≡ 2) Trouble((2, "Read failure in part file"));
  }
```

This code is used in section 14.

21. It is probably OK to run bidomain and read a monodomain status dump; this is a useful method to kickstart a bidomain simulation. Continuing a bidomain as a monodomain is less usual, but not necessarily wrong. All is well as long as we read or skip *ve* when it's present, and don't try to read it when it isn't there.

⟨ read or skip *ve* 21 ⟩ ≡

```
{
  if (domi > 1) {    /* if we want it, */
    if (gzread(zr, Vex + v0, nb) ≠ nb) trb = 2;    /* read it; */
  }
  else {
    if (gzseek(zr, nb, SEEK_CUR) ≡ −1) trb = 2;    /* else, skip it. */
  }
}
```

This code is used in section 20.

22. ⟨ compare *domi* 22 ⟩ ≡

```
{
  if (domi ≡ 1 ∧ rinfo.domi > 1) {
    Warning(29, "Status␣dump␣is␣bidomain,␣running␣monodomain.␣Ignored␣Ve.");
  }
  else if (domi > 1 ∧ rinfo.domi ≡ 1) {
    Warning(30, "Status␣dump␣is␣monodomain,␣running␣bidomain.");
  }
}
```

This code is used in section 19.

23. Bibliography.

[potse:isch] Mark Potse, Ruben Coronel, Stéphanie Falcao, A.-Robert LeBlanc, and Alain Vinet. The effect of lesion size and tissue remodeling on ST deviation in partial-thickness ischemia. *Heart Rhythm*, 4(2):200–206, 2007.

[potse:bidofex] Mark Potse, Bruno Dubé, Jacques Richer, Alain Vinet, and Ramesh M. Gulrajani. A comparison of monodomain and bidomain reaction-diffusion models for action potential propagation in the human heart. *IEEE Trans. Biomed. Eng.*, 53(12):2425–2435, 2006.

[trudel:prop] Marie-Claude Trudel, Bruno Dubé, Mark Potse, Ramesh M. Gulrajani, and L. Joshua Leon. Simulation of propagation in a membrane-based computer heart model with parallel processing. *IEEE Trans. Biomed. Eng.*, 51(8):1319–1329, 2004.

24. Index.

⟨ close master input file 17 ⟩ Used in section 14.
⟨ compare *domi* 22 ⟩ Used in section 19.
⟨ implementation of *load_status*( ) 14 ⟩ Used in section 2.
⟨ implementation of *save_status*( ) 4 ⟩ Used in section 2.
⟨ local declarations 3 ⟩ Used in section 2.
⟨ open master input file 16 ⟩ Used in section 14.
⟨ private functions 5, 6, 11 ⟩ Used in section 2.
⟨ read header 18 ⟩ Used in section 14.
⟨ read master info 19 ⟩ Used in section 14.
⟨ read or skip *ve* 21 ⟩ Used in section 20.
⟨ read part files 20 ⟩ Used in section 14.
⟨ read pointer file 13 ⟩ Used in section 14.
⟨ `status.c` 2 ⟩
⟨ `status.h` 1 ⟩
⟨ write header 8 ⟩ Used in section 4.
⟨ write master info 9 ⟩ Used in section 4.
⟨ write part files 10 ⟩ Used in section 4.
⟨ write pointer file; **return** if it fails 12 ⟩ Used in section 4.