Distributisation of an Implicit surface polygonizer

Developer Manual

End of Studies Project

Pierre-Alain Fayolle
Philippe Perrin

E.N.S.E.I.R.B.
Computer Science Department
3\textsuperscript{rd} year

June 2002
Contents

Foreword 3

1 Compiling the sources 4
  1.1 Overview ............................................. 4
  1.2 Building the distributed version ................... 4
  1.3 Compile with the HyperFun interpreter .............. 5
  1.4 Using the distributed polygonizer in another application ...........
     1.4.1 How to build an application with the distributed polygonizer 5
     1.4.2 How to use your own implicit surfaces ........... 6

2 General code structure 7
  2.1 Modular organization .................................. 7
     2.1.1 mainapp/main.o ................................... 7
     2.1.2 mainapp/mainapp.o ................................. 7
     2.1.3 mainapp/renderOut.o ............................... 8
     2.1.4 mainapp/evalNodes.o ............................... 8
     2.1.5 polyg/hfpm_variables.o ............................ 8
     2.1.6 polyg/hfpmesh.o .................................. 8
     2.1.7 polyg/hfpm_calc.o ................................. 8
     2.1.8 polyg/hfpm_cells.o ............................... 9
     2.1.9 polyg/hfpm_fillvertices.o ...................... 9
     2.1.10 polyg/hfpm_addface.o ......................... 9
     2.1.11 polyg/hfpm_rebalance.o ....................... 9
     2.1.12 polyg/hfpm_createfaces.o ................... 9
     2.1.13 polyg/hfpm_createedges.o ..................... 10
     2.1.14 polyg/hfpm_multicast.o ....................... 10
     2.1.15 polyg/hfpm_FinalizeDataAlone.o ............ 10
     2.1.16 polyg/hfpm_distribTime.o .................. 10
     2.1.17 polyg/hftimer.o ................................ 10
     2.1.18 polyg/hfpm_fridata.o ......................... 10
     2.1.19 comm.comm.o .................................. 11
     2.1.20 comm/error.o .................................. 11
  2.2 Naming convention .................................... 11
     2.2.1 In the polygonization module .................. 11
     2.2.2 In the communication module ................... 12
  2.3 Using another communication library ................ 12
     2.3.1 comm/comm.c .................................. 13
     2.3.2 comm/error.c .................................. 13
     2.3.3 mainapp/evalNodes.c ............................ 13
3 Error handling

3.1 Error handling in our polygonizer

3.1.1 Errors from the master

3.1.2 Errors from slaves

3.1.3 Segmentation faults

3.1.4 Timeouts

3.1.5 Spawning slaves

3.1.6 Conclusion

3.2 Error handling under the Microsoft Windows operating system

Bibliography

A Function call trees

B Functions and variables used in communication

B.1 Global variables

B.1.1 Global variables defined in comm.c

B.2 External functions

B.2.1 External functions defined in comm.c

B.2.2 External functions defined in error.c

B.3 Internal functions

B.3.1 Internal functions defined in comm.c

B.3.2 Internal functions defined in error.c

C Functions and variables used in polygonization

C.1 Global variables used by the polygonization module

C.2 External functions

C.2.1 Functions for initiating and ending polygonization

C.2.2 Functions for setting / getting the polygonizer’s behavior

C.2.3 The main function

C.2.4 Functions for retrieving the results of polygonization

C.3 Internal functions

C.3.1 Functions dealing with cells

C.3.2 Functions dealing with triangles

C.3.3 Various functions
Foreword

This document aims at helping developers to understand our implementation of a distributed polygonizer. We designed it as a module for the HyperFun project\textsuperscript{1}, and wrote a report\textsuperscript{2} [4] describing the whole algorithm in details. We advise you to read this document before going on with this one, at least its chapters 1, 2 and 5 (about HyperFun, PVM, and our algorithm).

This manual recalls some informations given in the report (in its 6\textsuperscript{th} chapter about implementation), and the annexes list the functions we designed (or modified) for distributing the original polygonizer (including a tree drawing of the function calls, in appendix A).

Please note that our work was to actually modify an existing module. Therefore, we will not detail here the files or functions that we did not alter. You will only read about the parts we designed or adapted ourselves.

\textsuperscript{1}http://www.hyperfun.org/
\textsuperscript{2}as part of our End of Studies Project in the ENSEIRB computer school
Chapter 1

Compiling the sources

1.1 Overview

The distributed polygonizer is provided as a library that can be used for rendering implicit surfaces.

In order to use the polygonizer, we added some extra code to parse the command line, to deal with options, and generate output. Actually, we propose the polygonizer as a library within the HyperFun package, but it can also be used within your own application for rendering implicit surfaces.

This chapter firstly explains how to compile and build the polygonizer within the HyperFun project. The first section explains that you can build two versions: a distributed and a non-distributed one. The second section explains how to compile the polygonizer within the HyperFun project, and what is actually done. Finally the last section explains how to build the distributed polygonizer within your own project (e.g. without using the HyperFun project and/or without using our main application); it shows especially which files need to be modified and what you should do in your application.

1.2 Building the distributed version

When building the polygonizer module, you can choose to compile it as a distributed or a single-CPU version. This is done by using the appropriate Makefile:

- Makefile: it is the Makefile for the single-CPU application,
- Makefile.pvm: it is the Makefile for the multi-CPU application.

If you are using the distributed polygonizer within the HyperFun package, it is quite easy to use the appropriate Makefiles for all the modules. The principal Makefile, located in the root directory, contains a line that should be kept commented for compiling all modules with mono-CPU support, or that should be uncommented to support multi-cpu (example below):

```plaintext
###
# For PVM support: uncomment the line below
USE_PVM=.pvm
###
```
1.3 Compile with the HyperFun interpreter

The current version of the distributed polygonizer is provided by default in the HyperFun package as a means to render the implicit surface described in the HyperFun language and interpreted by the HyperFun interpreter.

Compiling the polygonizer within the HyperFun package is a priori an easy task. At the root of the package, there is a Makefile that will “call” all the other ones (in the subdirectories). You need first to modify this Makefile, depending whether you want to use the distributed version, or the non-distributed version, as it is explained in the previous section above.

We will now study in details the order in which the other Makefiles are called, and to what part of the application they correspond.

- The first Makefile to be called is the one in mem_manager. In this directory stands a code that was used only during development and debug. This code overloads malloc, calloc, realloc and free in order to log information on the memory use during the execution of the program.

- The second Makefile called is in hlibrary_lib, which contains the core library of HyperFun.

- Then it goes into interpreter_lib and compiles the interpreter which is a shared object library. The interpreter is later used in the main application, especially to instantiate the polygonizer.

- Then it enters polyg and calls the Makefile for the polygonizer library.

- Finally it goes into mainapp to compile the main application. This compiles also some code located in the comm directory.

1.4 Using the distributed polygonizer in another application

You may wish to use the polygonizer inside your own application, or with your own implicit surfaces (in the case you don’t want to use the interpreter of HyperFun).

1.4.1 How to build an application with the distributed polygonizer

We will treat the case here where you want to use the polygonizer in your own application. You need the directories polyg and comm for the polygonizer, mem_manager and hlibrary_lib along with interpreter_lib (we suppose that you want to keep the interpreter).

The polygonizer uses some structures defined in mainapp, such as MAINAPP, mainly in the comm directory, especially in the file comm.c. The main application module has some mandatory task to achieve. Firstly, it must provide a procedure for evaluating the nodes of the grid. We provide one as an example in mainapp/evalNodes.c. Then the main application module needs to spawn slaves, to distinguish whether in master- or slave-mode, …
1.4.2 How to use your own implicit surfaces

If you want to use your own implicit surfaces (inputed as a C function) rather than the interpreter provided with HyperFun, we advice you to use the main application module (*mainapp/mainapp.c*) and to adapt it:

- **MAINAPPINFO *cmainapp()**: you don’t need to instantiate an interpreter any more in this function;
- **void miReadFile(MAINAPPINFO * mainapp)**: it can be removed;
- **void miParseString(MAINAPPINFO *mainapp)**: it can be deleted as well.

Finally, you have to modify **void miGenerateTriangles(MAINAPPINFO *mainapp,**
**DISTRIBINFO *distribInfo)**, especially the instantiation of the polygonizer (function *HFPM_INIT_HFPolyMesh*): the first argument will be a pointer to your function, and the second argument will be a NULL pointer, since there is no interpreter left.
Chapter 2

General code structure

2.1 Modular organization

2.1.1 mainapp/main.o

This module contains the main() function. It is responsible for calling the functions used to parse the command line, and therefore to detect if the user asked multi-CPU mode (with the -m multiCPUconf switch). Actually, the program can be in one of the 3 following cases:

- It should execute in mono-CPU mode. It means the -m switch was not in the command line, or that you didn’t compile the program with PVM support. In this mode, the process does all the work alone.

- It should execute in multi-CPU mode, and it was called explicitly by the user (who typed the command line). This process will be the master and will spawn the slaves mentioned in the configuration file (mentioned with the -m switch), and these slaves will be in the third case.

- It should execute in multi-CPU mode, and it is a slave process. It means that the process was spawned remotely by a master (see previous case).

The distinction between these 3 cases can be made by analyzing the command line, and every process knows exactly in which one it is; this is determined in the main.o module.

A master process will parse its command line and the configuration file, then will spawn the slaves (using the comm/comm.o module, see 2.1.19), and broadcast some information to them (the scene, the model, the options...). Then all processes (master and slaves) go on with module mainapp/mainapp.o (2.1.2 below).

2.1.2 mainapp/mainapp.o

In this module, all processes set the polygonization options (which the master has read and sent to its slaves) by calling the appropriate methods of the polyg/hfpolymesh.o module (2.1.6). Then they start the polygonization itself, by using the polyg/hfpm_calc.o module (2.1.7).
2.1.3  mainapp/renderOut.o

This is another module we made in mainapp: it writes the output .dat file in a custom format. Called at the end of polygonization, it writes the output in our custom format, which may be read by our custom renderers (please refer to the appendix G dealing with the renderers in our report [4]).

However, if the -wrl switch was provided in the command line, then this module is not used. Instead, the program will write the output in VRML 2.0 format, using the mainapp/vrmlout.o module (which was the default output in the original version).

2.1.4  mainapp/evalNodes.o

This last module of mainapp is used for the weight computing mode, when the -test switch was provided. The weights are used to balance the amount of work among the workers considering their computational power and their load level (for example, there could be other applications running and consuming lots of CPU resources). For more information on the weights, please refer to the report [4], in the chapter dealing with implementation (chapter 6), there is a paragraph on the goal and the use of weights (section 6.3 of [4]). It is called once the slaves have been spawned and the F-Rep object has been broadcasted. Every process starts computing values of this function during the same amount of time, then the master collects the results from the slaves. Lastly, this module overwrites the configuration file with a new one, containing the new weights.

When the program has entered this module, it is supposed to exit afterwards: there is no polygonization after the weights were computed.

2.1.5  polyg/hfpm_variables.o

This first module of polyg contain the declarations of global variables (mostly arrays). It does not implement any function. Most of the variables declared here also appear in a header file, private.h, included at the beginning of almost all our files (with the extern keyword in front of the variable declarations).

In this files, we put only variables which must be shared across different modules. Those used by only one are declared static inside it.

2.1.6  polyg/hfpolymesh.o

This module contains some the functions that should be called by modules outside the polygonizer (the polyg source directory). It contains two types of functions: those for setting options, and those for getting the results. The mainapp/mainapp.o module (2.1.2) starts by the setting functions, then starts the polygonization, and lastly the output modules use the getting functions. See also the polyg/hfpm_tridata.o module (2.1.18) for the other getting functions.

2.1.7  polyg/hfpm_calc.o

This module is used to really polygonize the model. It starts by initializing some structures, then calls the modules corresponding to the different steps of the distributed algorithm described in chapter 5 of the report [4], and lastly frees the temporary structures. Depending on the compiling mode (with or without
PVM) and the execution mode (mono-CPU, multi-CPU master or slave), this file will call the appropriate other modules.

2.1.8 \texttt{polyg/hfpm\_cells.o}

This module deals with the cells owned by a process. As stated in the report [4] in chapter 5 (on the distributed algorithm), workers are said to “own” cells: it means they are the only one to search them for vertices or triangles inside them. A first function of this file is called by \texttt{polyg/hfpm\_calc.o} (2.1.7) to build and store the owned cell list: it will distinguish between the 3 cases mentioned above (alone, master, slave). Then, the modules needing to go through all the owned cells will simply call a cell iterator, until all cells are visited.

In the algorithm detailed in [4], there is a step where cells are transferred among workers, in order to re balance the approximate number of triangles owned by each of them. This module contains all the auxiliary functions needed for this, which are called by the module \texttt{polyg/hfpm\_rebalance.o} (2.1.11). There are functions for storing the approximate number of triangles of each cell, deleting cells once they were transferred to another worker, and adding new cells once they were received.

2.1.9 \texttt{polyg/hfpm\_fillvertices.o}

This module is then called by \texttt{polyg/hfpm\_calc.o} (2.1.7) in order to achieve the vertex search among the owned cells. By using the iterator of \texttt{polyg/hfpm\_cells.o} (2.1.8), it does not need to distinguish between the cases where the process is alone or a master or a slave. It goes through all cells, detects vertices inside them, and computes an approximate number of triangles inside them.

2.1.10 \texttt{polyg/hfpm\_addface.o}

This auxiliary module is called by \texttt{polyg/hfpm\_fillvertices.o} (2.1.9) when a vertex is detected in a cell, and by \texttt{polyg/hfpm\_cells.o} (2.1.8) when a worker receives new cells and has to own them. It is used to build the connectivity graph of all owned vertices, as explained in [1].

2.1.11 \texttt{polyg/hfpm\_rebalance.o}

This module is used to redistribute cells among workers. Once the vertices are located and counted, the slaves send their total numbers of triangles to the master (they are estimations), who creates a list of cells to move from “sending workers” to “receiving workers”. Then all slaves receive this list, and if they are to send or receive cells, they call the appropriate functions in \texttt{polyg/hfpm\_cells.o} (2.1.8) in order to delete them (sending case) or to add them (receiving case) to the owned cell list. This new list will be used for the next step: finding triangles.

2.1.12 \texttt{polyg/hfpm\_createtriangles.o}

This is the module for going through the connectivity graph and group the vertices in triangles. It contains the main procedure to achieve this task, as well as some auxiliary static functions. There is also another function to do some
conversions between edge and vertex indices, and also one to revert triangles if their normals are not orientated according to the function defining the F-Rep object (both called by \texttt{polyg/hfpm}\_calc.o, 2.1.7).

2.1.13 \texttt{polyg/hfpm\_createnormals.o}

This module is used to go through the vertex list (built by \texttt{polyg/hfpm\_fillvertices.o} in 2.1.9), and compute the normal of the function at these locations.

2.1.14 \texttt{polyg/hfpm\_multicast.o}

This module deals with the global numbering of vertices and vertex gathering; these steps are detailed in the report [4]: this is where the vertex number tables are broadcasted between all workers, and the master receives copies of vertices and triangles. This module makes initialization work and calls communication functions from \texttt{comm/comm.o} (2.1.19) depending on the process successive states (master or slave, sending or receiving information, . . . ).

2.1.15 \texttt{polyg/hfpm\_FinalizeDataAlone.o}

This is a module used in mono-CPU mode, instead of \texttt{polyg/hfpm\_multicast.o} (2.1.14). Both of these modules are called by \texttt{polyg/hfpm\_calc.o} (2.1.7), once temporary structures are filled (and distributed, in the case of \texttt{hfpm\_multicast.o}), and both are supposed to finalize the output data (vertices, triangles and possibly normals) in other arrays. This module does the same work as \texttt{hfpm\_multicast.o}, except that it does not call communication functions.

2.1.16 \texttt{polyg/hfpm\_distribTime.o}

All the main polygonization steps were initiated by \texttt{polyg/hfpm\_calc.o} (in 2.1.7), where the times spent were stored in a timing structure, locally to all workers (see \texttt{polyg/hftime.o}, 2.1.17 below). At the end of polygonization, all slaves send their structures to the master, which uses \texttt{polyg/hfpm\_distribTime.o} to display them. This module goes through the timing structures of all workers, displays all of the steps for all workers in an array, and makes some statistics on them. This module is not called if the -t switch did not appear in the command line.

2.1.17 \texttt{polyg/hftime.o}

This is the module for starting and ending the timing of a polygonization step (see \texttt{polyg/hfpm\_distribTime.o}, in 2.1.16 above). It is used to get the time spent, and to display it (with a message) on the standard output.

2.1.18 \texttt{polyg/hfpm\_tridata.o}

This module contains some initialization and destruction procedures for arrays private to the polygonizer. It also contains some functions to read the final numerical data (vertex and normal coordinates, triangles).
2.1.19  *comm/comm.o*

This is the first module used as an interface between the polygonizer and the PVM library. There are no PVM functions called outside the modules in *comm* (except the particular case of *evalNodes.c* where the slaves send some integers to the master). This one is called at the initialization of the algorithm, it is used to spawn the slaves, for sending information between workers, and for detecting timeouts while waiting for informations (master only). The functions of this file are usually grouped in pairs, where one is used to send information and the other to receive it. The procedures are more than only PVM wrappers: they can also do some work on the received structures, or make some loops for treating a series of individual steps (like a master waiting for information from every slave, for example).

2.1.20  *comm/error.o*

This is the second module used as an interface between the polygonizer and the PVM library. It is dedicated to error handling, which is explained in 3.1. All the error functions are located in this module, which is quite oriented for UNIX systems.

2.2  **Naming convention**

2.2.1  **In the polygonization module**

**Naming convention for files**

The files in the polygonization module (in *polyg*) are preceded by “hfpm_”, then the rest of the name explains shortly what is inside the file. For example, *hfpm_rebalance.c* is a file of the polygonizer that contains code on the balancing of triangle between the workers.

Some files do not respect this naming convention:

- *hfpolymesh.c* and *hfpolymesh.h*: they come from the original names of the C++ version of the polygonizer, and we did not want to modify it.

- *hftimer.c* and *hftimer.h*: these files provide code for data mining, time measurement; it also comes from the original polygonizer in C++.

- *hFtPolyTypes.h*: this file contains our dedicated data structures, as well as macros, for manipulating the data structures. We designed it while converting the C++ code (especially the calls to the STL) into ANSI C. It is not really part of the polygonizer but provides material for it.

- *private.h*: it contains the declarations of the functions and variables private to the polygonizer module.

**Naming convention for the functions**

Most of the functions begin with “HFPM” prefix in upper case, then the names of the functions explain their aim shortly. For example, the function that calculates the vertices of the polygonized model is called: *HFPM_Calc*. 
There remains some functions that are prefixed by “TD” instead of “HFPM” within the polygonization module. All these functions are located in hfpm_tridata.c. There is a historical reason for that naming. In the original C++ version of the polygonizer, the whole code of the polygonizer was within a class hfpolymesh, and inside that class was an inner class TriData. An object TriData was responsible for handling the information on the triangles of the polygonized model. It provided methods for creating the triangles from the vertices, retrieving information on the triangles, vertices, ... In order to respect the idea of the original code we did not modify it.

**Naming convention for the variables**

All the variable are named with “its” followed by the name of the variable. For example itsTimer is a variable inside the polygonizer module that keeps timing information.

The variables corresponding to the special data types that hold information on the polygonized model have some specific names that should be also explained here. We give the example of “itsVTriangles” to explain it here. ItsVTriangles means that it holds information for the Vertices constituting each Triangle. Let’s consider another example: if you want to use the variable that holds information on the two triangles sharing an edge, you will use the variable named “itsTEdges”.

### 2.2.2 In the communication module

**Naming convention for the functions**

The functions used for communication begin with “comm”, then names are appended to explain their goals. They are directly appended without any “.” like in “commIsMaster”.

Usually, functions dedicated to the master’s use only start with the prefix commMaster, whereas the functions dedicated to the slaves start with the prefix commSlave.

Separation between two words is marked by putting the first letter of the next word in upper case.

**Naming convention for the variables**

Names of variables give indication on what they are used for. Names are composed of words; when the name has more than one word the next ones begin with a letter in upper case, like the naming convention applied to functions.

### 2.3 Using another communication library

For making our machines communicate between each others, we chose to use the PVM library. However, you may want to use another one, such as MPI. This section aims at explaining which files should be modified in that case. This should be an easy task, since we grouped all “low-level” communications in 2 or 3 files, thus designing wrappers used in all our functions (those in comm*).
2.3.1   *comm/comm.c*

Almost all functions in this file use the PVM library. In order to use another one, you should replace all the `pvm_*()` calls by their new equivalents. Note that some global variable would also be removed or adapted, like those dealing with the TIDs.

2.3.2   *comm/error.c*

This file contains all functions for our error handling procedures. Most of them deal with the POSIX signal API; however, the PVM library is used to transmit the slave errors to the master. For changing the communication module, you should modify some functions here, as well as take care of the global variables shared with `comm.c`.

- **void commSlaveReportsError(char *error):**  
  This function uses the PVM library for sending an alert to the master, along with the `error` character string.

- **void commMasterRecvSIGUSR1(int sig):**  
  This function is triggered on the master's side when the function above is used. It is the local `pvmd` daemon that sends a SIGUSR1 signal to the process, and this function uses the PVM library for retrieving the explanation character string.

- **void commMasterRecvSIGTERM(int sig):**  
  This function is triggered when the master has to cancel the whole polygonization procedure. It uses the PVM library to kill the slave processes.

2.3.3   *mainapp/evalNodes.c*

This is the module used for computing actual weights among the workers. In the function `testNode()`, the slaves send some integers to the master: this is the only use of the PVM library in this file.
Chapter 3

Error handling

3.1 Error handling in our polygonizer

Apart from segmentation faults (see in 3.1.3), the errors handled here are errors that may be caused by the network while communicating. Since the polygonization itself is supposed to be error-free, we only made error checking on sending or receiving functions using the communication module `comm/comm.o`.

You will see that our error handling procedures use many signals, for both the master and slaves. To achieve this, we made some functions to set signal handlers, and some to remove them. They are called at the beginning and end of polygonization.

Since this kind of error denotes a network error, that we consider unrecoverable, we simply abort the whole polygonization. We did not try to do some recovery on errors, or some work redistribution. Our program is designed for working on a fast, possibly commuted, and reliable network.

All functions concerning error handling display the events on the master terminal. By reading this standard output, it is easy to know what error occurred, with which worker, and how the error was handled (usually an abortion).

3.1.1 Errors from the master

When the master detects an error (a PVM function returns a negative value, or received structures do not match some size criteria), it sends itself a SIGTERM. This signal triggers the general aborting procedure: the master kills all spawned slaves, and exits.

What is more, if the master receives a SIGINT (case of a Ctrl-C), it also sends itself a SIGTERM, so that the slaves are not left without a master. Note that a SIGTERM can also be sent with the `kill` command, with the default signal.

3.1.2 Errors from slaves

If a slave detects a communication error (on the same criteria as the master), it sends a SIGUSR1 to the master (with PVM mechanisms), as well as an explanation message. The master gets interrupted, receives the signal and the message, and displays it (with the slave name), and sends itself a SIGTERM,
in order to abort polygonization. In this way, reading the master’s output
 guarantees to be alarmed in case any worker encounters an error, to know which
 one, and what happened.

 If a slave dies “brutally” (with a SIGKILL, for example), it doesn’t have the
 opportunity to notify the master: one of the worker is now dead, and no one
 knows it. This will be detected the next time that the master will be waiting
 for this slave: see the timeout handling in 3.1.4.

 3.1.3 Segmentation faults

 All workers (master and slaves) are able to catch segmentation faults and bus
 errors (these should never happen). If the master meets one, it sends itself a
 SIGTERM; if a slave meets one, it uses the usual error reporting function, thus
 sending a SIGUSR1 to the master.

 3.1.4 Timeouts

 Lastly, note that the master uses some timeout when waiting for slaves. If one of
 the waiting functions lasts more than the compiled timeout (which may mean
 that a slave is down or unreachable... or simply late), it triggers a checking
 procedure:

 • If the master was waiting for one particular slave (case where it receives
   information for slaves in the sequential order \( P_1 \ldots P_n \)), it uses PVM to
detect if it is still alive (and therefore late) or if it is dead (or unreachable).
   In the first case, it returns to its waiting state (until another timeout is
   over); in the second case, it considers the slave lost, and sends itself a
   SIGTERM for aborting polygonization.

 • If the master was waiting for any slave (case where it receives information
   from all slaves in any order, until all have responded), it uses PVM to
detect if one of the late ones is dead (the master keeps a list of the slaves
   that have successfully sent their data). It reacts like the case detailed
   above if all slaves are still alive, or if one is declared unreachable.

 3.1.5 Spawning slaves

 When the master is initially spawning its slaves, it goes through the list provided
 in the configuration file. Remember that “spawning a slave process” on a host
 requires that pvmd be running on it, and be attached to the same virtual machine
 as the master. When it attempts to spawn a polygonizer process on a remote
 host, there are different cases:

 • the remote host is reachable, and pvmd is already running: spawning the
   slave process is successful immediately.

 • the remote host is reachable, but pvmd is not running: the polygonizer
   attempts to add it to the virtual machine, and retries to spawn a slave
   process.

 • the remote host is unreachable.
In the first case, the spawning procedure is completed for the current slave, and the master goes on with the next one. In the second case, the slave host was not detected as a member of the virtual machine. The master uses the PVM functions to add it: if it is not successful, then the whole spawning process aborts (for all slaves). Usually, this operation fails when some temporary lock files were not deleted after the last pvmd was spawned (for example, if the computer was powered off instantly, or the pvmd process received a SIGKILL). The files to remove are located in /tmp and their names end with the user’s UID. Here is an example of messages that can be displayed during a spawning procedure:

```
poisson-hyperfun/: ./hfp core.hf -t -n -m multiCPUconf -test 5
numSlaves = 11
slave exec file = /usr/bin/hyperfun/hfp
verbose file on slaves = <none>
master's weight = 82341
Spawning balance: OK (weight = 80945)
Spawning sagitaire: OK (weight = 81587)
Spawning belier: Spawning remote pvmd: OK (weight = 81222)
Spawning gemeau: OK (weight = 81353)
Spawning vierge: Spawning remote pvmd: Error
    (try to run "rsh vierge rm /tmp/pvm*.5396")
Spawning error
  ===> Sending SIGTERM
  ===> Received SIGTERM
  ===> killing slaves...
  Killing slave 1: OK
  Killing slave 2: OK
  Killing slave 3: OK
  Killing slave 4: OK
Exiting
```

In this output, the master has counted 11 slaves to spawn (in the configuration file multiCPUconf), and starts spawning them. The hosts balance, sagitaire and gemeau caused no problem: pvmd was already running, and spawning the slave processes succeeded. The host belier did not belong to the virtual machine, so the polygonizer successfully added it (by spawning pvmd) before creating the slave process. Finally, there was an error on vierge. pvmd was not running on it, and it was impossible to spawn it. The polygonizer suggests a solution: running rsh to remove the temporary lock files. The current user’s UID is 5396, this is why it appears in the command. You can note that when spawning a slave fails, the ones that had successfully been spawned before are killed, so that they do not remain on the remote hosts.

### 3.1.6 Conclusion

All these error handling procedures have one principal aim: to avoid slaves going on with computations when an error was detected somewhere on the virtual machine. As one whole distributed polygonizer, if one part fails, the whole algorithm is stopped everywhere.
3.2 Error handling under the Microsoft Windows operating system

This distributed polygonizer has been developed and tested under Solaris, which is a UNIX system. It could prove useful to port it for the Windows operating systems.

The code for error reporting in `error.c` should be carefully watched, since it used signal handling, which is more or less specific to operating systems. More information on signal can be found on the MSDN site\(^1\) or within the Microsoft software development kit. For example, SIGINT, which we use for aborting the process, is not handled in the same way on Windows systems: a new thread is created as a triggered treatment. This can cause a single-threaded application, such as UNIX, to become multi-threaded, resulting in unexpected behavior.

\(^1\)http://msdn.microsoft.com/
Bibliography

[1] A.A. Pasko, V.V. Pilyugin 1988 *Geometric Modeling in the Analysis of Trivariate Functions*


Appendices
Appendix A

Function call trees

Figure A.1 draws a global view of the function calls (the box represents the error handling function, called in case any `comm*` function returns an error code). Figure A.2 (page 21) draws the details of the `HFPM_Calc` function.

![Function call tree diagram]

Figure A.1: First function call tree (main → `HFPM_Calc`)
Figure A.2: Second function call tree HFPM_Calc
Appendix B

Functions and variables used in communication

B.1 Global variables

B.1.1 Global variables defined in \textit{comm.c}

The following are the global variables used by the module dedicated to communication and error, these variables are defined in \textit{comm.c} and declared in \textit{commPrivate.h} (for internal use). They are only accessed inside the module of communication and error (in \textit{comm.c} and \textit{error.c} only) and should not be exported or used outside of these files.

- \texttt{int myTid}: the current worker’s TID
- \texttt{int parentTid}: the master TID
- \texttt{int *workerTids}: an array containing the TIDs of all workers
- \texttt{char **slaveNames}: an array containing the DNS names of all workers (valid for the master only)
- \texttt{char *execFile}: the path to the executable file used to spawn slaves (valid for the master only)
- \texttt{int numSlaves}: the number of slaves (number of workers - 1)
- \texttt{int myIcpu}: the identifier of current worker (from 0 to numSlaves)
- \texttt{unsigned int timeSpentInRecv}: used to measure the time spent in network reception procedures.

B.2 External functions

B.2.1 External functions defined in \textit{comm.c}

The following functions constitute the API for the communication module. These functions are declared in \textit{comm/comm.h}, defined in \textit{comm/comm.c} and
are exported to the other modules that require communication between processors; namely these modules are: \textit{polyg} and \textit{mainapp}.

These functions of communication are all wrapping function of the PVM API and add specific treatments dedicated to particular tasks, for example sending or receiving coordinates,…

\textbf{Functions for initialization and termination}

- \texttt{bool commInit():} Function used to initialize the PVM module. Returns \texttt{false} if any errors occur during initialization. Initializes the global values: \texttt{myTid} and \texttt{parentTid}.
- \texttt{void commEnd():} Function used to free the global variable \texttt{workerTids} (an array of integers) and to call the exit function of the PVM API (\texttt{pvm\_exit()}).

\textbf{Function for spawning slaves}

- \texttt{bool commMasterSpawnSlave(char **slaves, int iCPU, char *exec, char *argv1, DISTRIBUTINFO *distribInfo):}
  - \texttt{slaves:} array of the DNS names of all slaves
  - \texttt{iCPU:} index of the currently spawned slave in this array
  - \texttt{exec:} path to the executable file used to spawn the slave
  - \texttt{argv1:} a dedicated string to “tell” the spawned process it is a slave
  - \texttt{distribinfo:} a structure containing weight information to send to the spawned slave

  Function used by the master to spawn a given slave (corresponding to the slave number \texttt{iCPU} in the array \texttt{slaves}). This function initializes and alters the global values \texttt{workerTids}, \texttt{numSlaves}, \texttt{slaveNames}, \texttt{execFile}, and \texttt{myIcpu}. Returns \texttt{false} if an error occurs.

\textbf{Functions for sending / receiving configuration information}

- \texttt{bool commMasterSendMulticastInfo(MAINAPPINFO *mainapp):}
  - \texttt{mainapp:} used to retrieve some scene information (given in command line by the user).

  Function used by the master to broadcast:
  - the TIDs of all workers
  - the \texttt{testDuration} integer (used by the test procedure)

  Returns \texttt{false} if an error occurs.

- \texttt{bool commSlaveRecvMulticastInfo(MAINAPPINFO *mainapp):}
  - \texttt{mainapp:} used to retrieve some scene information (given in command line by the user).

  Function used by the slaves to retrieve the information sent above. Returns \texttt{false} if an error occurs.

- \texttt{bool commMasterSendHFfile(char *fileContent):}
  - \texttt{fileContent:} a buffer containing the HyperFun script.

  Function used by the master to send to all slaves a buffer containing the HyperFun script. Returns \texttt{false} if an error occurs.
• char *commSlaveRecvHFfile():
  Function used by a slave to retrieve the buffer containing the HyperFun
  script. Returns a pointer to the buffer containing the HyperFun script.

• bool commMasterSendSceneInfo(MAINAPPINFO *mainapp):
  mainapp: used to retrieve some scene information (given in command line
  by the user).
  Function used by the master to send to the slaves some information about
  the scene (contained in *mainapp). Returns false if an error occurs.

• bool commSlaveRecvSceneInfo(MAINAPPINFO *mainapp):
  mainapp: used to retrieve some scene information (given in command line
  by the user).
  Function used by the slaves to receive the information of the scene. Stores
  it in the structure *mainapp. Returns false if an error occurs.

Functions for communicating information on triangles

• bool commSlaveSendsNumTrianglesApprox(unsigned int totalNumTrianglesApprox):
  totalNumTrianglesApprox: rough number of triangles in the scene.
  Function used by the slaves to communicate to the master the approxi-
  mative number of triangles in their own cells. Returns false if an error
  occurs.

• bool commMasterRecvNumTrianglesApprox(uintPairArray *trianglesApproxWorkers,
  unsigned int totalNumTrianglesApprox):
  trianglesApproxWorkers: an array where each cell contains two infor-
  mations: the index of the worker, the approximative number of triangles
  in its cells (this is an output parameter).
  totalNumTrianglesApprox: an approximation of the number of triangles
  in the cells of the master.
  Function used by the master to collect (in any order) the number of tri-
  angles sent by the slaves. Returns false if an error occurs.

• bool commMasterRecvTriangles(intTripleArray *itsVTriangles):
  itsVTriangles: array that will hold the triangles (output parameter)
  Function used by the master to receive and store the triangles. Returns
  false if an error occurs.

• bool commSlaveSendTriangles(intTripleArray *itsVTriangles):
  itsVTriangles: array that contains the triangles that will be sent to the
  master.
  Function used by every slave to send its triangles. Returns false if an error
  occurs.

Communication of coordinates and indices

• bool commSlaveSendsCoordsToMaster(EVNarray *VNCoordinates, uintArray
  *VertexIndices, bool useNormals, unsigned int numTriangles):
  VNCoordinates: array of vertex [and normal] coordinates
  VertexIndices: list of indices for going through VNCoordinates
  useNormals: a boolean telling if normals are to be sent or not
numTriangles: the number of computed triangles
Function used by the slaves to send their coordinates to the master. Returns false if an error occurs.

- bool commMasterRecvCoordsFromSlave(VNarray *OUTVertexNormals, bool useNormals, int iCPUslave, unsigned int *OUTtotalTriangles):
  OUTVertexNormals: array where the master store vertices [and normals] (output parameter)
  useNormals: a boolean telling if normals are to be sent or not
  iCPUslave: index of the processor from which the master is receiving
  OUTtotalTriangles: integer for storing the total number of triangles (output parameter)
  Function used by the master to receive all the computed vertices (of the triangles) and eventually the normals at these vertices. Returns false if an error occurs.

- bool commMultiSendIndices(EVNarray *VNCoordinates, uintArray *VertexIndices, uintArray *OUTreceivedIndices):
  VNCoordinates: array containing the unsorted global edge numbers
  VertexIndices: array of indices to read the global edge numbers in a sorted order
  OUTreceivedIndices: array receiving the sorted global edge number (output parameter)
  Function used by every worker (master or slaves) to broadcast the array of sorted global edge numbers. Returns false if an error occurs.

- bool commMultiRecvIndices(uintArray *OUTreceivedIndices, int iCPUsender):
  OUTreceivedIndices: array for storing the received indices (global edge numbers) (output parameter)
  iCPUsender: index of the processor that sent this array
  Function used by the workers to receive the global edge numbers broadcasted by the other workers. Returns false if an error occurs.

Communication of cell moves between workers (exchange of triangles)

- bool commMasterSendMoves(uintTripleArray *listOfMoves):
  listOfMoves: an array of triplets, containing: the index of the sender, the index of the receiver, the number of triangles to exchange.
  Function used by the master to send the moves in multicast (to all slaves). A move is a triplet (index of sender, index of receiver, number of triangle to exchange). Returns false if an error occurs.

- bool commSlaveRecvMoves(uintTripleArray *listOfMoves):
  listOfMoves: an array of triplets, containing: the index of the sender, the index of the receiver, the number of triangles to exchange.
  Function used by each slave to receive the list of moves. Returns false if an error occurs.

- bool commWorkerTreatMoves(uintTripleArray *listOfMoves):
  listOfMoves: an array of triplets, containing: the index of the sender, the index of the receiver, the number of triangles to exchange.
  Function used by every worker to know what moves it has to treat. Every
worker goes through the list of moves. If the current worker takes part in a move, it sends or receives a given number of cells that hold the demanded number of triangles. Returns \textit{false} if an error occurs.

**Communicate time information**

- \textbf{bool commSlaveSendFinalInfos(runInfo *infos):}
  
  \textit{infos:} structure containing information on the time spent in the different sub-processes of the algorithm

  Function used by every slave to send back their local information about the time spent in the different operations (compute vertices, triangles, \ldots). Returns \textit{false} if an error occurs.

- \textbf{runInfo *commMasterRecvFinalInfos(runInfo *infos):}
  
  \textit{infos:} structure containing informations on the time spent by the master in the different sub-process

  Function used by the master to gather all informations about the time spent in the different operations, by all workers. The input \textit{*infos} corresponds to the timing information of the master. Returns a pointer to an array of \textit{runInfo} structures. This array contains the time informations for all the processors. Returns \textit{NULL} if an error occurs.

**Various functions**

- \textbf{unsigned int commMillisInRecv():}

  Function used by a worker to know how much time it spent in network reception functions. Returns the time spent in milliseconds.

- \textbf{char *workerName(int worker):}

  Function used by the master to know the name of a particular worker (by knowing its processor number). Returns the name (\textit{char *}).

- \textbf{bool commIsMaster():}

  Function used to return \textit{true} if the current worker is the master, or \textit{false} else.

- \textbf{bool commIsLastSlave():}

  Function used to return \textit{true} if the current worker is the last worker (where iCPU = numSlaves).

- \textbf{bool commSlaveRecvInts(DISTRIBINFO *distribInfo):}

  \textit{distribInfo:} a structure containing weight information sent to the spawned slave by the master

  Function called by each slave to receive its own weight information. Modify \textit{*distribInfo} with the information received. Modify also the following global variables: \textit{myIcpu}, and \textit{numSlaves}. Returns a boolean: \textit{true} if no error occur.

**B.2.2 External functions defined in \textit{error.c}**

These functions are declared in \textit{error.h} and defined in \textit{error.c}. They are used in the module of polygonization (\textit{polyg}) to report errors. The error functions use
communication functions, because all slave errors need to be finally treated by
the master.

The error functions rely on the signal API, therefore they are specific to a
type of operating system. Actually they are only implemented to use the POSIX
signal API.

Report error

- **void commReportError(char *error):**
  - error: a character string describing the error.
  - Function used by any worker to report a fatal error. Calls commSlaveReportsError
    or commMasterReportsError, depending of the worker’s case.

Setting and unseting signals

- **void commMasterSetSignalRecv():**
  - Function used by the master to set the signal handlers.

- **void commMasterUnsetSignalRecv():**
  - Function used by the master to unset the signal handlers. It sets back the
    signal handlers that were previously saved.

- **void commWorkerSetSignalRecv():**
  - Function used by any worker to set some common signal handlers.

- **void commWorkerUnsetSignalRecv():**
  - Function used by any worker to unset the signal handlers: it sets back the
    old signal handlers, previously saved.

B.3 Internal functions

B.3.1 Internal functions defined in `comm.c`

- **int chrono_recv(int tid, int tag):**
  - This function wraps the PVM `pvm_recv()` function, with the same argu-
    ments (the expected TID and tag). It determines how much time is spent
    in reception state, and adds it to the timeSpentInRecv variable.

- **int chrono_trecv(int tid, int tag):**
  - It does exactly the same with the `pvm_trecv()` function. Note that this
    one exits after a specified timeout has expired (see the TIMEOUT macro-
    constant at the beginning) of the file.

- **void triggerTimeOutTestOneSlave(int slaveLate):**
  - slaveLate: the index (between 1 and numSlaves) of the slave being late.
  - This function is called when a timeout has occurred while waiting for a
    particular slave (with chrono_trecv). It checks that the slave is still
    alive.

- **void triggerTimeOutTestSeveralSlaves(int *slavesFinished):**
  - slavesFinished: an array (of size numSlaves) of booleans; they indicate
    which slaves are waited for (those with the false value).
This function is called when a timeout has occurred while waiting for any slave (again with `chrono.trecv`). The calling function keeps track of the responding slaves with an array of booleans (the `slavesFinished` parameter); this function goes through all of those where the boolean is `false`, and checks if it is still alive.

### B.3.2 Internal functions defined in `error.c`

For a general explanation about error handling, please see 3.1 page 14.

#### Error handlers

- **`void commMasterRecvSIGUSR1(int sig)`**:
  The master gets in this function when a slave has reported an error. It receives the error explanation, displays it, and jumps to the SIGTERM procedure.

- **`void commMasterRecvSIGINT(int sig)`**:
  This function is triggered when the master receives a SIGINT (through a Ctrl-C for example).

- **`void commMasterRecvSIGTERM(int sig)`**:
  When receiving the SIGTERM signal, the master aborts the whole polygonization.

- **`void commWorkerRecvSIGSEGVBUS(int sig)`**:
  When a worker provokes a segmentation fault (which should not happen!), it aborts the whole polygonization.

These functions are set as signal handlers with the `signal()` POSIX function. The first three ones are set by the master with the `commMasterSetSignalRecv` function; they are triggered by SIGUSR1, SIGINT and SIGTERM. The last function is set as a signal handler by all workers (with `commWorkerSetSignalRecv`) to handle both SIGSEGV and SIGBUS.

#### Dedicated error reporting functions

- **`void commSlaveReportsError(char *error)`**:
  This function is called by `commReportError` when the process is a slave. It reports the error to the master.

- **`commMasterReportsError(char *error)`**:
  This function is called by `commReportError` when the process is the master.
Appendix C

Functions and variables used in polygonization

C.1 Global variables used by the polygonization module

These shared variables are defined in \textit{hfpm\_variables.c}. They represent internal states of the polygonizing process.

Some of the data structures used are defined in \textit{hfPolyTypes.h}, along with the macros to manipulate them.

- **EVNarray itsVNCoordinates**: array of informations on vertices: a global edge number (identifies in a unique way the vertex on the 3D grid), the coordinates of the vertex, and the coordinates of the normal (if needed).

- **uintArray itsVertexIndices**: contains indices in itsVNCoordinates, for reading in increasingly sorted global edge numbers.

- **intPairArray itsEVertices**: Given an edge of the polygonized model, we can retrieve in this array the indices of the two vertices bound by the edge. These indices are the global edge numbers (of the 3D grid) on which the vertices are. For example:
  - \texttt{itsEVertices[e][0]}: gives the global edge number on which the first vertex of \texttt{e} lies;
  - \texttt{itsEVertices[e][1]}: gives the global edge number on which the second vertex of \texttt{e} lies.

- **intTripleArray itsVTriangles**: Given a triangle in the polygonized object, this array permits to retrieve the indices (global edge number of the grid) identifying the vertices of the triangle. \texttt{itsVTriangles[triangleNum][0]}, \texttt{itsVTriangles[triangleNum][1]}, \texttt{itsVTriangles[triangleNum][2]} are these indices.

- **intTripleArray itsTEdges**: given a triangle in the polygonized object, this array helps to retrieve the indices of the edges composing the triangle.
- VNarray itsVertexNormal: an array that contains, for each vertex, its coordinates and eventually the coordinates of its normal. Global edge numbers are not stored here. This array is used to store the vertices (and information on them) at the end of the algorithm after they have been globally renumbered.

- double ***itsValGrid: 3D grid holding the value of the function defining the F-Rep object at each node.

- bool ***itsBoolGrid: 3D grid holding a boolean at each node of the grid, describing whether the function is $> 0$ or $< 0$ in itsValGrid.

- int ***itsVertGrid[3]: For each node $(i, j, k)$ of the 3D grid, you consider 3 edges $(0, 1, \text{and } 2)$. itsVertGrid[e][i][j][k] gives the index of the vertex located on the edge $e \in \{0, 1, 2\}$.

- int ***itsCellGrid: 3D Grid holding bitmap style information about cell faces.

C.2 External functions

These functions are usable outside of this module. They are called for example within the module mainapp.

C.2.1 Functions for initiating and ending polygonization

- void HFPM_INIT_HFPolyMesh(int XSize, double (*Calc)(doubleArray coord, void *interpreter), void *interpreter):
  XSize: size of the X array, usually 3 (Euclidean space).
  Calc: initialized with the function we want to polygonize.
  interpreter: a pointer to the interpreter of the HyperFun language, see the module interpreter.lib.
  Defined in hfpolymesh.c. Initializes the polygonization process, the second argument is the function (defining the F-Rep model) we want to polygonize. Usually Calc is called by passing the “interpreter” value as the second argument.

- void HFPM_DESTRUCT_HFPolyMesh():
  Defined in hfpolymesh.c. Frees the resources allocated during the polygonization and still used by the program after polygonization (they are the structures holding the final results).

- void HFPM_storeWeights(unsigned int *w, int size, unsigned int tw):
  Defined in hfpm_rebalance.c. Function used by the master to copy the parsed weights in a static array of this module (hidden outside of this module).
C.2.2 Functions for setting / getting the polygonizer’s behavior

The following functions are only here to initialize the options with the parsed command line. Options are global variables defined in hfpm_variables.c. They are all declared in hfpolymesh.h and defined in hfpolymesh.c.

- void HFPM_SetTimer(bool t)
- void HFPM_SetNormals(bool s)
- void HFPM_SetIsoValue(double isov)
- void HFPM_SetMinMax(double *mmIN): size of mmIN is 6.
- void HFPM_SetGrid(int *gIN): size of gIN is 3.
- void HFPM_SetDimMap(int *dmIN): Size of dmIN is 3
- void HFPM_SetConstants(doubleArray *conIN): Size of conIN is unknown.

The following functions are here for getting the options stored with the previous functions in global variables. They are all declared in hfpolymesh.h and defined in hfpolymesh.c.

- void HFPM_GetTimer(bool t)
- void HFPM_GetNormals(bool s)
- void HFPM_GetIsoValue(double isov)
- void HFPM_GetMinMax(double *mmIN): size of mmIN is 6.
- void HFPM_GetGrid(int *gIN): size of gIN is 3.
- void HFPM_GetDimMap(int *dmIN): Size of dmIN is 3
- void HFPM_GetConstants(doubleArray *conIN): Size of conIN is unknown.

C.2.3 The main function

HFPM_Calc(DISTRIBINFO *distribInfo) is the main function of the polygonizer. This function wraps all the functions of the polygonization module and calls them. This function is declared in hfpolymesh.h and defined in hfpm_calc.c. distribinfo contains, for each processor (node), all the information concerning its weight. Be aware that the structure DISTRIBINFO is defined in mainapp/mainapp.h.
C.2.4 Functions for retrieving the results of polygonization

These functions are declared in hfpolymesh.h. They are used in the code of the main application to retrieve the results of the polygonization process.

Defined in hfpolymesh.c

- **int HFPM_VertexNum()**: It returns the total number of vertices that have been found.
- **int HFPM_TriangleNum()**: It returns the total number of found triangles.

Defined in hfpm_tridata.c

- **double TD_getVertex2(int v, int x)**: It returns the \( x \)th coordinate of the vertex \( v \).
- **double TD_getNormal2(int n, int x)**: It returns the \( x \)th coordinate of the normal \( n \).
- **int TD_getTriangle2(int t, int vnum)**: \( vnum \in \{0,1,2\} \). It represents the three vertices of the triangle \( t \).

C.3 Internal functions

The functions described here are all declared in private.h. They are used within the polyg module, and should not be called outside. They can be sorted in three classes, according to the data on which they work:

- cell functions (C.3.1),
- triangle functions (C.3.2),
- various functions (C.3.3).

C.3.1 Functions dealing with cells

- **void HFPM_findComputedCells (DISTRIBUTINFO *distribInfo)**: Defined in hfpm_cells.c.
  Function used by a worker to know what cells are to be treated. It uses the weight of the current worker (stored in distribInfo) to know how many cells will be treated by this worker. This function initially fills CellsAndTriangles, a variable locally defined.

- **void HFPM_Cells_DESTRUCT()**: Defined in hfpm_cells.c.
  Function used to free the memory allocated for CellsAndTriangles (allocated in HFPM_findComputedCells).
- **void HFPM_SetNumTrianglesInCurrentCell(unsigned int numTriangles):**
  Defined in hfpm_cells.c.
  Function used to store the approximative number of triangles numTriangles found in a cell. Modifies totalNumTrianglesApprox.

- **unsigned int HFPM_totalNumTrianglesApprox():**
  Defined in hfpm_cells.c.
  Function used to know the approximative total number of triangles that the current worker would have to deal with, in its owned cells. This number is in fact stored in the global variable totalNumTrianglesApprox.

- **unsigned int HFPM_numCellsToSend(unsigned int numTriangles):**
  Defined in hfpm_cells.c.
  Function used to return an integer \( N \) so that the first \( N \) cells in cellsAndTriangles contain numTriangles triangles (by summing the number of triangles in each cell).

- **void HFPM_findNextCellToSend(sentCellInfo *info, unsigned int iSend, unsigned int numCellsToSend):**
  Defined in hfpm_cells.c.
  This function finds the next cell to send and deletes it (the first non-empty, actually). This function is called from commWorkerTreatMoves in comm/comm.c.
  “Deleting” means erasing the information stored in cellsAndTriangles concerning that cell. We do this at the end of cell re-balancing, when all the cells that needed to be removed were removed for a worker.

- **void HFPM_addNewCell(sentCellInfo *info):**
  Defined in hfpm_cells.c.
  Function used by a worker when receiving a cell from another worker. This updates the internal informations of the worker. It adds the new owned vertices, and adds the other vertices (which are in the cells but not owned).

- **void HFPM_startCellEnumeration():**
  Defined in hfpm_cells.c.
  Initializes the static global variable iterator, defined in hfpm_cells.c. This iterator is used to know on which cell we are currently working, in an enumeration. This function is to be used with HFPM_nextCell.

- **void bool HFPM_nextCell(int *i, int *j, int *k):**
  Defined in hfpm_cells.c.
  i, j, k: identify the next cell (in the 3D grid) on which we will work next. They are modified by this function.
  The boolean returned indicates whether there was actually a next cell (true), or if we have reached the end of the list (false). This boolean can be used to end an enumeration loop.

### C.3.2 Functions dealing with triangles

- **int TD_pushBackTriangle(int v[3]):**
  Defined in hfpm_tridata.c.
Function used to register a triangle, knowing its 3 vertices. It is called in the module of triangle building (*hfpm_createtriangles.c*).

- **void HFPM_gatherNumTrianglesApprox()**: Defined in *hfpm_rebalance.c*. Function used by every worker (both the master and the slaves). The first step is to send (slave’s case) or receive (master’s case) the triangle numbers. The second step for the master is to create a list of triangle moves, and to broadcast it to the slaves. The last step is, for every worker, to exchange its triangles according to the list of moves. This function calls the functions defined in *comm/comm.c* for moving triangles among the workers.

- **void HFPM_CreateTriangles()**: Defined in *hfpm_createtriangles.c*. It goes through all the found vertices and groups them in triangles.

- **unsigned int HFPM_RevertTriangles()**: Defined in *hfpm_createtriangles.c*. This function is called by the master at the end of polygonization, when all triangles have been computed. The functions considers each triangle to see if it is clockwise oriented: if it is not, it corrects the vertex order accordingly.

- **void HFPM_ConvertLocalNumsInTriangles()**: Defined in *hfpm_createtriangles.c*. This function is used by all workers to convert vertex numbers expressed with indices in *itsVNCoordinates* into global edge numbers.

### C.3.3 Various functions

- **void TD_guessSize(int tNum, int vNum)**:
  - *tNum* is the number of triangles
  - *vNum* is the number of vertices
  - Defined in *hfpm_tridata.c*. It guesses the number of edges in the polygonized model. Please note that *tNum* and *vNum* are already an approximation. The number of edges is needed to allocate several data structures. These data structures (whose sizes depend on the number of edges) are also allocated in this function.

- **void HFPM_CreateNormals()**: Defined in *hfpm_createnormals.c*. Function used to compute the normal at each vertex.

- **void HFPM_BroadcastVNCoordinates(DISTRIBINFO *distribInfo)**: Defined in *hfpm_multicast.c*. Function used by all workers to:
  1. broadcast the vertex indices;
  2. send the vertex [and normal] coordinates to the master;
  3. send the triangles to the master.
- `void HFPM_FinalizeData();`  
  Defined in `hfpm_FinalizeDataAlone.c`.  
  This function has been added for the case of mono-CPU run. It substitutes to the `HFPM_BroadcastVNCoordinates` function.

- `HFPM_SortVertexIndices();`  
  Defined in `hfpm_fillvertices.c`.  
  Function used to sort the vertices in `itsEVNCoordinates` (with a quick-sort algorithm).