



Scotch and PT-Scotch 7.0 Hands-On Guide

(version 7.0.18)

François Pellegrini
Université de Bordeaux & LaBRI, UMR CNRS 5800
TadAAM team, INRIA Bordeaux Sud-Ouest
351 cours de la Libération, 33405 TALENCE, FRANCE
francois.pellegrini@u-bordeaux.fr

October 31, 2026

Contents

1 Introduction **2**

2 Compiling and executing Scotch **3**
2.1 Required tools 3
2.2 32 or 64 bits? 3
2.2.1 Integer size issues in Scotch 4
2.2.2 Integer size issues in PT-Scotch 4
2.3 Linking with the proper Scotch library 4
2.4 Linking with the proper Scotch error handling library 5
2.5 Multi-threading 5
2.6 Execution issues 6
2.6.1 Intel MPI 6

3 Programming with Scotch **7**
3.1 Integer type 7
3.2 Data structures 7
3.2.1 Allocation of data structures 7
3.2.2 Initialization and deconstruction of data structures 8
3.3 Error handling 8
3.4 File input/output 9
3.5 Checking user input 9
3.6 Basic use of Scotch 9
3.6.1 Graph partitioning 9
3.6.2 Static mapping 10
3.6.3 Sparse matrix ordering 10
3.7 Multi-threading 10

4 Running Scotch **10**
4.1 Multi-threading 11

5 Troubleshooting **11**
5.1 Troubleshooting check-list 12

1 Introduction

This document is a quick reference guide for people willing to use Scotch in their projects. By "Scotch", we designate collectively both the centralized-memory software Scotch and the distributed-memory PT-Scotch software, which are components of the Scotch project.

This guide provides information about the proper way to install and use Scotch, by providing snippets of code and simple examples. It is therefore incomplete by nature. Users willing to obtain more information may refer to the following documents:

- The Scotch User's Manual;
- The PT-Scotch User's Manual;
- The source code of the various test programs in the `src/check` directory.

Users willing to install their own local version of Scotch should also refer to the following documents:

- The `README.md` file;
- The `INSTALL.txt` file.

People willing to do research on graph partitioning with Scotch and implement their own algorithms in Scotch may refer to the Scotch Maintenance Manual.

2 Compiling and executing Scotch

2.1 Required tools

In order to compile Scotch, one needs:

- GNU make;
- possibly CMake, version at least 3.10;
- a C compiler that can handle the C99 standard (and is parametrized to do so);
- Flex and Bison, version at least 3.4;
- a MPI implementation, for PT-Scotch.

2.2 32 or 64 bits?

Scotch libraries can typically be found in 32 bit or 64 bit implementations. It is possible to use both, but this requires a bit of tweaking (using the `SCOTCH_NAME_SUFFIX` flag), because two coexisting versions cannot expose simultaneously routines with same names, all the more when they expect integer types of different sizes. In most of the cases, however, only one version will be linked against the user's program, which requires no specific action.

One should not mistake the size of machine integers, that is, `int`s, and the size of integer values used within Scotch, that is, `SCOTCH_Num`s and `SCOTCH_Idx`s.

- `SCOTCH_Num` is the generic Scotch integer type for values. It is used, e.g., as the cell type for vertex and edge arrays that describe graphs. Its width can be defined at compile time by way of the `INTSIZE32` or `INTSIZE64` flags. By default, its width is that of the `int` type.
- `SCOTCH_Idx` is the generic Scotch memory index type. Its width should be that of the address space. It is used, e.g., to represent the global amount of memory consumed by Scotch routines (see, e.g., `SCOTCH_memMax()`), or the indices in memory of the cells of vertex and edge arrays, with respect to a given reference (see, e.g., `scotchfgraphdata()`). Its witdh can be defined at compile time by way of the `IDXSIZE32` or `IDXSIZE64` flags. By default, its width is that of the `int` type.

2.2.1 Integer size issues in Scotch

Since Scotch does not depend on third-party libraries, there is no risk induced by using Scotch integer datatypes of a size that differs from that of the `int` type. One may use 32-bit `SCOTCH_Num`s in a 64-bit environment, in order to save memory, or use 64-bit `SCOTCH_Num`s in a 32-bit environment, to handle big graphs (provided the address space is large enough). In all cases, the width of `SCOTCH_Idx` should be that of an address, that is, nowadays, 64 bits.

RULE: `sizeof (SCOTCH_Num) <= sizeof (SCOTCH_Idx) .`
RULE: `sizeof (SCOTCH_Idx) >= sizeof (void *) .`

2.2.2 Integer size issues in PT-Scotch

The fact that `SCOTCH_Num`s are bigger than `int`s may cause integer overflow issues in PT-Scotch. Indeed, in the prototypes of most communication routines of the MPI interface, counts and array indices are declared as `int`s. Consequently, for big graphs comprising more that 2 billion vertices and/or edges, that can only be represented with 64-bit `SCOTCH_Num`s, it may happen that message counts and displacement values go beyond the 2 billion boundary, inevitably resulting in a communication subsystem crash.

The ability to use 64-bit `SCOTCH_Num`s with 32-bit MPI implementations was a temporary hack to break the "2-billion barrier", in a time when there were no 64-bit implementations of MPI, while knowing that there would be a breaking point beyond which this solution would not work (*i.e.*, when the amount of data to be exchanged also breaks this boundary).

To be on the safe side when using PT-Scotch on big graphs, the width of the `SCOTCH_Num` datatype should always be that of the `int` datatype, whatever it is.

ADVICE: `sizeof (SCOTCH_Num) == sizeof (int) .` This is the behavior by default.
ADVICE: To use PT-Scotch on big graphs, use 64-bit `int`s and link against a 64-bit implementation of MPI.

```
1  if (sizeof (SCOTCH_Num) > sizeof (int)) {
2    SCOTCH_errorPrintW ("PT-Scotch users, beware: here be dragons");
3    proceedWithCaution ();
4  }
```

2.3 Linking with the proper Scotch library

Users should make sure that they link their programs with the proper Scotch library, that is, the library whose type width matches the one which was defined in the `scotch.h` and `ptscotch.h` header files.

This can be verified dynamically at run time, by comparing the value returned by the `SCOTCH_numSizeof()` routine with that defined in the Scotch header files that were used at compile time.

RULE: `SCOTCH_numSizeof () == sizeof (SCOTCH_Num) .`
ADVICE: Insert a sanity check at the beginning of your code, to rule out this issue.

```
1  if (SCOTCH_numSizeof () != sizeof (SCOTCH_Num)) {
2    SCOTCH_errorPrint ("Don't even try to call any Scotch routine");
3    betterQuitNow ();
4  }
```

2.4 Linking with the proper Scotch error handling library

When a SCOTCH routine encounters an error, it generates an error message, by calling the `SCOTCH_errorPrint()` routine, and tries to returns an error value. By design, this routine is not part of the standard SCOTCH libraries, to allow editors of third-party software to funnel SCOTCH error messages to their own error logging system, by providing their own implementation of `SCOTCH_errorPrint()`. SCOTCH users who do not want to undertake this task have just to link their software with one of the defaut error handling libraries that are part of the SCOTCH distribution.

- `libscotcherr`, which sends the error message to the standard error stream, and tries to return an error value to the caller routine. Several error messages may be produced, as control returns to upper layers of SCOTCH routines and errors are detected in turn;
- `libscotcherrexit`, which sends the error message to the standard error stream, and exits immediately after. No subsequent messages are produced, which may have helped to locate the error;
- `libptscotcherr`, which sends the error message, comprising the MPI process number, to the standard error stream, and tries to return an error value to the caller routine;
- `libptscotcherrexit`, which sends the error message, comprising the MPI process number, to the standard error stream, and exits immediately after.

RULE: Do not link with a `libptscotcherr*` library if you do not use MPI within your program.
ADVICE: Link with a `libptscotcherr*` library when using PT-SCOTCH routines within a MPI program.

2.5 Multi-threading

Since version v7.0, SCOTCH benefits from dynamic multi-threading: most compute-intensive algorithms will use several threads, whenever available. The use of threaded algorithms has to be activated by setting some compilation flags, notably:

- `COMMON_PTHREAD`: activate threads at the service routine level (e.g.: I/O compression-decompression).
- `COMMON_PTHREAD_AFFINITY_LINUX`: use the Linux API for thread affinity management. Indeed, to enhance memory locality and efficiency, it is better for each thread to be assigned to a given processing element. While the API for this feature is not normalized, the Linux API is quite standard. No other thread affinity API is used in SCOTCH at the time being.
- `SCOTCH_PTHREAD`: activate threads for the shared-memory algorithms of SCOTCH and PT-SCOTCH (which do not require to have a thread-safe implementation of MPI);

- SCOTCH_PTHREAD_MPI: activate threads for the distributed-memory algorithms of PT-Scotch. Starting from v7.0.4, PT-Scotch dynamically adapts its behavior to the thread-safety level of the MPI library against which it is linked. In particular, it will take advantage of the capabilities of the MPI_THREAD_MULTIPLE level, if it is enabled, to run some threaded algorithms in parallel across compute nodes.

- SCOTCH_PTHREAD_NUMBER: default maximum number of threads to be used. A value of 1 means that no multi-threading will take place in absence of specific user action, and a value of -1 that Scotch will use all the threads provided by the system at run time. Set to -1 by default if no value provided.

These flags are usually activated in compilation configuration files, but this may depend on the packager for your distribution and of the thread-safety level of your local MPI package. For each of these classes, there exist sub-flags to activate or deactivate specific features and algorithms (e.g., COMMON_PTHREAD_FILE). For CMake, these flags may have different names. Please refer to the INSTALL.txt file of your version of Scotch.

Additionnally, environment variables may be set to dynamically change the behavior of the libScotch and all the executables that rely on it (including the Scotch standalone programs themselves). These variables take precedence over the compilation flags with same purpose.

- SCOTCH_PTHREAD_NUMBER: this environment variable sets the prescribed maximum number of threads that Scotch may use in the course of its computations. A value of -1 indicates to use as many threads as provided by the system at launch time. Setting this number to 1 will coerce Scotch into using only purely sequential algorithms (which may differ in nature from their multi-threaded counterparts).

- SCOTCH_DETERMINISTIC: when set to 0, faster, non-deterministic, multi-threaded algorithms may be used; when set to 1, fully deterministic, albeit sometimes slower, multi-threaded algorithms are always used.

- SCOTCH_RANDOM_FIXED_SEED: when set to 1, the same pseudo-random seed is used for each run, allowing for reproducibility in case only deterministic algorithms are used; when set to 0, a new pseudo-random seed is used for each run.

RULE: Only activate multi-threaded, distributed memory algorithms at compile-time if your local MPI implementation supports it.
ADVICE: Activate threads whenever possible, since they are likely to speed-up computations.
ADVICE: Activate the extension for thread affinity whenever possible.

2.6 Execution issues

2.6.1 Intel MPI

Some hanging issues have been reported when running PT-Scotch in multi-threaded mode on Intel MPI. Until Intel fixes this issue, a workaround, available since Intel MPI version 2021.17, is to run Intel's mpiexec command with some configuration variables set as follows:

```
I_MPI_THREAD_SPLIT=1 I_MPI_THREAD_SPLIT_MODE=implicit I_MPI_THREAD_MAX=x
```

where the x integer value is the maximum number of threads that Scotch will use, which must be set explicitly at compile time (see Section 2.5), or at execution time as an environment variable:

```
export SCOTCH_PTHREAD_NUMBER=x
```

3 Programming with Scotch

3.1 Integer type

All integer values used by Scotch to describe its objects are of type SCOTCH_Num, which may not be the standard integer type, depending on compilation options (see Section 2.2). Make sure to declare and use SCOTCH_Nums in your program whenever necessary, even for integer constants.

```
#include <scotch.h>
  ...
  SCOTCH_Num vertnum;
```

```
INCLUDE "scotchf.h"
  ...
  INTEGER*SCOTCH_NUMSIZE VERTNUM
```

```
include 'scotchf.h'
  ...
  integer(SCOTCH_NUMSIZE) :: vertnum
```

RULE: In FORTRAN, in the absence of function prototypes, always pass integer constants through variables of type SCOTCH_Num, to make sure the integer type width will always be the proper one.

3.2 Data structures

All data structures in the Scotch API are "opaque objects", that is, mock data structures meant to hide their contents from the user. Hence, interactions with Scotch can only take place through its API, which preserves ascending compatibility with future versions (and backwards compatibility as well, to some extent).

3.2.1 Allocation of data structures

Scotch data structures can be allocated at compile time by specifying memory areas of the adequate size. For data alignment concerns, these structures are exposed as arrays of doubles, of a size defined in the API.

```
#include <scotch.h>
  ...
  SCOTCH_Graph grafdat;
```

```
INCLUDE "scotchf.h"
  ...
  DOUBLEPRECISION GRAFDAT (SCOTCH_GRAPHDIM)
```

```
include 'scotchf.h'
 ...
 doubleprecision, dimension (SCOTCH_GRAPHDIM) :: grafdat
```

Alternately, users can allocate memory for SCOTCH objects from the heap at run time. The size of these objects can be based on the size values provided at compile time (e.g., SCOTCH_GRAPHDIM), or returned by the API routines SCOTCH_*Sizeof(). In the latter case, users will be able to link with any version of the SCOTCH library without having to recompile their code.

Users may also directly obtain a memory pointer to dynamically allocated SCOTCH objects, using the C API SCOTCH_*Alloc() routines. In this case, this memory must be freed using the SCOTCH_memFree() routine.

3.2.2 Initialization and deconstruction of data structures

Whether they are statically or dynamically allocated, all SCOTCH opaque objects should be initialized before use. To do so, one must use the *Init() routine defined for each object category.

Similarly, after their last use, these structures must be deconstructed, using the *Exit() routine associated with each object category. The deconstructor routine is needed to free the internal memory associated with the object. When objects depend on other objects, by way of references, the referenced object must not be deconstructed before the referring object (e.g., a SCOTCH_Graph refrenced by a SCOTCH_Mapping).

3.3 Error handling

For the C-language API of SCOTCH, errors are usually reported by returning a non-zero *int* value ; please check the user's manuals for more information. For the FORTRAN-language API, errors are reported through a specific variable, of integer type, which is passed as the last argument of SCOTCH routines. This variable should be initialized to 0, and checked afterwards to see whether its value has changed. For both APIs, the type of these error values is a standard int, not a SCOTCH_Num.

```
SCOTCH_Graph grafdat;
FILE * fileptr;

if (SCOTCH_graphLoad (&grafdat, fileptr, -1, 0) != 0) {
  fprintf (stderr, "Could not read graph\n");
  fclose (fileptr);
  exit (EXIT_FAILURE);
}
```

```
DOUBLEPRECISION GRAFDAT (SCOTCH_GRAPHDIM)
INTEGER FILENUM
INTEGER IERR

IERR = 0
...
CALL SCOTCHFGRAPHLOAD (GRAFDAT (1), 42, -1, 0, IERR)
IF IERR .NEQ. 0 THEN
  PRINT *, "Could not read graph"
  CALL CLOSE (42)
  CALL EXIT_C (EXIT_FAILURE)
END IF
```

3.4 File input/output

C-language API routines handle files as streams, of type `FILE *`. This allows users to read and/or write from any kind of I/O structure (e.g., regular files, but also pipes, network sockets, etc.). FORTRAN-language API routines handle files as I/O units, which are turned internally into file descriptors and then into C streams, by adding buffering capabilities. This added buffering improves performance but may bring input over-consumption, if the same unit is used for different read operations.

```
SCOTCH_Graph grafdat;
FILE * fileptr;

SCOTCH_graphInit (&grafdat);
fileptr = fopen ("path/to/the/graph/file.grf");
SCOTCH_graphLoad (&grafdat, fileptr, -1, 0);
fclose (fileptr);
```

```
DOUBLEPRECISION GRAFDAT (SCOTCH_GRAPHDIM)
INTEGER FILENUM
INTEGER IERR

IERR = 0
CALL SCOTCHFGRAPHINIT (GRAFDAT (1), IERR)
OPEN (UNIT = 42, FILE = "path/to/the/graph/file.grf")
CALL SCOTCHFGRAPHLOAD (GRAFDAT (1), 42, -1, 0, IERR)
CLOSE (UNIT = 42)
```

3.5 Checking user input

Scotch provides graph consistency checking routines, both for centralized and distributed graphs. These routines will perform a set of sanity checks on the graph data (e.g., assert that when vertex v' is in the list of neighbors of v, then v is in the list of neighbors of v', etc.).

Before considering reporting an issue to the Scotch team, please run the graph checking routine on your graph structure (similarly, the `gtst` and `dgtst` command-line programs check the consistency of graph files). If the graph consistency checking routine reports an error, then all subsequent Scotch errors are likely to derive from this non-consistent input. If no error is reported and Scotch crashes afterwards, then the issue is most likely on Scotch's side.

> **ADVICE**: In the early stages of development of your program and/or in debug mode, always call the relevant Scotch graph checking routine (`SCOTCH_graphCheck()` or `SCOTCH_dgraphCheck()` before launching partitioning tasks.

3.6 Basic use of Scotch

The behavior of Scotch can be parametrized in many ways (threading, strategy strings, etc.). However, routines have been designed for a simplified use, for graph partitioning, static mapping, and sparse mapping ordering.

3.6.1 Graph partitioning

```
SCOTCH_Strat stradat;
SCOTCH_Graph grafdat;
```

```
SCOTCH_Num vertnbr;
SCOTCH_Num partnbr;
SCOTCH_Num * parttab;

SCOTCH_StratInit (&stradat); /* Default strategy will be used */
SCOTCH_graphInit (&grafdat);
... /* Fill-in graph structure */
SCOTCH_graphSize (&grafdat, &vertnbr, NULL);
parttab = (SCOTCH_Num *) malloc (vertnbr * sizeof (SCOTCH_Num));
partnbr = ... /* Set number of parts */
SCOTCH_graphPart (&grafdat, partnbr, &stradat, parttab);
SCOTCH_graphExit (&grafdat);
SCOTCH_StratExit (&stradat);
```

```
doubleprecision :: stradat (SCOTCH_STRATDIM)
doubleprecision :: grafdat (SCOTCH_GRAPHDIM)
integer(SCOTCH_NUMSIZE) :: vertnbr
integer(SCOTCH_NUMSIZE), allocatable :: parttab (:)
integer :: ierr

ierr = 0
call scotchfstratinit (stradat (1), ierr)
call scotchfgraphinit (grafdat (1), ierr)
... ! Fill-in graph structure
call scotchfgraphsize (grafdat (1), vertnbr, grafdat (1), ierr)
allocate (parttab (vertnbr))
partnbr = ... ! Set number of parts
call scotchfgraphpart (grafdat (1), partnbr, stradat (1), parttab (1),
    ierr)
call scotchfgraphexit (grafdat (1))
call scotchfstratexit (stradat (1))
```

3.6.2 Static mapping

Static mapping is carried out in the same way as graph partitioning. The sole difference is that, instead of providing a number of parts, one has to provide a target architecture, on a prescribed topology. See the Scotch user's manual for all the available sorts of target architectures, including how to turn a graph into a target architecture.

3.6.3 Sparse matrix ordering

3.7 Multi-threading

Since version `v7.0`, Scotch implements a dynamic thread management system.

RULE: Only activate multi-threaded, distributed memory algorithms at compile-time if your local MPI implementation supports it.
ADVICE: 4 to 8 threads per MPI process is a good compromise in terms of performance.

4 Running Scotch

In addition to the functions it provides through the Scotch and PT-Scotch libraries, the Scotch project provides a set of standalone programs to perform basic functions: graph and target architecture management, partitioning and mapping, and sparse matrix reordering.

Centralized-memory programs can be launched directly from the command line, while distributed-memory program, based on MPI, have to be launched through a specific program like `mpiexec` or `mpirun`, depending on the MPI implementations.

4.1 Multi-threading

Since version `v7.0`, Scotch implements a dynamic thread management system. Without specific user instruction, Scotch programs and routines will try to use all the threads available on the node. The user can coerce the behavior of Scotch at run time by using the `SCOTCH_PTHREAD_NUMBER` environment variable, which can override the `-DSCOTCH_PTHREAD_NUMBER=x` definition provided at compile time. Setting its value to `-1` lets the software use all the threads available, while a positive value defines the maimum number of threads that can be used at run time, among all the available threads.

```
% export SCOTCH_PTHREAD_NUMBER=2
% gpart 5 /tmp/brol.grf /tmp/brol.map -vmt
% export SCOTCH_PTHREAD_NUMBER=4
% gpart 5 /tmp/brol.grf /tmp/brol.map -vmt
% export SCOTCH_PTHREAD_NUMBER=-1
% gpart 5 /tmp/brol.grf /tmp/brol.map -vmt
```

When running PT-Scotch in a multi-threaded way, and in case several MPI processes are mapped onto the same compute nodes, it is important to ensure that each of these processes will not try to create as many threads as it can on its node. Else, a plethora of competing threads would lead to huge performance loss on each node. Options can be passed to the `mpiexec` command to make sure each MPI process spawned on some node is assigned a non-overlapping set of thread slots, within which it can safely create a smaller set of threads.

ADVICE: Make sure MPI processes have dedicated, non-overlapping, thread slots attached to them, so that the threads they may create will not overlap and compete for the same thread slots on the same compute nodes.
ADVICE: In case non-overlapping thread slots cannot be created for MPI processes, coerce Scotch to using only one thread per MPI process.

5 Troubleshooting

Although Scotch is a mature and extensively tested software, it is still possible that remaining bugs may show off in specific cases, or that specific configurations and graph topologies may induce an exceptional behavior (poor partition quality, etc.).

Most systematic errors, which occur when using Scotch for the first time, are due to configuration issues, such as integer type mismatch (see Section 2.2). These errors can be ruled out quite quickly, e.g. by comparing the sizes of integer types used by the application and by Scotch, etc.

In case of execution errors (e.g., memory shortage, or other run-time errors), Scotch routines will output an error message on the standard error stream, and either return an error value or terminate the program, depending on the Scotch error handling library against which the program is linked (see Section 2.4).

Debug compilation flags can make Scotch perform more internal checks and provide finer insights into execution issues. Of course, the level of extra

sanity checking impacts compute time. The most relevant debugging level for assessing issues is obtained by compiling Scotch with the flag -DSCOTCH_DEBUG_ALL.

5.1 Troubleshooting check-list

1. Check whether the Scotch test programs run without errors.
2. Check that your execution environment does not fall into the categories listed in Section 2.6; if yes, apply the provided workarounds, if they exist.
3. Check integer size match between your code and the libScotch, using the library function SCOTCH_numSizeof(). This general sanity check may be beneficially included in all software linked against the libScotch library.
4. In case Scotch works for small cases, yet produces "out of memory" errors for bigger cases when the number of vertices and/or edges nears the billion, Scotch must be recompiled in 64-bit integer mode, by setting the INTSIZE64 flag (see Section 2.2); the IDXSIZE64 flag should always be set on current architectures.
5. Apply the Scotch input data consistency checking routines (e.g., SCOTCH_graphCheck(), SCOTCH_dgraphCheck(), etc.) before calling partitioning or ordering routines.
6. Compile Scotch with symbolic debugging options, so that system error messages (e.g., stack traces) are as informative as possible.
7. Whenever possible, run a memory checker (e.g., Valgrind) on the executables. Alternately, use a memory library that allows for consistency checking (e.g., by setting the MALLOC_CHECK_ environment variable when using the glibc). Alternately, compile Scotch with the -DCOMMON_MEMORY_CHECK flag, to activate a (minimal) memory consistency checking in Scotch.
8. Whenever possible, try to find the smallest possible reproducer, in terms of graph size and/or number of nodes and threads.

ADVICE: Configure your execution environment so that error messages are collected and displayed to the end user. For PT-Scotch, this may require to tweak your MPI execution environment so as to funnel error messages from remote nodes.
RULE: Before reporting a bug to the Scotch team, always run Scotch in debug mode and collect its error messages from the standard error stream.