Par$k$way
A Parallel Hypergraph Partitioning Tool

Version 2.0

Aleksandar Trifunovic and William Knottenbelt
Department of Computing, Imperial College, London
Email: $\{$at701,wjk$\}$@doc.ic.ac.uk

January 5, 2005

Contents

1 Introduction

A hypergraph is an extension of a graph data structure in which edges (*hyperedges*) are allowed to connect arbitrary, non-empty sets of vertices. Like graphs, hypergraphs can be used to represent the structure of many sparse irregular problems. Also like graphs, hypergraphs may be partitioned such that a cut metric or objective function (a function of the interconnect between the parts) is optimised subject to a load balancing criterion. However, hypergraph cut metrics provide a more accurate model than graph partitioning in many cases of practical interest. These include sparse matrix partitioning for parallel matrix-vector multiplication [4, 5, 3, 6, 2, 16, 17] and VLSI circuit design [9, 1].

2 How Does Par$k$way Work?

Par$k$way is an MPI-based parallel library that has been developed during research into parallel hypergraph partitioning at Imperial College London. Algorithms for serial hypergraph partitioning have been studied extensively [7, 1, 11] and tool support exists (e.g. hMeTiS [10] and PaToH [4]). However, these are limited by the computing power and memory available on a single processor. Partitioning in parallel can offer two-fold benefits: it can tackle significantly larger problems than possible serially and it may be able to partition a given problem faster than the equivalent serial algorithm.

2.1 Parallel Hypergraph Partitioning

The multilevel paradigm has been the preferred approach in serial hypergraph partitioning [8]. A parallel formulation was first presented in [15]. The algorithms in Par$k$way are based on parallel algorithms described in [14, 13]. Schematically, the algorithm pipeline is described in Fig. 1.

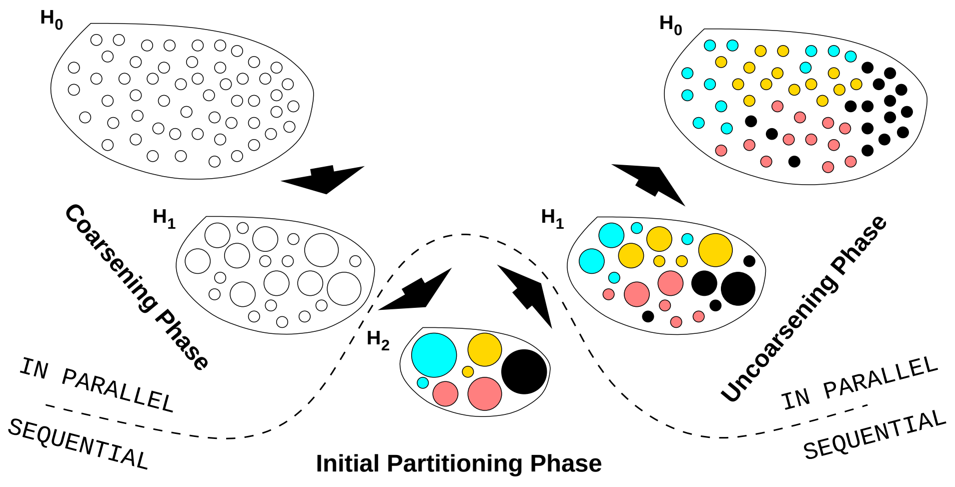

Figure 1. The Parallel Multilevel Pipeline

2.2 Algorithms Used In Par$k$way

Par$k$way uses parallel algorithms during the coarsening and the uncoarsening phases of the multilevel paradigm. The initial partitioning phase is carried out with multiple runs of the serial partitioning algorithm.

Parallel Coarsening Phase During this phase, a sequence of successively smaller hypergraphs is constructed, such that the structure of the original hypergraph is maintained as much as possible. Par$k$way uses a parallel formulation of the $FirstChoice$ coarsening algorithm.

Initial Partitioning Phase During this phase, a partition of the smallest hypergraph is computed by a heuristic partitioning algorithm. Par$k$way can use a generic recursive bisection algorithm and it provides an interface to functions from the hMeTiS [10] or PaToH [4] libraries.

Parallel Uncoarsening Phase During this phase, a partition of the smallest hypergraph is projected through the sucessively larger hypergraphs, that were constructed during the coarsening phase. At each successive level, the projected partition can be further refined to improve its quality, because the larger hypergraph has more degrees of freedom. Par$k$way uses a parallel formulation of the greedy k-way refinement algorithm [14]. Furthermore, the resulting refined partition can be used as a starting point of an iterative cycle of coarsening and uncoarsening (known as a V-Cycle [8]).

The remainder of this manual is organized as follows. Section 3 is a guide to the Par$k$way library interface. Section 4 describes the format of the input and output parameters to the library routines. Section 5 describes the system requirements and provides contact information.

3 Using the Par$k$way Library Interface

Two partitioning routines are provided in the Par$k$way library, depending on the way that the hypergraph is stored in the calling program. Routines are provided in the case that the hypergraph is stored in memory and in the case that it stored in files on disk. The following subsections discuss the routines in more detail.

3.1 Partitioning Routines

The partitioning routines **ParaPartKway** are used to compute a k-way partition of a hypergraph on p processors, assuming that one process is allocated to each processor. That is to say, the vertex set of the hypergraph is partitioned into k disjoint sets (parts) such that a balance criterion on the part weights is satisfied and an objective function over the hyperedges is minimised. Part weight is computed as the sum of the constituent vertex weights. The objective function is the $k-1$ metric, computed as the sum over all hyperedges spanning more than one part, of hyperedge weight times the number of parts spanned by the hyperedge minus one. There is some restriction on the values of k for which a partition is computed - the reader is referred to the description of the `nparts` parameter in the partitioning routines below.

3.1.1 Hypergraph to be read in from disk

ParaPartKway (char `*in_file`, char `*out_file`, int `nparts`, double `constraint`,
int `&k-1cut`, int `*options`, MPI_Comm `comm`)

Parameters

`in_file` String from which the hypergraph file name is constructed. The process with rank i will read in its portion of the hypergraph from the file `in_file-i`. Files `in_file-0` to `in_file-(p-1)` must be provided by the user.

out_file String representing the name of the file that program output (such as program information during execution) should be written to. If out_file $=$ NULL, then program information will be written to **cout**.

nparts Integer specifying the number of parts in the partition, however, there are some constraints on nparts to bear in mind - these are considered in Section 3.1.3.

constraint Double-precision floating point value that specifies the balance constraint on the computed partition. The weight of the largest partition must not exceed $\lfloor (1+\epsilon) W_{avg} \rfloor$, where $\epsilon =$ constraint, assuming integer weights on the vertices.

k-1cut Integer that stores the cutsize value of the best computed partition. The cutsize is computed using the $k-1$ metric.

options Integer array of length 29 that sets the user defined options in **ParaPartKway**. The meaning of the individual entries in the array is described in Section 3.2.

comm The MPI communicator of the processes that call **ParaPartKway**.

3.1.2 Hypergraph stored in memory

ParaPartKway (int nvertices, int nhedges, int *vwts, int *hedgewts, int *hoffsets, int *pinlist, int nparts, double constraint, int &k-1cut, int *options, int *pvector, char *out_file, MPI_Comm comm)

Parameters

nvertices Integer specifying the number of vertices stored by the process. This means that the process stores the vertex weights corresponding to the nvertices locally stored vertices. These vertices must be contiguous in terms of their index. Vertex with index 0 must reside on process with rank 0.

nhedges Integer specifying the number of hyperedges stored by the process. The hyperedges (i.e. the indices of the vertices belonging to each hyperedge) are stored in a $pinlist$ format, as shown in Fig. 2.

vwts Integer array of size nvertices containing the weights of the locally stored vertices. Local vertex with lowest index has weight vwts[0]. Consecutive entries in the array store the weights of consecutive vertices, in terms of their index.

hedgewts Integer array of size nhedges that stores the weights of the locally held hyperedges. The weight of the ith hyperedge in the pin list is stored in hedgewts[i].

hoffsets Integer array of length nhedges+1 specifying the offsets of the hyperedge in the pinlist array. Hyperedge i is stored between the hoffsets[i] and hoffsets[i+1] entries in the pinlist array. That is to say, hyperedge i has length hoffsets[i+1]$-$hoffsets[i] and the index of its first vertex is stored in pinlist[hoffsets[i]]. By default, it needs to be the case that hoffsets[0] = 0.

pinlist Integer array that stores the list of vertex indices of hyperedges in consecutive array locations. These locations are accessed via the hoffsets array. The hyperedge with weight hedgewts[i] has the indices of its vertices stored in locations pinlist[i] through to pinlist[hoffsets[i]-1].

`nparts` Integer specifying the number of parts in the partition, however, there are some constraints on `nparts` to bear in mind - these are considered in Section 3.1.3.

`constraint` Double-precision floating point value that specifies the balance constraint on the computed partition. The weight of the largest partition must not exceed $\lfloor(1+\epsilon)W_{avg}\rfloor$, where $\epsilon =$ `constraint`, assuming integer weights on the vertices.

`k-1cut` Integer that stores the cutsize value of the best computed partition. The cutsize is computed using the $k-1$ metric and the parameter is initialised on return from **ParaPartKway**

`options` Integer array of length 29 that sets the user defined options in **ParaPartKway**. The meaning of the individual entries in the array is described in Section 3.2.

`pvector` Integer array of length `nvertices` where the computed part values of the local vertices will be stored on return from partitioning routine.

`out_file` String representing the name of the file that program output (such as program information during execution) should be written to. If `out_file` = `NULL`, then program information will be written to **cout**.

`comm` The MPI communicator of the processes that call **ParaPartKway**.

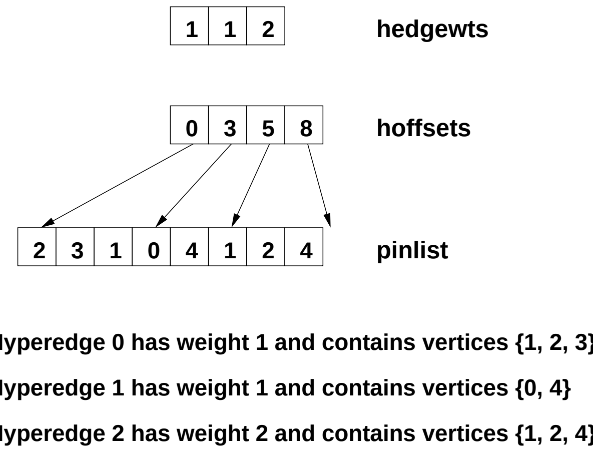
1 1 2
hedgewts
0 3 5 8
hoffsets
2 3 1 0 4 1 2 4
pinlist
Hyperedge 0 has weight 1 and contains vertices {1, 2, 3}
Hyperedge 1 has weight 1 and contains vertices {0, 4}
Hyperedge 2 has weight 2 and contains vertices {1, 2, 4}

Figure 2. Representing hyperedges in a pinlist

3.1.3 Remark on the Number of Processes vs. Size of Partition Sought

The `nparts` parameter in the partitioning routines from Sections 3.1.2 and 3.1.1 determines the size of the partition sought (i.e. k for a k-way partition). The range of values allowed depends on the number of parallel processes p and the partitioning algorithms used. Furthermore, certain algorithms can currently only be used when p is a power of two, although this restriction should be removed from future releases.

When the user selects the parallel V-Cycle algorithms (see 3.2.4), p the integer number of processes must be a power of two and integer k must satisfy $p = ak$, where $a \geq 1$ is also an integer. These conditions are sufficient when interfaces to khMeTiS and PaToH are used for the serial partitioning (see 3.2.3). If the serial algorithm chosen is the generic recursive bisection, then k must also be a power of two.

When the user selects algorithms without the parallel V-Cycle, p can take any integer value greater than one. When interfaces to khMeTiS and PaToH are used for the serial partitioning, k can also take any integer value greater than one. However, if the generic recursive bisection is used for serial partitioning, the value of k must be a power of two.

Note that, in general, the partitioning routines from the Par$k$way library should not be used to compute bipartitions unless it is not possible to compute the bipartition serially.

3.2 User-defined Options

The user is required to specify options with respect to the algorithms that will be used by the partitioning routine. This is done via the `options` array of integers, that is passed to the partitioning routine. The array contains 26 integers, whose meaning is interpreted as follows.

3.2.1 General Options

The following options (`options[0]` through to `options[5]` inclusive) determine general parameters such as the level of IO during the execution of the parallel routines.

`options[0]` Specifies whether default partitioning parameters are to be used. If `options[0]=0`, then default values are used. Otherwise, they need to be specified by the user.

`options[1]` Specifies the seed for the parallel pseudorandom number generator SPRNG [12], and represents an encoding of the starting state. If `options[1]=0`, then SPRNG will create its own seed using system date and time information. This is the default. Note that different pseudorandom sequences will be produced on different processes even if the same seed value is used on each.

`options[2]` Defines the amount of information that is to be displayed during the execution of the algorithm. The values are as follows:

- **0** There is no information displayed. Proceed 'silent'. This is the default.
- **1** Only display program options and cutsize values for each parallel run.
- **2** Display all information about the partitioning process.

`options[3]` Specifies whether the computed partition should be written to disk. When written to disk, the partition is written to the file `in_file.part.nparts`. For more information about the format of the output partition file, refer to Section 4. The values are as follows:

- **0** No output to file. This is the default.
- **1** Write the computed partition to disk.

`options[4]` Specifies the number of runs of the parallel partitioning algorithm. On return from **ParaPartKway**, `k-1cut` will be initialised to the cutsize corresponding to the cutsize of the partition with the smallest cutsize out of the `options[4]` partitions of the computed. Default value is `options[4]=1`.

`options[5]` Specifies whether **ParaPartKway** performs a random shuffle of vertices across the processors before commencing the coarsening phase. Can be useful in load balancing computation and communication, or to randomize the coarsening algorithm. Setting `options[5]=1` will turn the shuffling on while setting `options[5]=0` will turn it off, which is the default.

3.2.2 Parallel Coarsening Options

The following options (`options[6]` through to `options[13]` inclusive) determine the algorithm to be used during the parallel coarsening phase.

`options[6]`, `options[7]` These specify whether the coarsening computation is to be performed using all the hyperedges of the hypergraph or only those hyperedges whose length does not exceed a prescribed length. The value of `options[6]` determines the percentile of hyperedge weight l, such that `options[6]` percent of the total hyperedge weight in the hypergraph is found in hyperedges with length less than or equal to l. The value of `options[7]` determines the increment to `options[6]` for each subsequent coarsening step, provided that the required percentile does not exceed 100.

`options[8]` Specifies the maximum number of vertices in the coarsest hypergraph for the parallel coarsening algorithm. When the number of vertices in the coarsest hypergraph is less than `nparts`$\times$`options[8]`, the parallel coarsening phase terminates and the coarsest hypergraph is partitioned serially. Typically, `options[8]` takes values between 100 and 200. The default value is `options[8]=200`.

`options[9]`, `options[10]` These specify the reduction ratio during the parallel coarsening phase. The ratio r is given by $r =$`options[9]`/`options[10]`. During the parallel coarsening phase, each process computes the matching vector from its local vertices to the coarse vertices. When the ratio of the number of local vertices to the number of their coarse counterparts exceeds r, the process terminates local coarsening and proceeds to the communication stage to resolve matching conflicts with other processors. Typically, r takes values between 1.5 and 2.0. Default is $r = 1.75$.

`options[11]` Specifies the order in which a process visits its local vertices during the parallel coarsening phase. The values are as follows:

- **1** Increasing order (of vertex index).
- **2** Decreasing order (of vertex index).
- **3** Random Order (this is the default).
- **4** In order of increasing weight.
- **5** In order of decreasing weight.

`options[12]` Specifies how the connectivity metric between two vertices should be computed. Determines whether it should be inversely proportional to the weight of the resulting cluster and the lengths of connecting hyperedges. Values are as follows:

- **0** Cluster weight and hyperedge length not taken into account (default value).
- **1** Inversely proportional to resulting cluster weight only.
- **2** Inversely proportional to lengths of connecting hyperedges only.
- **3** Inversely proportional to both resulting cluster weight and connecting hyperedge lengths.

`options[13]` Specifies the order in which a process considers matching requests for its vertices from other procesors during the parallel coarsening phase. The values are as follows:

- **2** In order that they arrive.
- **3** Random order (this is the default).

3.2.3 Serial Partitioning Options

The following options (`options[14]` through to `options[21]` inclusive) determine the algorithm to be used during the serial partitioning phase. A generic serial recursive bisection algorithm implementation is provided, as well as interfaces to PaToH [4] and khMeTiS [10].

`options[14]` Number of serial partitioning runs. If `HMETIS_PartKway()` or `PaToH_Partition()` are used, each process will compute $\max($`options[14]`$/p, 1)$ runs, where p is the number of processes. Otherwise, all p processors will be involved in each of the `options[14]` recursive bisection runs.

`options[15]` Specifies the routine that will be used for serial partitioning. The values are as follows.

1 A generic recursive bisection method, without V-Cycles.

2 The generic recursive bisection method (as in `options[15] = 1`), but uses V-Cycle refinement for the final bisection. This is the default.

3 The generic recursive bisection method (as in `options[15] = 1`), but uses V-Cycle refinement during each multilevel bisection step.

4 Uses `HMETIS_PartKway()`. In order to use this, must link the par$k$way library with libhmetis.a (the hmetis library).

5 Uses `PaToH_Partition()`. In order to use this, must link the par$k$way library with libpatohv3-linux.a (the patoh library).

`options[16]` Serial coarsening algorithm. This option is only significant if `1`$\leq$`options[15]`$\leq$`3`. Values are as follows:

1 FC Coarsening, metric does not involve resulting cluster weight (default value).

2 FC Coarsening, metric does involve resulting cluster weight. Metric inversely proportional to resulting cluster weight.

`options[17]` Number of bisection runs during serial partitioning. This option is only significant if `1`$\leq$`options[15]`$\leq$`3`. If possible, the runs are carried out in parallel across available processors. Default value is `options[17]=2`.

`options[18]` Specifies the number of initial partitioning runs (i.e. on the coarsest hypergraph during serial partitioning). This argument is only significant if `1`$\leq$`options[15]`$\leq$`3`. Default value is `options[18]=10`.

`options[19]` Specifies the `HMETIS_PartKway()` coarsening option. Only significant if `options[15] = 4`. For more information about the following options, see [10]. The values are as follows.

1 Hybrid First-Choice scheme.

2 First-Choice scheme (default value).

3 Greedy First-Choice scheme.

4 Hyperedge scheme.

5 Edge scheme.

`options[20]` Specifies the `HMETIS_PartKway()` refinement option. Only significant if `options[15] = 4`. For more information about the folowing options, see [10]. Values are as follows.

0 No V-Cycle.

1 V-Cycle final.

2 V-Cycle best intermediate (default value).

3 V-Cycle each intermediate.

`options[21]` Specifies the `PaToH_Partition()` parameter settings. Only significant if `options[15] = 5`. For more information see [4]. Values are as follows.

1 Sets the values to PATOH_SUGPARAM_DEFAULT (default value).

2 Sets the values to PATOH_SUGPARAM_SPEED.

3 Sets the values to PATOH_SUGPARAM_QUALITY.

3.2.4 Parallel Uncoarsening Options

The following options (`options[22]` through to `options[27]` inclusive) determine the algorithm to be used during the parallel uncoarsening phase.

`options[22]` Specifies the parallel uncoarsening algorithm. Values are as follows.

1 Parallel greedy k-way refinement algorithm, no parallel V-Cycles used (default value).

2 Use parallel V-Cycles, but only iterate from the final partition (i.e. the partition of the original hypergraph).

3 Use parallel V-Cycles, iterating the best partition at each intermediate stage of the multilevel pipeline.

`options[23]` Specifies the limit on the number of parallel V-Cycle iterations and is applied at each stage of the multilevel pipeline in the case when `options[22]=3` and only at the original hypergraph when `options[22]=2`. Parameter only significant when `options[22]` is equal to 2 or 3. Default behaviour specifies `options[23]=MAX_INT`.

`options[24]` Specifies the minimum acceptable gain of a parallel V-Cycle iteration in order to call a further V-Cycle iteration. The value of `options[24]` is interpreted as the percentage of the cutsize of the partition before the first V-Cycle iteration was called at this stage of the multilevel pipeline. Only significant when parallel V-Cycles are used. Default behaviour `options[24]=0`.

`options[25]` Specifies the percentage threshold used to determine whether to accept or reject partitions during the parallel uncoarsening phase. As multiple partitions may be recorded for a particular coarse hypergraph during the multilevel pipeline, those partitions whose cutsizes are within `options[25]%` of the best partition cutsize will be projected onto the successive finer hypergraph. The threshold percentage is used for the coarsest hypergraph and subsequently reduced at each successive stage of the multilevel pipeline by a factor determined by `options[25]`. Default value is `options[25]=70`.

`options[26]` Specifies the reduction in the threshold defined in `options[25]`, applied at each successive uncoarsening step. The percentage threshold t_{i+1} for the subsequent uncoarsening step becomes `options[26]`$\times(t_i/100)$. Default is `options[26]=70`.

`options[27]` Defines the criterion for early exit from the parallel greedy k-way refinement. The criterion is the number of consecutive vertices that are visited, all of whose moves do not result in positive gain in the objective function. That is, the refinement algorithm will terminate locally on a processor if $(\texttt{options[27]}/100) \times numV$ consecutive moves with negative gain are seen, where $numV$ is the number of locally stored vertices. Thus, must have $0 <$`options[27]`≤ 100. Default is `options[27]=100`.

`options[28]` Specifies whether the refinement parallel computation is to be performed using all the hyperedges of the hypergraph or only those whose length does not exceed a prescribed length. When `options[28]=0`, then all hyperedges are used (default behaviour). Otherwise `options[28]=1` and the hyperedges used are determined by `options[6]` and `options[7]`, as for the parallel coarsening algorithm.

3.3 Memory Allocation

Par$k$way dynamically allocates the memory required for the computation. The user is not required to allocate any additional memory for the partitioning routines. The input arrays to **ParaPartKway**, as described in 3.1.2 are not modified by the routine, except for the array that stores the partition vector. Furthermore, the different partitioning algorithm implementations all require roughly the same amount of memory across the processors. If there is not enough memory on a particular machine, then **ParaPartKway** will abort.

4 Format of Input and Output to Par$k$way

4.1 Input File Format

When **ParaPartKway** is used to partition a hypergraph stored on disk, the following file format is used and following naming criteria should be adhered to. Given p processes, the hypergraph is stored across p input files. File to be read in by process i should have name `in_file-i`, where `in_file` is the name of the file passed as argument to **ParaPartKway**. The files should all have the same directory path (i.e. should be stored in the same directory). Each of the p files will be in the following binary format. The current version of Par$k$way assumes four-byte integer types.

The first 12 bytes are read in as three consecutive integers into a buffer where they are interpreted as follows.

`buffer[0]` The total number of vertices in the hypergraph.

`buffer[1]` The number of vertices for whom information is stored on the file (local vertices). The local vertices have contiguous index values. Process 0 is required to store vertex with index 0, so that the file `in_file-0` will contain information about the vertex with index 0.

`buffer[2]` Length of the portion of the file storing the hyperedges. It is in fact the number of integers in this portion of the file (which is thus `buffer[2]`$\times 4$ bytes long).

Each file `in_file-i` is required to store `buffer[0]`$/p$ vertices. File `in_file-j`, where $j = p - 1$, in addition also stores the remaining vertices, when p does not divide `buffer[0]` exactly. The next `buffer[1]`$\times 4$ bytes, after the 12-byte header, are occupied by the weights of the `buffer[1]` local vertices. Each weight is stored in consecutive 4 bytes and read in as a 4-byte integer value. The first 4 bytes store the weight of the local vertex with minimum index value, with consecutive 4-byte chunks storing the weight of subsequent consecutive vertices in terms of their index. The rest of the file stores hyperedges from the hypergraph. This is read in as an integer array by the program and interpreted as follows. The individual hyperedges are stored in the file in blocks. The first integer is the block-length (representing the number of integers in the hyperedge block, including itself). The next integer is the weight of the hyperedge. The remaining block-length minus two integers are the indices of the vertices belonging to this hyperedge. When a hypergraph is stored on file, it should be stored in such a way that each of the p files contains approximately the same number of hyperedges.

4.2 Output File Format

ParaPartKway may create up to two output files. The first is the binary file storing the partition of the hypergraph, as computed by **ParaPartKway** and the second is the text file containg the information output during the execution of **ParaPartKway**.

The binary file storing the partition computed by **ParaPartKway** has length $|V| \times 4$ bytes, where $|V|$ is the number of vertices in the hypergraph. Vertex v_i has part value stored in the ith four-byte chunk from the beginning of the file. The binary file will be written to the same directory that the hypergraph files are stored in.

The name of the text file (including the path if in another directory) containing program output needs to be supplied as an argument to the **ParaPartKway**. The file will be created by the program if it does not already exist. This argument is discussed in 3.1.1.

5 System Requirements and Contact Information

5.1 System Requirements

Par$k$way is written in C++, using the MPI library for interprocessor communication. It has been written for and tested on a Linux platform. The library was tested using gcc, versions 3.2.2-2.96 and MPICH, versions 1.2.4-1.2.5 on Mandrake Linux 9.1 and Red Hat Linux 7.2.

5.2 Installation Instructions

The following are the instructions for building the Par$k$way library. Information is also provided on the interfaces to `PaToH_Partition()` [4] and `HMETIS_PartKway()` [10], as well as the test code provided.

5.2.1 Building the Par$k$way Library

The Par$k$way library is built by typing `make` in the parkway-2.0 directory. The user may need to modify the file `Makefile.in` such that the appropriate compilation parameters are set. The compiler can be changed by modifying the `CC` variable and compiler options by modifying the `OPTS` variable. For some compilers, the path to the MPICH header file `mpi.h` needs to be provided (e.g. by using the `DIRS` variable). The user can also set the size of the hyperedge hash at compile time. In order to use a 32-bit hash key, the user needs to define `KEY_UINT` in the `Config.h file`. When `KEY_UINT` is not defined, a 64-bit hash key is used for each hyperedge.

5.2.2 Linking

In order to make use of the library, the user is required to link with a number of other libraries in addition to his/her code. The program `mpiCC` (from MPICH) will implicitly link Par$k$way with the MPI implementations from MPICH. In addition, the user also needs to link with the parallel pseudorandom number generator library SPRNG [12], whose routines are used for pseudorandom number generation in the Par$k$way library. In order to take advantage of the interface to partitioning routines from PaToH [4] and hMeTiS [10], the user is required to link with the PaToH and/or hMeTiS libraries. More information about these libraries and download can be found at `http://bmi.osu.edu/ umit/software.htm` and `http://www-users.cs.umn.edu/ karypis/metis/hmetis/` respectively. To link successfully with either `libhmetis.a` and/or `libpatoh-v3-linux.a` the user needs to modify the `Config.h` file before compilation. Define `LINK_HMETIS` if linking with `libhmetis.a` and define `LINK_PATOH` if linking with `libpatoh-v3-linux.a`. The user also needs to modify the file `Makefile.in` in the top-level directory, specifically the `LIBS` variable, so that the appropriate libraries are linked in. An example makefile is provided in the Test directory.

5.2.3 Testing

Sample code to test the library is provided in the directory Test. The file `driver.cpp` calls and tests the **ParaPartKway** partitioning routines from the Par$k$way library. The test code should be run as follows.

1. Distribute the hypergraph into p files (where p is the number of processes). Alternatively, use program `convert` from the Utilities directory to convert an existing hypergraph file.

2. Having compiled the library and the test code, run the program `parkway` with the `mpirun` command from MPICH

   ```
   % mpirun -np <p> -machinefile <m-file> parkway [parkway_options...]  <hgraph_file>
   ```

   p number of processes.

   m-file file specifying the processes to be used (see MPICH).

   parkway_options Are the options available for testing the library:

   `-oFile <filename>` Indicates that the user wants program output to be written to the file `filename` rather than to screen.

   `-nParts <number_parts>` Integer that indicates the number of parts that the hypergraph should be partitioned into (partition size). If not specified, parkway will seek to partition the hypergraph into a partition of size four.

   `-tType <test_type>` Integer that indicates which partitioning routine will be tested. When equal to 0, will run **ParaPartKway** with hypergraph read in from disk, which is also default behaviour if the option is not specified. Otherwise, will test **ParaPartKway** with the hypergraph stored in memory.

3. Test information will be recorded in a file called `test_output.p.txt`, where p is the number of processes.

5.2.4 The Utilities Directory

The `Utilities` directory contains the `convert` program and its source. This can be used to convert the files from text-based hypergraph formats to their binary counterparts, as used by Par$k$way. Having made `convert`, typing `./convert` at the command line should invoke the instructions on using `convert`.

5.2.5 Bug Report

Any bugs or problems encountered should be reported to `parkway@doc.ic.ac.uk`. A bug report should include a description of the problem and, if posible, include a copy of the hypergraph that was being partitioned. Once a bug is found, the user is encouraged to recompile the library with `DEBUG_ALL` defined in the `Config.h` file and to run the program again so that the bug may be identified more easily.

References

[1] C.J. Alpert, J.H. Huang, and A.B. Kahng. Recent Directions in Netlist Partitioning. *Integration, the VLSI Journal*, 19(1–2):1–81, 1995.

[2] C. Aykanat, A. Pinar, and U.V. Catalyurek. Permuting Sparse Rectangular Matrices Into Block-Diagonal Form. *SIAM Journal on Scientific Computing*, 25(6):1860–1879, 2004.

[3] J.T. Bradley, N.J. Dingle, W.J. Knottenbelt, and H.J. Wilson. Hypergraph-based Parallel Computation of Passage Time Densities in Large semi-Markov Models. In *Proc. 4th International Conference on the Numerical Solution of Markov Chains (NSMC'03)*, pages 99–120, Urbana-Champaign IL, USA, September 2nd–5th 2003.

[4] U.V. Catalyurek and C. Aykanat. Hypergraph-Partitioning-Based Decomposition for Parallel Sparse-Matrix Vector Multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, 1999.

[5] U.V. Catalyurek and C. Aykanat. A Fine-Grain Hypergraph Model for 2D Decomposition of Sparse Matrices. In *Proc. 8th International Workshop on Solving Irregularly Structured Problems in Parallel*, San Francisco, USA, April 2001.

[6] N.J. Dingle, W.J. Knottenbelt, and P.G. Harrison. Uniformization and Hypergraph Partitioning for the Distributed Computation of Response Time Densities in Very Large Markov Models. *Journal of Parallel and Distributed Computing*, 64(8):908–920, August 2004.

[7] C.M. Fiduccia and R.M. Mattheyses. A Linear Time Heuristic For Improving Network Partitions. In *Proc. 19th IEEE Design Automation Conference*, pages 175–181, 1982.

[8] G. Karypis. Multilevel Hypergraph Partitioning. Technical Report #02-25, University of Minnesota, 2002.

[9] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel Hypergraph Partitioning: Applications in VLSI Domain. In *Proc. 34th Annual ACM/IEEE Design Automation Conference*, pages 526–529, 1997.

[10] G. Karypis and V. Kumar. *hMeTiS: A Hypergraph Partitioning Package, Version 1.5.3*. University of Minnesota, 1998.

[11] G. Karypis and V. Kumar. Multilevel k-way Hypergraph Partitioning. Technical Report #98-036, University of Minnesota, 1998.

[12] M. Mascagni and A. Srinivasan. Algorithm 806: SPRNG: A Scalable Library for Pseudorandom Number Generation. *ACM Transactions on Mathematical Software*, (26):436–461, 2000.

[13] A. Trifunovic and W. Knottenbelt. Par$k$way2.0: A Parallel Multilevel Hypergraph Partitioning Tool. In *Proc. 19th International Symposium on Computer and Information Sciences*, volume 3280 of *Lecture Notes in Computer Science*, pages 789–800. Springer, 2004.

[14] A. Trifunovic and W.J. Knottenbelt. A Parallel Algorithm for Multilevel k-way Hypergraph Partitioning. In *Proc. 3rd International Symposium on Parallel and Distributed Computing*, University College Cork, Ireland, July 2004.

[15] A. Trifunovic and W.J. Knottenbelt. Towards a Parallel Disk Based Algorithm for Multilevel k-way Hypergraph Partitioning. In *Proc. 5th Workshop on Parallel and Distributed Scientific and Engineering Computing*, Santa Fe, NM, USA, April 2004.

[16] B. Ucar and C. Aykanat. Encapsulating Multiple Communication-Cost Metrics in Partitioning Sparse Rectangular Matrices for Parallel Matrix-Vector Multiples. *SIAM Journal on Scientific Computing*, 25(6):1837–1859, 2004.

[17] B. Vastenhouw and R.H. Bisseling. A two-Dimensional Data Distribution Method for Parallel Sparse Matrix-vector Multiplication. *SIAM Review*, 2004. Accepted for publication.