The `jostle` user manual: Version 2.2

Chris Walshaw

School of Computing & Mathematical Sciences,
University of Greenwich, London, SE10 9LS, UK
email: jostle@gre.ac.uk

March 28, 2000

Contents

1 The standalone `jostle` package

The public domain standalone `jostle` package comprises this userguide, `usrguide.ps`, one or more executables compiled for different machines (e.g. `jostle.sgi`) as requested on the licence agreement and an example mesh, `tri10k.graph` & `tri10k.coords`. Executables are available for most UNIX based machines with an ANSI C compiler and the authors are usually able to supply others.

The file `tri10k.coords` contains the x & y coordinates of the graph nodes but is not actually required for partitioning since `jostle` does not use geometrical information.

2 Running `jostle`

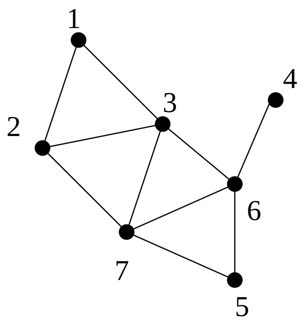
`jostle` (i.e. `jostle.sgi`, `jostle.linux`, etc.) is run with two inputs – the graph filename and the required number of partitions. These can either be entered on the command line, e.g.

```
jostle tri10k 32
```

or at the prompt. `jostle` will first try to open the filename given and if that fails it will try `[filename].graph`

3 Input & output

3.1 The input graph file



`jostle` uses the Chaco file input format, [1] (also used by Metis, [3]). In its simplest form the initial line of the file should contain the integers N, the number of nodes in the graph, and E, the number of edges. There should then follow N lines each containing the lists of neighbours for the corresponding node (with the nodes being numbered from 1 up to N). An example graph is shown above and the graph file is then

```
7 10
2 3
1 3 4
1 2 4 6
2 3 5 6
4 6
3 4 5 7
6
```

In more detail, there are 7 nodes and 10 edges in the graph; node 1 is adjacent to 2 & 3; node 2 is adjacent to 1, 3 & 4; etc.

The graph may also have weighted nodes and edges and the initial line should then contain a third integer, the format, to specify which. The possible formats are

format	node weights	edge weights
0	unitary	unitary
1	unitary	user specified
10	user specified	unitary
11	user specified	user specified

If node weights are required they should be specified as an integer, one on each of the N lines, prior to the neighbour lists. If edge weights are required they are specified after each neighbour (i.e. the weight for edge (n_1, n_2) is specified after n_2 on the line corresponding to n_1).

To give some examples, suppose that node 3 of the example graph has weight 12 and all the rest have weight 1, then the graph file would be specified as

```
7 10 10
1  2 3
1  1 3 4
12 1 2 4 6
1  2 3 5 6
1  4 6
1  3 4 5 7
1  6
```

Alternatively, suppose that edge (4,6) has weight 9, all the rest have weight 1 and the nodes are unweighted, then the graph file would be specified as

```
7 10 1
2 1 3 1
1 1 3 1 4 1
1 1 2 1 4 1 6 1
2 1 3 1 5 1 6 9
4 1 6 1
3 1 4 9 5 1 7 1
6 1
```

Note that the weight 9 occurs twice; once on the line corresponding to node 4 (the 4th line after the header) after the entry 6 and once on the line corresponding to node 6 (the 6th line after the header) after the entry 4.

Finally, suppose that all nodes and edges have unit weights, as in the first example, but we wish to explicitly record this in the file, then the graph file would be specified as

```
7 10 11
1 2 1 3 1
1 1 1 3 1 4 1
1 1 1 2 1 4 1 6 1
1 2 1 3 1 5 1 6 1
1 4 1 6 1
1 3 1 4 1 5 1 7 1
1 6 1
```

Note that the graph must be undirected so that if an edge (n_1, n_2) appears then the corresponding edge (n_2, n_1) must also appear, and if the edge weights are user specified, they must both have the same weight. Node and edge weights should be positive integers although nodes with zero weights are allowed. For example to use graphs with non-contiguous sets of nodes, the missing nodes can be treated as having zero weight and zero degree (i.e. not adjacent to any others). Finally the file may be headed by an arbitrary number of comment lines, (lines beginning with `%` or #) which will be ignored.

3.2 The number of subdomains

. . . should be an integer $2 \leq P << N$.

3.3 The output partition file

After partitioning a graph, the partition is written out into a file `[filename].ptn`, where `[filename]` is the name of the original graph input file. This file will contain N lines each containing one integer p, with $0 \leq p < P$, giving the resulting assignment of the corresponding node.

3.4 Repartitioning

To repartition a mesh, call `jostle` with the `-repartition` flag set, e.g.

```
jostle -repartition tri10k 16
```

The code will then open the corresponding `.ptn` file (described above) and read the existing partition. All of the existing subdomains should be non-empty and the `-repartition` flag should come before any customised settings (see below).

4 Customising the behaviour

`jostle` has a range of algorithms and modes of operations built in and it is easy to reset the default environment to tailor the performance to a users particular requirements.

4.1 Balance tolerance

As an example, `jostle` will try to create perfect load balance while optimising the partitions, but it is usually able to do a slightly better optimisation if a certain amount of imbalance tolerance is allowed. The balance factor is defined as $B = S_{\text{max}}/S_{\text{opt}}$ where S_{max} is the weight of the largest subdomain and S_{opt} is the optimum subdomain size given by $S_{\text{opt}} = \lceil G/P \rceil$ (where G is the total weight of nodes in the graph and P is the number of subdomains). The current default tolerance is $B = 1.03$ (or 3% imbalance). To reset this, to 1.05 say, call

```
jostle imbalance=5 tri10k 16
```

or

```
jostle "imbalance = 5" tri10k 16
```

Note that if there is whitespace in the setting then the quotation marks are required (so that `jostle` interprets it as a single argument). Alternatively, create a file called `defaults` in the same directory as the `jostle` executable containing the line

```
imbalance = 5
```

The command line settings have a higher priority than the `defaults` file. Thus if `jostle` is called with a command line setting, the `defaults` file is ignored.

Note that for various technical reasons `jostle` will not guarantee to give a partition which falls within this balance tolerance (particularly if the original graph has weighted nodes in which case it may be impossible).

4.2 Faster partitioning

By default `jostle` uses a Kernighan-Lin type refinement algorithm during the optimisation stage, as described in [5]. However this incurs a certain penalty on the run-time and to carry out slightly lower quality but faster partitioning it is possible to use a greedy refinement algorithm with the setting

```
refinement = greedy
```

again either in the `defaults` file or on the command line.

4.3 Dynamic (re)partitioning

Using `jostle` for dynamic repartitioning, for example on a series of adaptive meshes, can considerably ease the partitioning problem because it is a reasonable assumption that the initial partitions at each repartition may already be of a high quality. Recall first of all from §3.4 that to reuse the existing partition the flag `-repartition` must be used. One optimisation possibility then is to increase the coarsening/reduction threshold – the level at which graph coarsening ceases. This should have two main effects; the first is that it should speed up the partitioning and the second is that since coarsening gives a more global perspective to the partitioning, it should reduce globality of the repartitioning and hence reduce the amount of data that needs to be migrated at each repartition. Currently the code authors use a threshold of 20 nodes per processor which is set with

```
threshold = 20
```

However, this parameter should be tuned to suit particular applications.

A second possibility, which speeds up the coarsening and reduces the amount of data migration is to only allow nodes to match with local neighbours (rather than those in other subdomains), and this can be set with

```
matching = local
```

However, this option should only be used if the existing partition is of reasonably high quality.

For a really fast optimisation, without graph coarsening use

```
reduction = off
```

which should also result in a minimum of data migration. However, it may also result in a deterioration of partition quality, and this will be very dependent on the quality of both the initial partition and also how much the mesh changes at each remesh. Therefore, for a long series of meshes it may be worth calling `jostle` with default settings every 10 remeshes or so to return to a high quality partition.

Finally note that some results for different `jostle` configurations are given in [9]. The configuration JOSTLE-MS is the default behaviour if `jostle` is called without an existing partition. The settings to achieve similar behaviour as the other configurations are

Configuration	Setting
JOSTLE-D	`reduction = off`
JOSTLE-MD	`threshold = 20,matching = local`
JOSTLE-MS	–

5 Disconnected Graphs

Disconnected graphs (i.e. graphs that contain two or more components which are not connected by any edge) *can* adversely affect the partitioning problem by preventing the free flow of load between subdomains. In theory it is difficult to see why a disconnected graph would be used to represent a problem since the lack of connections between two parts of the domain implies that there are no data dependencies between the two parts and hence the problem could be split into two entirely disjoint problems. However, in practice disconnected graphs seem to occur reasonably frequently for various reasons and so facilities exist to deal with them in two ways.

5.1 Isolated nodes

A special case of a disconnected graph is one in which the disconnectivity arises solely because of one or more *isolated* nodes or nodes which are not connected to any other nodes. These are handled automatically by `jostle`. If desired they can be left out of the load-balancing by setting their weights to zero, but in either case, if not already partitioned, they are distributed to all subdomains on a cyclic basis.

5.2 Disconnected components

If the disconnectivity arises because of disconnected parts of the domain which are not isolated nodes, then `jostle` may detect that the graph is disconnected and abort with an error message or it may succeed in partitioning the graph but may not achieve a very good load-balance (the variation in behaviour depends on how much graph reduction is used). To check whether the graph is connected, use the graph checking facility (see §6). To partition a graph that is disconnected use the setting

```
connect = on
```

This finds all the components of the graph (ignoring isolated nodes which are dealt with separately) and connects them together with a chain of edges between nodes of minimal degree in each component. However, the user should be aware that (a) the process of connecting the graph adds to the partitioning time and (b) the additional edges are essentially arbitrary and may bear no relation to data dependencies in the mesh. With these in mind, therefore, it is much better for the user to connect the graph before calling `jostle` (using knowledge of the underlying mesh not available to `jostle`). Finally note that, although ideally these additional edges should be of zero weight, for complicated technical reasons this has not been implemented yet and so the additional edges have weight 1 (which may be included in the count of cut edges).

6 Troubleshooting

`jostle` has a facility for checking the input data to establish that the graph is correct and that the graph is connected. If `jostle` crashes or hangs, the first test to make, therefore, is to switch it on with the setting

```
check = on
```

The checking process takes a fair amount of time however, and once the call to `jostle` is set up correctly it should be avoided.

Note that, if after checking, the graph still causes errors it may be necessary to send the input to the authors of `jostle` for debugging. In this case, `jostle` should be called with the setting

```
write = input
```

`jostle` will then generate a subdomain file, `jostle.`*nprocs*`.sdm`, containing the input it has been given, and this data should be passed on to the `jostle` authors. It may also be helpful to send details of the problem either by email or by filling in a bug report form available from the JOSTLE home page

```
        http://www.gre.ac.uk/jostle
```

7 Timing `jostle`

The code contains its own timing routine. It is switched on once the main routine is called and includes the time for the code to construct its own graph but does not include the time to read the input files, check the validity of the input or write the output files. The timing routine used is `times` which gives cpu usage.

By default the output goes to `stderr` but this can be changed with the setting

```
stopwatch output = stdout
```

to switch it to `stdout`, or

```
stopwatch output = off
```

to switch it off entirely.

8 Memory requirements

The memory requirements of `jostle` are difficult to estimate exactly (because of the graph coarsening) but will depend on n (the total number of graph nodes) and e (the total number of graph edges). In general, if using graph coarsening, at each coarsening level n is approximately reduced by a factor of 1/2 and e is reduced by a factor of approximately 2/3. Thus the total storage required is approximately $2n + 3e$.

The memory requirement for each node is 3 pointers, 3 int's and 5 short's and for each edge is 2 pointers and 2 int's. On 32-bit architectures (where a pointer and an int requires 4 bytes and a short requires 2 bytes) this gives 36 bytes per node (strictly it's 34 but C structures are aligned on 4 byte segments) and 16 bytes per edge. On architectures which use 64 bit arithmetic, such as the Cray T3E, these requirements are doubled. Thus the storage requirements (in bytes) for `jostle` are approximately:

	32-bit	64-bit
graph coarsening on	$(72n + 48e)$	$(144n + 96e)$
graph coarsening off	$(36n + 16e)$	$(72n + 32e)$

9 Algorithmic details and further information

`jostle` uses a multilevel refinement and balancing strategy, [5], i.e. a series of increasingly coarser graphs are constructed, an initial partition calculated on the coarsest graph and the partition is then repeatedly interpolated onto the next coarsest graph and refined and balanced there. The refinement algorithm is a multiway version of the Kernighan-Lin iterative optimisation algorithm which incorporates a balancing flow, [5]. The balancing flow is calculated either with a diffusive type algorithm, [2] or with an intuitive asynchronous algorithm, [4]. `jostle` can be used to dynamically repartition a changing series of meshes both load-balancing and attempting to minimise the amount of data movement and hence redistribution costs. Sample recent results can be found in [5, 6, 9].

`jostle` also has a range of experimental algorithms and modes of operations built in such as mapping the partition onto heterogeneous communications networks, [7], and optimising subdomain aspect ratio (subdomain shape), [8]. Whilst these features are not described here, the authors are happy to collaborate with users to exploit such additional functionality.

Further information may be obtained from the JOSTLE home page:

`http://www.gre.ac.uk/jostle`

and a list of relevant papers may be found at

`http://www.gre.ac.uk/~c.walshaw/papers/`

Please let us know about any interesting results obtained by `jostle`, particularly any published work. Also mail any comments (favourable or otherwise), suggestions or bug reports to C.Walshaw@gre.ac.uk.

References

[1] B. Hendrickson and R. Leland. The Chaco User's Guide Version 2.0. Tech. Rep. SAND 94-2692, Sandia National Labs, Albuquerque, NM, 1994.

[2] Y. F. Hu, R. J. Blake, and D. R. Emerson. An optimal migration algorithm for dynamic load balancing. *Concurrency: Practice & Experience*, 10(6):467–483, 1998.

[3] G. Karypis and V. Kumar. Metis unstructured graph partitioning and sparse matrix ordering system version 2.0. Technical report, Dept. Comp. Sci., Univ. Minnesota, Minneapolis, MN 55455, 1995.

[4] J. Song. A partially asynchronous and iterative algorithm for distributed load balancing. *Parallel Comput.*, 20(6):853–868, 1994.

[5] C. Walshaw and M. Cross. Mesh Partitioning: a Multilevel Balancing and Refinement Algorithm. To appear in *SIAM J. Sci. Comput.* (originally published as Univ. Greenwich Tech. Rep. 98/IM/35), 1998.

[6] C. Walshaw and M. Cross. Parallel Optimisation Algorithms for Multilevel Mesh Partitioning. To appear in *Parallel Comput.* (originally published as Univ. Greenwich Tech. Rep. 99/IM/44), 1999.

[7] C. Walshaw and M. Cross. Multilevel Mesh Partitioning for Heterogeneous Communication Networks. Tech. Rep. 00/IM/57, Univ. Greenwich, London SE10 9LS, UK, March 2000.

[8] C. Walshaw, M. Cross, R. Diekmann, and F. Schlimbach. Multilevel Mesh Partitioning for Optimising Domain Shape. *Int. J. High Performance Comput. Appl.*, 13(4):334–353, 1999. (originally published as Univ. Greenwich Tech. Rep. 98/IM/38).

[9] C. Walshaw, M. Cross, and M. Everett. Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes. *J. Par. Dist. Comput.*, 47(2):102–108, 1997. (originally published as Univ. Greenwich Tech. Rep. 97/IM/20).