The parallel JOSTLE library user guide : Version 3.0

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

July 8, 2002

Contents

1 The parallel JOSTLE library package

The `pjostle` library package comprises this userguide and one or more library files compiled for different machines (e.g. `libjostle.sgi.a`) as requested on the licence agreement. It also includes a header file `jostle.h` containing C prototypes for all available jostle subroutines and `jmpilib.c`, an interface to the MPI communications library. Copies of the library are available for most UNIX based machines with an ANSI C compiler and the authors are usually able to supply others.

2 Calling `pjostle` from C

2.1 Compilation

The library should be linked to the calling program as for any other library. However, the `pjostle` interface to MPI, `jmpilib.c`, is not part of the library but is supplied as source code and must be compiled separately by the user. For example to link with `libjostle.sgi.a` and assuming that `MPIINCLUDE` and `MPILIB` are environment variables stating where the MPI include files and library are installed, the compilation should include:

```
cc -c -I$MPIINCLUDE jmpilib.c
cc -o myprog myprog.o jmpilib.o -ljostle.sgi -lm -L$MPILIB -lmpi
```

The `-lm` flag is necessary as `pjostle` uses the maths library.

2.2 Usage

The declarations for `pjostle` functions are

```
extern  void    jostle_env(char*);
extern  void    pjostle_init(int*,int*);
extern  void    pjostle(int*,int*,int*,int*,int*,int*,int*,int*,
                        int*,int*,int*,int*,int*,int*,int*,double*);
```

Before calling `pjostle` you must call `MPI_Init` (as for any other parallel MPI based program) and subsequently call

```
        pjostle_init(&nprocs, &pid);
```

where `nprocs` is the number of processes and `pid` is the number of the calling process ($0 \leq$ `pid` $<$ `nprocs`). `pjostle_init` should only be called once in a program and then `pjostle` may be called repeatedly. By default, `pjostle` uses the MPI communicator `MPI_COMM_WORLD` for group communications purposes, but to reset this call

```
        pjostle_comm(&comm);
```

where `comm` is a valid MPI communicator (of type `MPI_Comm`) for the required group of processes.

The call to `pjostle` is

```
        pjostle(&nnodes, &offset, &core, &halo,
         index, degree, node_wt, partition,
         &local_nedges, edges, edge_wt,
         &nparts, part_wt, &output_level,
         &dimension, coords);
```

The variable `nparts` specifies how many parts (subdomains) the graph is to be divided up into.

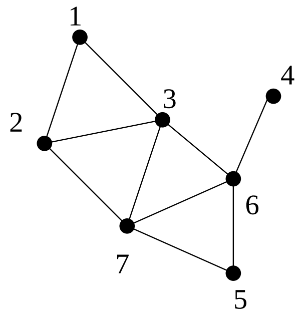
The array `part_wt` is only used for variable subdomain weights (see §5.2) and should be set to `(int*) NULL` if these are not being used.

The variable `output_level` reflects how much output the code gives and can be set to 0, 1, 2 or 3 (0 is recommended).

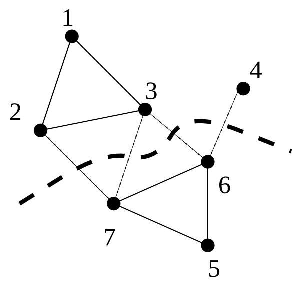
The variables `nnodes`, `offset`, `nparts`, `part_wt`, `output_level` and `dimension` should be set the same on every processor. Also currently `nparts` in the call to `pjostle` should be the same as `nprocs` in the call to `pjostle_init`.

2.3 Values of the graph variables

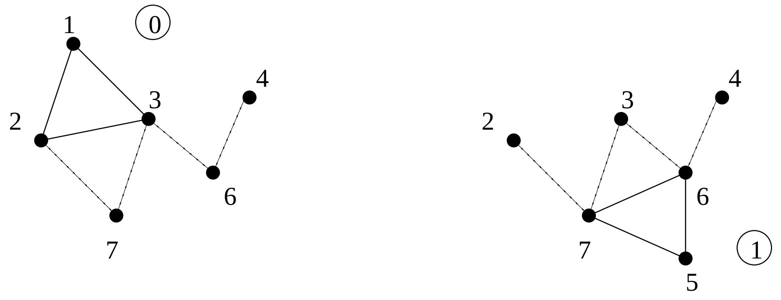
Consider an example graph



and suppose that it is partitioned into two pieces



so that the two subdomains look like



For each processor, nodes are referred to as core nodes if the processor owns them or halo nodes if not. Thus processor 0 has core nodes 1, 2, 3 & 4 and halo nodes 6 & 7, while processor 1 has core nodes 5, 6 & 7 and halo nodes 2, 3 & 4. Note that every node in the graph must be a core node of one and only one processor.

The values of the scalar quantities for processor 0 are then

`nnodes`	7	there are 7 nodes in the whole graph
`offset`	1	the index of the first node is 1
`core`	4	processor 0 owns 4 nodes (1,2,3,4)
`halo`	2	processor 0 has 2 halo nodes (6,7)
		(although `halo` should be set to 0 for the contiguous format – see below)
`local_nedges`	10	the sum of degrees of core nodes (i.e. the length of the edge array) is 10
`dimension`	0	not used – set to zero

Note that `offset` should be set to 0 or 1 depending on whether the node numbering starts at 0 (as is common in C programs) or 1 (as is common in FORTRAN programs).

There are then two possible formats to use when specifying the arrays to `pjostle`. The more general is the *node based format* where core & halo information (`index`, `partition`, `degree` and `node_wt`) is passed in for each node.

A special case is the *contiguous format* which may be used if the initial partition of the graph is strictly contiguous, i.e. nodes $1, \ldots, k_1$ on processor 0; nodes $k_1 + 1, \ldots, k_2$ on processor 1; $\ldots$; and nodes $k_{P-1} + 1, \ldots, N$ on processor $P - 1$. In this case the `index` array contains the number of `core` nodes in each subdomain, halo information does not need to be passed in for each node (in the arrays `partition`, `degree` and `node_wt`) and the variable `halo` should be set to 0.

For the example graph using the node-based format, the lengths of the arrays `index`, `partition`, `degree` and `node_wt` should be `core` + `halo`. Alternatively for the contiguous format the length of `index` should be `nparts` and the lengths of the arrays `partition`, `degree` and `node_wt` should be `core`. For either format the lengths of the arrays `edges` and, if used, `edge_wt` should be `local_nedges`.

For the node-based format the values of the arrays for processor 0 are

`index`	`[1,2,3,4,6,7]`	indices of core nodes followed by indices of halo nodes
`partition`	`[-,-,-,-,1,1]`	on entry, the last `halo` entries should contain the processor number of the halo nodes on exit, the first `core` entries contain the new processor number of the core nodes
`degree`	`[2,3,4,1,4,4]`	degree of the nodes listed in `index` – e.g. the degree of node 6 is 4 even though processor 0 only knows about 2 of the edges
`node_wt`	`(int*) NULL`	if all the nodes have weight 1; OR
`node_wt`	`[1,1,4,1,1,1]`	if, for example, node 3 has weight 4
`edges`	`[2,3,1,3,7,1,2,6,7,6]`	numbers representing the neighbours of each core node (i.e. 1 is adjacent to {2,3}; 2 is adjacent to {1,3,7}; 3 is adjacent to {1,2,6,7}; 4 is adjacent to {6})
`edge_wt`	`(int*) NULL`	if all the edges have weight 1; OR
`edge_wt`	`[1,1,1,3,1,1,3,1,1,1]`	if, for example, edge (2,3) has weight 3 (note that the weight is repeated – once in the position corresponding to node 2's edge with 3 and again in the position corresponding node 3's edge with 2)
`coords`	`(double*) NULL`	not used

For the contiguous format the values of the arrays for processor 0 are

`index`	`[4,3]`	there are 4 core nodes on processor 0 and 3 on processor 1
`partition`	`[-,-,-,-]`	on exit, the entries contain the new processor number of the core nodes
`degree`	`[2,3,4,1]`	degree of the core nodes
`node_wt`	`(int*) NULL`	if all the nodes have weight 1; OR
`node_wt`	`[1,1,4,1]`	if, for example, node 3 has weight 4
`edges`	`[2,3,1,3,7,1,2,6,7,6]`	numbers representing the neighbours of each core node (identical to node-based format – see above)
`edge_wt`	`(int*) NULL`	if all the edges have weight 1; OR
`edge_wt`	`[1,1,1,3,1,1,3,1,1,1]`	if, for example, edge (2,3) has weight 3 (identical to node-based format – see above)
`coords`	`(double*) NULL`	not used

The values of the scalar quantities for processor 1 are

```
nnodes         7
offset         1
core           3
halo           3
local_nedges  10
dimension      0
```

For the node-based format the values of the arrays for processor 1 are

```
index      [5,6,7,2,3,4]
partition  [-,-,-,0,0,0]
degree     [2,4,4,3,4,1]
node_wt    (int*) NULL
edges      [6,7,3,4,5,7,2,3,5,6]
edge_wt    (int*) NULL
coords     (double*) NULL
```

For the contiguous format the values of the arrays for processor 1 are

```
index      [4,3]
partition  [-,-,-]
degree     [2,4,4]
node_wt    (int*) NULL
edges      [6,7,3,4,5,7,2,3,5,6]
edge_wt    (int*) NULL
coords     (double*) NULL
```

The node based format is the default and if the contiguous format is used, each call to `pjostle` must be preceded by a call

```
        jostle_env("format = contiguous");
```

2.4 Non-contiguous input data

To use graphs with non-contiguous sets of nodes, the missing nodes are regarded as nodes of zero weight and zero degree (i.e. not adjacent to any others). They should be left out of the input data, but the variable `nnodes` should refer to the total number of nodes *including* those of zero weight. Thus the index, i, of any node must lie within the range $0 \leq i - \texttt{offset} < \texttt{nnodes}$.

2.5 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 principle 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.

2.5.1 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 `pjostle`. 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.

2.5.2 Disconnected components

If the disconnectivity arises because of disconnected parts of the domain which are not isolated nodes, then `pjostle` 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 §4.1). To partition a graph that is disconnected use the setting

```
        jostle_env("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 `pjostle` (using knowledge of the underlying mesh not available to `pjostle`). 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).

3 Customising the behaviour

`pjostle` 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.

3.1 Balance tolerance

As an example, `pjostle` 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_env("imbalance = 5");
```

before a call to `pjostle`. This call will only affect the following call to `pjostle` and any subsequent calls will not be affected.

Note that for various technical reasons `pjostle` 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).

3.2 Dynamic (re)partitioning

Using `pjostle` 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. 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 (e.g. see [3]). Currently the code authors use a threshold of 20 nodes per processor which is set with

```
jostle_env("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

```
jostle_env("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

```
jostle_env("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 `pjostle` with default settings every 10 remeshes or so to return to a high quality partition.

Finally note that some results for different `pjostle` configurations are given in [8]. The configuration JOSTLE-MS is not strictly available with `pjostle` as this uses serial optimisation. However calling `pjostle` with the default behaviour is the closest parallel version. 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	–

4 Additional functionality

4.1 Troubleshooting

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

```
        jostle_env("check = on");
```

or

```
        jostle_env("check = limited");
```

Note that full checking requires $O(N)$ memory (where N is the number of nodes in the graph) and so if this is not possible limited checking should be used. The checking process takes a fair amount of time however, and once the call to `pjostle` is set up correctly it should be avoided.

If the graph is incorrect or inconsistent, `pjostle` will abort with an error message. Unfortunately if several processes abort at once their error messages all get written to `stderr` and may all appear at once giving confusing output. However, the error messages are also written to the files `pjostle.`*pid*`.log`, where *pid* is the identification number of the process.

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

```
        jostle_env("write = input");
```

Each process *pid* will then generate a subdomain file, `pjostle.`*nparts*`.`*pid*`.sdm`, containing the input it has been given, and this data should be passed on to the `pjostle` authors.

4.2 Timing `pjostle`

The code contains its own internal stopwatch which can be used to time the length of a run. It calls a barrier synchronisation immediately before the start and end of timing. The timing routine used is `MPI_Wtime` which is elapsed (or wall clock) time. Note that for optimal timings the graph checking (§4.1) should not be switched on.

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

```
        jostle_env("timer = stdout");
```

to switch it to `stdout`, or

```
        jostle_env("timer = off");
```

to switch it off entirely. Note that switching it off also switches off the barrier synchronisations and should be employed when using external timers around the call to `pjostle`.

4.3 Memory considerations

4.3.1 Memory requirements

The memory requirements of `pjostle` are difficult to estimate exactly (because of the graph coarsening and halo sizes) 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 $2N + 3E$.

For `pjostle` this total can be divided by P – the number of processors, but must be increased to allow for the sizes of the halos and the fact that the internal graph may become slightly unbalanced during the load-balancing. A factor of 2 is perhaps excessive for large grained problems but gives a good safety margin. Thus the storage requirements are approximately $(4N + 6E)/P$ if using graph coarsening and $(2N + 2E)/P$ if not.

The memory requirement for each node is 3 pointers, 3 int's and 6 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 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 `pjostle` are approximately:

	32-bit	64-bit
graph coarsening on	$(144N + 96E)/P$	$(288N + 192E)/P$
graph coarsening off	$(72N + 32E)/P$	$(144N + 64E)/P$

4.3.2 Reusing application workspace

To save memory, `pjostle` has a facility for using a workspace allocated externally by the application (whether permanently allocated in FORTRAN as an array declaration, e.g. `integer workspace(10000)`, or dynamically allocated in C using `malloc`). To make use of this facility, call

```
jostle_wrkspc_input(&length, workspace);
```

before every call to `pjostle` where `workspace` is the address of the workspace (cast as a `char*`) and `length` is its length in **bytes**. Note that the workspace need not be a type `char` array and only needs to be cast this way to fit with the function prototypes in `jostle.h`. Should this workspace prove insufficient for its memory requirements, `pjostle` will use as much of it as possible until it runs out and subsequently allocate additional memory using `malloc`.

4.4 Statistics functions

A number of subroutines are available to return statistics about the partition calculated by `pjostle` and about the partitioning process. They are as follows:

```
int     jostle_cut();
```

The `int` function `jostle_cut` returns the total weight of cut edges.

```
double  jostle_bal();
```

The `double` function `jostle_bal` returns the balance expressed as a ratio of the maximum subdomain weight over the optimal subdomain weight (as defined in Section 3.1), i.e. a return value of 1.00 is perfect balance.

```
double  jostle_tim();
```

The `double` function `jostle_tim` returns the run time in seconds of `pjostle`.

```
int     jostle_mem();
```

The `int` function `jostle_mem` returns the memory used by `pjostle` (in bytes). Note that this is the memory used on the processor from which the call is made, not a global maximum or sum of usage.

4.5 Calling `pjostle` from FORTRAN

Almost everything described in this document will work the same way when calling `pjostle` from FORTRAN rather than C. Thus the call to `pjostle` is

```
        call pjostle(nnodes, offset, core, halo,
    +    index, degree, node_wt, partition,
    +    local_nedges, edges, edge_wt,
    +    network, part_wt, output_level,
    +    dimension, coords)
```

where `nnodes`, `offset`, `core`, `halo`, `local_nedges`, `output_level` & `dimension` are `integer` variables, `index`, `degree`, `node_wt`, `partition`, `edges`, `edge_wt`, `network` & `part_wt` are `integer` arrays and `coords` is a `double precision` array. Note that for arrays which can be set to `NULL` when called from C (e.g. if all the nodes have weight 1, the array `node_wt` can be set to `NULL`), the same effect can be achieved by passing in a single variable set to -1). Also since `coords` is never used it can be a single variable rather than an array. In other words, if `node_wt`, `edge_wt` & `part_wt` are not being used, the following piece of code will work:

```
        double precision dummy_coords
        integer idummy, dimension
        idummy = -1
        dimension = 0
        call pjostle_init(nprocs, pid)
        call pjostle(nnodes, offset, core, halo,
    +    index, degree, idummy, partition,
    +    local_nedges, edges, idummy,
    +    network, idummy, output_level,
    +    dimension, dummy_coords)
```

To use the `jostle_env` calls, just replace the double quotation marks in the string with single ones, e.g.:

```
        call jostle_env('check = on')
```

5 Advanced/experimental features

5.1 Heterogeneous processor networks

`pjostle` can be used to map graphs onto heterogeneous processor networks in two ways (which may also be combined). Firstly, if the processors have different speeds, `pjostle` can give variable vertex weightings to different subdomains by using processor weights – see §5.2 for details.

For heterogeneous communications links (e.g. such as SMP clusters consisting of multiprocessor compute nodes with very fast intra-node cmmmunications but relatively slow inter-node networks) a weighted complete graph representing the communications network can be passed to `pjostle`. For an arbitrary network of P processors numbered from $0, \ldots, P-1$, let $l_{p:q}$ be the relative cost of a communication between processor p and processor q. It is assumed that these cost are symmetric (i.e. $l_{p:q} = l_{q:p}$) and that the cost expresses, in some averaged sense, both the latency and bandwidth penalties of such a communication. For example, for a cluster of compute nodes $l_{p:q}$ might be set to 1 for all intra-node communications and 10 for all inter-node communications.

To pass the information into `pjostle` the `nparts` variable is replaced with an array, `network` say (since all scalar variables are passed in as pointers this causes no problems). The `network` array should then be set to

$$\begin{array}{lcllll}
\texttt{network}[0] & = & -1 & & & \\
\texttt{network}[1] & = & P & & & \\
\texttt{network}[2, \ldots, P(P-1)/2+1] & = & l_{0:1}, & l_{0:2}, & \ldots, & l_{0:P-1}, \\
 & & & l_{1:2}, & \ldots, & l_{1:P-1}, \\
 & & & & \ldots, & \\
 & & & & & l_{P-2:P-1}
\end{array}$$

The -1 signifies that it is an arbitrary network and P then gives the number of processors. The following $P(P-1)/2$ entries are the upper triangular part of the network cost matrix (e.g. see [6, Fig. 2]).

The choice of the network cost matrix coefficients is not straightforward and is discussed in [6].

This functionality is available in parallel but `pjostle` actually solves the mapping/partitioning problem in serial (invisibly from the user).

5.2 Variable subdomain weights

It is sometimes useful to partition the graph into differently weighted parts and this is done by giving the required subdomains an additional fixed weight which is taken into account when balancing. For example suppose `pjostle` is being used to balance a graph of 60 nodes in 3 subdomains. If subdomain 1 were given an additional fixed weight of 10 say and subdomain 2 were given an additional fixed weight of 20, then the total weight is 90 ($= 60 + 10 + 20$) and so `pjostle` would attempt to give a weight of 30 to each subdomain and thus 30 nodes to subdomain 0, 20 nodes to subdomain 1 and 10 nodes to subdomain 2.

These weights can be specified to `pjostle` using the `part_wt` argument. Thus in the example above `pjostle` should be called with `part_wt` set to `[0,10,20]`. Note that this array should be the same on every processor.

Often it is more useful to think about the additional weights as a proportion of the total and in this case a simple formula can be used. For example, suppose a partition is required where Q of the P subdomains have f times the optimal subdomain weight S_{opt} (where $0 \leq f \leq 1$). Suppose that the total weight of the graph is W so that the optimal subdomain weight without any additional fixed weight is $S_{\text{opt}} = W/P$. Now let W' represent the new total graph weight (including the additional fixed weights) and let S'_{opt} represent the new optimal subdomain weight. The additional fixed weight must be $(1-f) \times S'_{\text{opt}}$ in each of the Q subdomains and so S'_{opt} can be calculated from:

$$S'_{\text{opt}} = \frac{W'}{P} = \frac{W + Q(1-f)S'_{\text{opt}}}{P}$$

and hence

$$S'_{\text{opt}} = \frac{W}{P - Q(1-f)}$$

Thus the additional fixed weight on each of the Q subdomains should be set to

$$(1-f) \times S'_{\text{opt}} = (1-f) \times \frac{W}{P - Q(1-f)}$$

Thus if, say, $P = 5$, $Q = 3$, $W = 900$ and $f = 1/3$ (i.e. three of the five subdomains have one third the weight of the other two) then

$$S'_{\text{opt}} = \frac{W}{P - Q(1-f)} = \frac{900}{5 - 3(2/3)} = 300$$

and so the additional fixed weight is

$$(1-f) \times S'_{\text{opt}} = 2/3 \times 300 = 200$$

and the `part_wt` array would be set to `[0,0,200,200,200]`.

5.3 Empty subdomains

It is possible for one or more processors to start the partitioning process empty (i.e. owning no nodes) and, assuming that they have not been designated as idle (as above §5.2) and that the balancing succeeds, then a fair share of the load should be distributed to them by `pjostle`. Typically the subdomains will be seeded with a single node at the top of the graph contraction process and then the optimisation should ensure load-balance. In this way `pjostle` can even be used before the graph has been distributed (i.e. if one processor owns the entire graph).

Alternatively, if all but one of the subdomains are empty, a more efficient usage of memory is for `pjostle` to distribute the data (using a contiguous block based distribution) before starting partitioning. In this case all processors must call

```
        jostle_env("scatter = on");
```

before calling `pjostle` and processor 0 should pass in the entire graph (using the node based format described in section 2.3, although with the `index` array set to `(int*) NULL`) whilst all the other processors should have `core`, `halo` and `local_nedges` set to zero.

6 Algorithmic details and further information

`pjostle` uses a multilevel refinement and balancing strategy, [4], i.e. a series of increasingly coarser graphs are constructed, an initial partition calculated on the coarsest graph and the partition is then repeatedly extended to the next coarsest graph and refined and balanced there. The refinement algorithm is a parallel iterative optimisation algorithm which uses the concept of *relative gain* and which incorporates a balancing flow, [5]. The balancing flow is calculated either with a diffusive type algorithm, [1] or with an intuitive asynchronous algorithm, [2]. `pjostle` 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 [4, 5, 8].

The modifications required to map graphs onto heterogeneous communications networks (see §5.1) are described in [6].

`pjostle` also has a range of experimental algorithms and modes of operations built in such as optimising subdomain aspect ratio (subdomain shape), [7]. 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 `pjostle`, particularly any published work. Also mail any comments (favourable or otherwise), suggestions or bug reports to jostle@gre.ac.uk.

References

[1] 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.

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

[3] C. Walshaw and M. Cross. Load-balancing for parallel adaptive unstructured meshes. In M. Cross *et al.*, editor, *Proc. Numerical Grid Generation in Computational Field Simulations*, pages 781–790. ISGG, Mississippi, 1998.

[4] C. Walshaw and M. Cross. Mesh Partitioning: a Multilevel Balancing and Refinement Algorithm. *SIAM J. Sci. Comput.*, 22(1):63–80, 2000. (originally published as Univ. Greenwich Tech. Rep. 98/IM/35).

[5] C. Walshaw and M. Cross. Parallel Optimisation Algorithms for Multilevel Mesh Partitioning. *Parallel Comput.*, 26(12):1635–1660, 2000. (originally published as Univ. Greenwich Tech. Rep. 99/IM/44).

[6] C. Walshaw and M. Cross. Multilevel Mesh Partitioning for Heterogeneous Communication Networks. *Future Generation Comput. Syst.*, 17(5):601–623, 2001. (originally published as Univ. Greenwich Tech. Rep. 00/IM/57).

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

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