PAMPAC user manual

D.A. Aruliah, Lennaert van Veen, Alex Dubitski

1 Overview

PAMPAC (Parallel Adaptive Method for Pseudo-Arclength Continuation) is a C library using MPI (Message-Passing Interface) routines that allows users to adapt serial codes for pseudo-arclength continuation to achieve a modest amount of concurrent processing. Specifically, the library permits scientific researchers to extend their continuation codes with a parallel strategy for concurrent computation of corrector steps in a predictor-corrector framework. The details of the algorithm are provided in the article [1] by the authors.

The user supplies a few basic routines with interfaces explained:

- a function `main` to act as a driver for the other routines;
- a function `compute_residual` for evaluating the nonlinear residual of the system of nonlinear equations at a given point;
- a function `single_corrector_step` that, given a putative point on the continuation curve and a tangent direction, computes an update (e.g., via Newton's method or some similar strategy) to obtain an improved point;
- a function `write_coordinates` for writing output as a user chooses; and
- a `main` function as typical for a C program to act as a driver.

In addition, the user must supply a text file storing the numerical values corresponding to an initial point on the continuation curve and a text file with parameters to control the core PAMPAC algorithm. An example application (that solves a modified Kuramoto-Sivashinsky equation) is provided to serve as a template from which users can build their own applications.

2 System requirements

PAMPAC has been developed using Linux/Unix systems with the following configuration:

- a ISO/IEC 9899:1999-compliant C compiler (`gcc` 4.8.2 on our systems);
- an implementation of the Message Passing Interface (MPI-2, `openmpi` 1.6.5 on our systems);

- the GNU Scientific Library (GSL 1.16 on our systems); and
- the Automatically Tuned Linear Algebra Software (ATLAS 3.10.1 on our systems).

In the core PAMPAC library, the GSL is used only as an interface to BLAS routines (Basic Linear Algebra Subroutines). Users are welcome to link to BLAS libraries tuned for their hardware when building the PAMPAC library instead. The example application provided uses the GSL for computing FFTs (Fast Fourier Transforms) and uses ATLAS as an interface for linear algebra solvers packaged in LAPACK libraries. The dependence on ATLAS is strictly for the example application (and not for the PAMPAC library itself).

3 Installation

After downloading and unpacking the source code, the `INSTALL` file in the top-level directory provides the key instructions. Briefly, they are as follows.

1. Customize the file `Make.config` in the top-level directory to suit your system.
2. `make lib` in the top-level directory to build the library.
3. Customize the file `driver.config` in the `example` directory to suit your system.
4. `make driver` in the `example` directory to build the driver executable.
5. Edit the file `parameters.txt` in the `example` directory as suitable for your run.
6. `make run` in the `example` directory to start a run. You will want to tune the number of processors in the call to `mpiexec` to reflect your system.

The main PAMPAC library comes with template `Makefiles` for easy building; some configuration of the files `Make.config` and `example/driver.config` is required to ensure that all library dependencies are met on your system. The build should work on any POSIX system; it has been tested on a cluster with QDR Infiniband interconnects and 2.2GHz AMD Opteron processors as well as desktop machines with two quad-core Intel Xeon X5482 processors.

More details are provided within the `INSTALL` file.

4 Structure of the code

The PAMPAC library is written in C (ISO/IEC 9899:1999) using MPI for parallelization. The PAMPAC library is modular in its design to aid in debugging and understanding its structure; each of the core tasks is performed by a separate function, located in the `/src` directory. Many of the core routines in the

PAMPAC library require traversal of the rooted tree in a depth-first (using recursion) or a breadth-first (using a queue) fashion. The node and queue data structures for managing the tree are documented in `pampac.h` in the `src` subdirectory. This file also describes a data structure for storing and communicating the parameter options parsed from the user's parameter file. The PAMPAC library is designed so that users need not know the details of the implementations of these data structures (nor the routines for allocation/deallocation of memory, management of pointers, etc.). The user need only specify the depth of the underlying tree and the related tunable parameters that control the parallel algorithm.

The `main` function performs two primary tasks: it initializes and finalizes MPI communication and it divides work between the master and the slave processes (this is typical in the Single-Program-Multiple-Processor (SPMP) paradigm). Within the `main` function, the master processor parses the user-provided parameter file to determine the algorithm-tuning parameters; if it does so successfully, the master process broadcasts the number of degrees of freedom (`N_DIM`) to the other processes and initiates the principal algorithm by invoking the routine `master_process`. After some preprocessing, such as loading the initial point from the user's input file and computing an initial tangent direction, the master process initiates the parallel algorithm. The slave processes all call the function `slave_process` in which they idle until receiving data from the master process—the data being a point `z` and some tangent direction `T` from which a corrector iteration can be computed. The only interprocess communication during the continuation loop consists of the root process sending these data to the slaves and each slave returning the result of a corrector step to the root process. The routines `master_process` and `slave_process` package the core components of the PAMPAC algorithm in a manner that alleviates the burden of managing the parallel computation from the user. The corrector iterations are independent and should be much more expensive than the cost of inter-process communication in order for PAMPAC to yield a speed-up.

User-supplied functions

To use the PAMPAC library, the user needs to supply functions with the following signatures:

`int main (int argc, char *argv[])` or alternatively,
`void main (int argc, char *argv[])`

```
int computer_residual (int N_dim, double *z, double *res)
```

```
int single_corrector_step (int N_dim, double *z, double *T)
```

```
write_coordinates (int N_dim, double *z)
```

The function `main` is a standard entry point that can accept command-line arguments. The function `compute_residual` evaluates the residual of the nonlinear function that determines whether points lie on the continuation curve.

The input parameter `z` is an array (vector) of `N_dim` double precision values; the computed residual `res` is an array of `N_dim-1` double precision values. The function `single_corrector_step` is a routine to compute the updated corrector iterate using the input point `z` and the tangent vector `T`. The corrected value of `z` overwrites the array `T` on exit from this function. Finally, the function `write_coordinates` is a user-tuned output routine for recording points computed on the continuation curve. These user-supplied functions need to be compiled with `main.c`—and any user-required dependencies—to produce an executable that can be run in parallel on numerous processors. Assuming that the user's `Makefile` is suitably configured, the user can link the executable with external library functions required by their routines.

Notice that, relative to the mathematical description of the template continuation problem in [1], `N_dim`$= n + 1$, i.e., `N_dim` refers to the dimension of the vector $\mathbf{z} = (\mathbf{x}, \lambda)$ rather than the dimension of the vector $\mathbf{x}$. Thus, in `compute_residual`, the "input" values are the (integer) dimension `N_dim` of the problem and the `N_dim`-vector pointed to by the pointer `z`; the "output" is the residual vector of the nonlinear problem, stored in an array of length `N_dim-1` in memory pointed to by the pointer `res`. Similarly, in `single_corrector_step`, the "input" values are the (integer) dimension `N_dim` of the problem, the `N_dim`-vector pointed to by the pointer `z`, and the `N_dim`-vector pointed to by the pointer `T` (corresponding to $\mathbf{T}$). After calling `single_corrector_step`, the array pointed to by `z` has been overwritten with the updated corrector iterate.

The parameter file

Parameters controlling the parallel continuation algorithm are loaded from a plain text file at run-time by the master processor. A template is provided in the file `example/parameters.txt`. Notice in the template `example/main.c` file, the name of the parameter file is read as a command-line argument and passed into the function `parse_parameters` directly.

The parameters in the parameter file are all denoted by the string `@param@` preceding the parameter's name and value. The parameter identifiers are as follows:

- `N_DIM`: the number of unknowns/degrees of freedom $n + 1$;
- `LAMBDA_MIN` and `LAMBDA_MAX`: bounds on the interval $[\lambda_{\min}, \lambda_{\max}]$ in which the continuation parameter λ lies;
- `LAMBDA_INDEX`: integer between `0` and `N_DIM-1` that is the index of the parameter λ in any `N_DIM`-vector;
- `DELTA_LAMBDA`: parameter for initial corrector iterations to generate a second point on the curve from the first (required to bootstrap the algorithm);
- `H_MIN` and `H_MAX`: the minimal, maximal and initial pseudo-arclength stepsize;

- `H_INIT`: the initial step-length used to determine the initial secant direction on the continuation curve. Notice that if `H_INIT` < 0, the initial secant direction points backwards with respect to the parameter λ.
- `MAX_ITER`: the maximum number of corrector steps before a node is designated as `FAILED`;
- `TOL_RESIDUAL`: the threshold for accepting `CONVERGED` nodes (*i.e.*, when $\left\|\mathbf{r}_\alpha^{(\nu_\alpha)}\right\|_2 \leq$ `TOL_RESIDUAL`);
- `MU`: the threshold reduction in residual for `FAILED` nodes (*i.e.*, when $\left\|\mathbf{r}_\alpha^{(\nu_\alpha)}\right\|_2 >$ `MU` $\left\|\mathbf{r}_\alpha^{(\nu_\alpha-1)}\right\|_2$);
- `GAMMA`: the threshold rate of residual reduction for `CONVERGING` nodes (*i.e.*, when `GAMMA` $\log \left\|\mathbf{r}_\alpha^{(\nu_\alpha)}\right\|_2 \leq \log$ `TOL_RESIDUAL`;
- `MAX_DEPTH`: the maximum depth of the tree, D (excluding the root node);
- `MAX_GLOBAL_ITER`: the maximum number of global PAMPAC iterations before continuation is halted;
- `SCALE_FACTOR`: a real,positive factor $t_{\mathtt{K}}$ by which step-sizes are multiplied when a new predictor step is spawned from a given point;
- `VERBOSE`: an integer parameter controlling the verbosity of output;
- `INPUT_FILENAME`: a string giving the path to the input file from which the initial point is read; and
- `TREE_BASE_FILENAME`: a string giving the path and the base filename that can be used if visualizations of the rooted trees at each stage of the algorithm are to be generated by the program.

Most of the parameters are self-explanatory. For instance, the parameters `MAX_ITER` and `TOL_RESIDUAL` are used to assess convergence of corrector iterations and to circumvent stagnating corrector loops respectively. In the descriptions above, the quantity $\mathbf{r}_\alpha^{(\nu_\alpha)}$ is the nonlinear residual (*i.e.*, the vector yielded by evaluating the function `compute_residual`) by node α when the local iteration counter is ν_α.

The parameter `GAMMA` is used to classify corrector iterations as `CONVERGING` according to the relation $\left\|\mathbf{r}_\alpha^{(\nu_\alpha)}\right\|_2^{\mathtt{GAMMA}} \leq$ `TOL_RESIDUAL` (*i.e.*, the *next iterate* computed from that iterate is expected to have converged onto the continuation curve). The parameter `MU` is used to classify corrector iterations as `FAILED` according to whether $\nu_\alpha >$ `MAX_ITER` or $\left\|\mathbf{r}_\alpha^{(\nu_\alpha)}\right\|_2 >$ `MU` $\left\|\mathbf{r}_\alpha^{(\nu_\alpha-1)}\right\|_2$ (*i.e.*, either the maximum number of corrector iterations is exceeded or the reduction of the residual is insufficient in consecutive corrector iterations). Tuning these parameters determines how the PAMPAC algorithm will preserve or destroy prospective corrector iteration sequences by deciding which iterates are deemed to be making sufficient progress to keep.

The parameter `LAMBDA_INDEX` provides additional flexibility by permitting the user to specify any integer index of $\mathbf{z}$—using 0-based indexing as is conventional in C—for the continuation parameter. That is, the parameter λ does not need to be the $(n+1)^{\text{st}}$ component of the $(n+1)$-vector $\mathbf{z}$.

The user specifies the depth of the tree in the parameter `MAX_DEPTH`. By contrast, the maximum number of children any node can spawn is given by the *number of lines containing the keyword*`SCALE_FACTOR`. That is, rather than specifying the width of the tree—that is, the number of children each node can potentially spawn—in the parameter, the user specifies a sequence of lines with the keyword `SCALE_FACTOR`. The numerical value in each of these lines is the multiplicative factor t_k that the PAMPAC algorithm uses to scale the pseudo-arclength step-size in generating predictor steps to seed concurrent corrector iterations.

To bootstrap the algorithm, the master processor requires an initial tangent direction in addition to the initial point loaded from the user's input file. It generates an approximate tangent direction by carrying out a few corrector iterations to generate another point near the initial point and computing a secant direction between those two points. At any given point on the continuation curve, there are two anti-parallel tangent directions; as such, the sign of `H_INIT` is used to fix the initial direction of the continuation (*i.e.*, the tangent direction used to generate the second point on the curve is oriented in the direction of λ increasing or decreasing when `H_INIT`> 0 or `H_INIT`< 0, respectively). The user also needs to specify `DELTA_LAMBDA` (roughly how far from the initial point to look for the neighboring point) to control this bootstrapping process.

The user can specify an integer parameter `VERBOSE` to control output generated at run-time. No output is generated unless the parameter `VERBOSE` is positive; With `VERBOSE>=1`, the master process displays diagnostic messages to standard output as the algorithm progresses. When `VERBOSE>=2`, the master process also creates data files in a user-specified path that display the structure of the rooted trees. The data files generated are compatible with the `dot` language for specifying directed graphs with the Graphviz software for visualization of graphs (see `www.graphviz.org`). Such graphs are useful for performance-tuning, *i.e.*, for understanding how the data in `parameters.txt` affect the use of processors.

5 Determining the number of processors

The user specifies the number of available CPUs using the `-np` flag in the call to `mpiexec` when running an executable compiled against the PAMPAC library. For a given tree of width W and depth D (D is measured excluding the root node), the maximal number of nodes—including the root node—is

$$np_{\max} = \begin{cases} D+1 & \text{if } W = 1 \\ \frac{W^{D+1}-1}{W-1} & \text{if } W > 1 \end{cases}$$

If the user-specified number of available CPUs is smaller than $np_{\max}$, PAMPAC will occasionally need to select a number of nodes to stall. This is done by traversing the tree breadth-first, *i.e.*, first over increasing step-sizes at each level of the tree and then over an increasing number of extrapolations. Thus, the processes corresponding to the tentative solutions with the largest step-size and the most extrapolations from not-quite-converged results are stalled first; this strategy makes sense since these are the iterates that are most likely to fail. They are not pruned, though, and the corrector sequences corresponding to these nodes may resume in the next iteration of the main loop.

References

[1] D. A. Aruliah, Lennaert van Veen, and Alex Dubitski. Pampac: a parallel adaptive method for arclength continuation. *ACM Trans. Math. Software*, 2014.