PULSAR Users' Guide

Parallel Ultra Light Systolic Array Runtime

Version 1.0

August 30th, 2013

LAPACK Working Note XXX

Technical Report UT-CS-XX-XXX

Electrical Engineering and Computer Science
University of Tennessee

Jack Dongarra
Jakub Kurzak
Piotr Luszczek
Ichitaro Yamazaki

Contents

Preface

PULSAR version 1.0 was released in August 2013 as a prototype software providing *proof-of-concept* implementation of a distributed memory runtime based on the systolic array concept [1] implemented in software rather than in hardware. The principles of this new implementation are based on thorough collaborative research and PULSAR's performance, scalability, and efficiency have been shown at large supercomputing installations [2, 3].

CHAPTER 1

Essentials

PULSAR is a software library that is implemented using the C programming language. It provides a programming interface for C codes and it has been designed to be efficient on distributed-memory machines based on *homogeneous* multicore processors and multi-socket systems of multicore processors. The name PULSAR is an acronym for *Parallel Ultra Light Systolic Array Runtime*.

PULSAR project website is located at:

http://icl.cs.utk.edu/pulsar/

PULSAR users' forum is located at:

http://icl.eecs.utk.edu/pulsar/forum/

1.1 Problems that PULSAR Can Solve

PULSAR is, by design, perfectly suited for systolic algorithms which are too numerous to name here. PULSAR's developers have succesfully solved dense systems of linear equations and linear least squares problems and associated computations such as matrix factorizations.

1.2 Computers for which PULSAR is Suitable

PULSAR is designed to give high efficiency and scalability on homogeneous multicore processors and multi-socket systems of multicore processors connected with a reasonably fast interconnect. As of today, the majority of such systems are on-chip symmetric multiprocessors with classic *super-scalar* processors as their building blocks (x86 and the like) augmented with short-vector SIMD extensions (SSE, AVX and the like).

Support for heterogeneous (hybrid) systems, equipped with hardware accelerators, is planned for the future.

1.3 PULSAR versus LAPACK and ScaLAPACK

PULSAR has been designed to realize the potential of systolic arrays when implemented in software. Some of the linear algebra algorithms may be implemented on top of PULSAR with substantial efficiency. For these tasks, this may offer a practical replacement for LAPACK and particularly ScaLAPACK. The performance gains over these established packages come principally from a dramatically different structure of the PULSAR code. Some of the improvement comes from the use of new or improved algorithms.

1.4 Availability of PULSAR

PULSAR is distributed as source code and is, for the most part, meant to be compiled from source on the host system. In certain cases, a pre-built binary may be provided along with the source code. Such packages, built by the PULSAR developers, will be provided as separate archives on the PULSAR download page:

http://icl.cs.utk.edu/pulsar/software/

The PULSAR team does not reserve exclusive right to provide such packages. They can be provided by other individuals or institutions. However, in case of problems with binary distributions acquired from other places, the provider needs to be asked for support rather than PULSAR developers.

1.5 Commercial Use of PULSAR

PULSAR is a freely available software package with a license that allows its use or inclusion in commercial packages. The PULSAR team asks only that proper credit be given by citing this users' guide as the official reference for PULSAR.

Like all software, this package is copyrighted. It is not trademarked. However, if modifications are made that affect the interface, functionality, or accuracy of the resulting software, the name of the routine should be changed and the modifications to the software should be noted in the modifier's documentation.

The PULSAR team will gladly answer questions regarding this software. If modifications are made to the software, however, it is the responsibility of the individual or institution who modified the routine to provide support.

1.6 Documentation of PULSAR

The PULSAR package comes with a variety of pdf and html documentation including:

- The PULSAR Users' Guide (this document)
- The PULSAR README
- The PULSAR Release Notes
- The PULSAR Installation Instructions
- The PULSAR Doxygen Reference

You will find all of these in the documentation section on the PULSAR website http://icl.cs.utk.edu/pulsar/.

1.7 PULSAR Support

PULSAR has been thoroughly tested before its release, using multiple combinations of machine architectures, compilers and BLAS libraries. The PULSAR project supports the package in the sense that reports of errors or poor performance will gain immediate attention from the developers. Such reports – and also descriptions of interesting applications and other comments – should be posted to the PULSAR users' forum:

http://icl.cs.utk.edu/pulsar/forum/

1.8 Funding

The PULSAR project was funded in part by the National Science Foundation.

CHAPTER 2

Fundamentals

2.1 Design Principles

Systolic arrays has long been touted to offer a data-centric execution model that is a viable alternative to the von Neumann architecture. However, the need for fully custom hardware to realize their potential turned out to be a major obstacle to their wide-spread use. In the past, only a few implementations of systolic arrays were attempted, which is insufficient to become a viable solution capable of competing with the proliferation of integrated circuits based on the von Neumann model. The inherent limits of a single systolic design manifest themselves in the inability of a single systolic array tackle more than one specific algorithmic challenge. This stark contrast with the general-purpose CPUs, which can tackle virtually any programmatic task, albeit with potentially low performance. The PULSAR project is a proof-of-concept effort whose aim is to show how the systolic design principles may be applied as a software solution to deliver algorithms with unprecedented strong scaling [4] capabilities in addition to very good weak scaling performance [5]. Systolic array for the QR decomposition was developed and a software virtualization layer which was used for mapping of the algorithm to a large distributed memory system. In multiple experiments [2, 3], strong scaling properties were discovered, far superior to the best existing solutions.

2.2 Conceptual Framework

The software realization of the systolic design is accomplished in PULSAR with a software infrastructure that virtualizes systolic array and/or wavefront array architectures. In this infrastructure, the node of the virtual systolic array is referred to as a Virtual Data Processor (VDP), as opposed to the Data Processing Unit (DPU) known from the original systolic array literature. The actual hardware processor (a core, a multicore CPU, or a hardware accelerator) will be referred to as a Physical Data Processor (PDP). The connections structure of the virtual array will be called a Virtual Network Topology (VNT) and to the topology of the actual hardware interconnect – Physical Network Topology (PNT). The basic idea underlying PULSAR is to map the computational problem to a systolic array architecture and then map the systolic array design to the underlying hardware through a software virtualization layer. Three distinct stages of mapping can be distinguished as listed below:

- mapping of the Direct Acyclic Graph (DAG) of the computation to the systolic array architecture (mapping of computational tasks to the VDPs and mapping of dataflow to the VNT),
- mapping of the VDPs of the systolic array to the PDPs of the hardware (typically mapping multiple VDPs to a single PDP),
- mapping of the VNT of the systolic array to the PNT of the hardware (embedding the graph of the VNT in the graph of the PNT).

PULSAR is a prototype library that implements the above concepts and is available for experimentation with new and exisiting systolic algorithms.

2.2.1 Systolic Algorithms

Some of the algorithms required by PULSAR can be categorized as standard linear algebra algorithms. Triangular solve is a good example here. Such algorithms have been very well studied and multiple systolic implementations have been proposed before. Some of PULSAR's showcase algorithms have been developed recently, specifically the class of tile algorithms including tile QR and LU factorizations [6]. Since the systolic array era precedes these discoveries, no systolic arrays for these algorithms have been proposed but recent research results[2, 3] suggest the they map well to the old paradigm.

Both the classic and the novel algorithms are very much suitable for systolic designs. The majority of them can be defined by a few loop nests with three levels

of nesting. As a result, these algorithms have regular DAGs with clearly visible structure. Also, they have easily identifiable wavefronts, mostly corresponding to the *left looking* versions of the algorithms (lazy evaluation). Nevertheless, some of these algorithms, especially the new ones can still pose design challenges due to non-trivial data dependency patterns.

Besides the opportunity to design systolic arrays for the new algorithms, a new range of opportunities presents itself in the context of software implementation. PULSAR systolic array design opportunities are much greater due to the lack of constraints that solely hardware implementations would otherwise present. In particular, different shapes of systolic arrays (e.g. rectangular, hexagonal, triangular) may be freely explored on top of PULSAR's runtime and it is possible to freely venture into higher dimensions. Planar layouts are no longer a limiting factor. Since most of the algorithms of interest may be expressed with loop nests of varying a levels of nesting, it is reasonable to suppose that their communication profile could be improved by increasing the number communication channels when moving to three dimensions.

And last, but not least, an important aspect of systolic array design is the problem of composition, i.e, connecting different systolic arrays to build one which combines different operations. The canonical example here is combining of Gaussian elimination with triangular solve to arrive at a solution to a linear system. Given the much larger design space for each of the component systolic arrays, we have a much greater freedom due to a larger design space for the composition of systolic arrays.

To sum up, PULSAR provides:

- improved virtual systolic array designs for classic algorithms,
- improved virtual systolic array designs for novel algorithms,
- novel compositions of classic and novel systolic arrays.

CHAPTER 3

Software Interfaces

3.1 Naming Conventions

PULSAR API presents an object-oriented interface to the user through C programming language calls. PULSAR functions begin with the `prt` prefix which stands for PULSAR runtime. PULSAR-specific constants are prefixed with the string `PRT`. The header file with PULSAR's C bindings is called `prt.h` and the user codes should include it to ensure proper calling sequences:

```
#include <prt.h>
```

Throughout this chapter, it is assumed that this header file have been included in the user source code; preferably, somewhere at the beginning of the file to inform the compiler of PULSAR's data structures and functions' calling conventions and trigger compile-time errors if these are not adhered to.

3.2 Memory Management

For the most part, memory allocation and deallocation of PULSAR has to be managed by the user. The routines for this purpose have the suffix `new` and `delete`,

respectively. However, PULSAR eases the burden of manual memory management by taking over the ownership of object references when they are passed into PULSAR functions. For example, functions with suffix `insert` inserts one object (`obj1`) into another object (`obj2`). After insertion, PULSAR takes over the ownership of the inserted object (`obj1`) and the user does not (and should not) have deallocate it any more. Instead, when deallocation of `obj2` will take care of deallocating both objects.

3.3 Creation and Disposal of PULSAR's VSAs

Before any other PULSAR API calls, the user should first initialize the MPI library by a call to `MPI_Init()`. The PULSAR library does not have the corresponding initialization call. Instead, initialization is performed on a per-VSA basis. But first, a new VSA has to be allocated by a call to `prt_vsa_new()`:

```
prt_vsa_t *my_vsa = prt_vsa_new(num_threads, &global_store,
  vdp_to_core);
```

The code above declares a VSA variable `my_vsa` and initializes it with an empty VSA through a call to `prt_vsa_new()`. Parameter `num_threads` specifies the number of local threads to be used for execution of VDP code. Upon execution, each VDP will receive the pointer `global_store`, which provides information common to all VDP and is defined by the user. The function `vdp_to_core` defines a user mapping between VDPs and cores. A sample mapping function that uses the first element of VDP's identifier tuple and returns the remainder of division by the number of cores:

```
int
vdp_to_core(int *vdp_id_tuple, void *global_store,
  int num_cores) {
  return vdp_id_tuple[0] % num_cores;
}
```

Just as there is no explicit function to initialize PULSAR, there is also no single call to shut down it. This is due to the fact that PULSAR does not have any internal data that require explicit allocate and deallocation. The only requirement from the memory management perspective is a proper deallocation of all the allocated VSAs with calls to `prt_vsa_delete()`:

```
prt_vsa_delete(my_vsa);
```

As a convenience to the user, the call to the `prt_vsa_delete()` function deallocates not only the VSA itself but all its associated VDPs, their channels, and tuple identifiers.

3.4 Configuration of PULSAR's VSAs

Before executing VSAs code, it is possible to configure the runtime behavior of PULSAR by calling the `prt_vsa_config_set()` function with an appropriate constant. Currently, the following calling sequences are supported:

- `prt_vsa_config_set(my_vsa, PRT_VDP_SCHEDULING, PRT_VDP_SCHEDULING_AGGRESSIVE)` forces PULSAR to execute current VDP's code as long as possible and preempt only if it runs out of data to process and/or blocks on a communication call.
- `prt_vsa_config_set(my_vsa, PRT_VDP_SCHEDULING, PRT_VDP_SCHEDULING_LAZY)` forces PULSAR to start execution of VDP's code and let it proceed independently (either block or continue running) depending on availability of data.
- `prt_vsa_config_set(my_vsa, PRT_SVG_TRACING, PRT_SVG_TRACING_ON)` turns tracing on and produces an SVG (Scalable Vector Graphics) file at the end of the VSA's execution.
- `prt_vsa_config_set(my_vsa, PRT_SVG_TRACING, PRT_SVG_TRACING_OFF)` turns the tracing facility off.

3.5 Initialization Procedure for an Empty VSA

A newly allocated VSA is empty – it has no VDPs and no connecting channels. This may be changed with a series of calls that operate on VDPs and their channels.

3.5.1 Creating a VDP

A new VDP is created with a call to `prt_vdp_new()`:

```
prt_vdp_t *my_vdp = prt_vdp_new(tuple, counter, function,
  local_store_size, num_inputs, num_outputs, trace_color);
```

this single call allocates and fully initializes a VDP. The parameters of the call are as follows:

1. `tuple` (type: `int *`) is a PULSAR tuple (see §3.7.2) that provides a unique identifier to the VDP that may be used for creating channels of communication between VDPs.
2. `counter` (type: `int`) is an integer value that determines how many execution steps the VDP has to perform.
3. `function` (type: `prt_vdp_function_t`) is a function that contains the VDP code.
4. `local_store_size` (type: `size_t`) is the size of VDP's local storage that needs to be allocated for the VDP before it starts executing.
5. `num_inputs` (type: `int`) is the number of input channels that will be used by PULSAR to communicate data to the VDP.
6. `num_outputs` (type: `int`) is the number of output channels that will be used by PULSAR to communicate data to the VDP.
7. `trace_color` (type: `int`) is the integer representation of the color which will be used to show this VDP's execution in the trace. The color components, red, green, blue, are specified in the three least significant bytes of the value. For example, black color is `0`, white is `0xffffff`, red is `0xff0000`, green is `0x00ff00`, and blue is `0x0000ff`.

After the call, references to the tuple identifier is taken over by PULSAR and the user should not deallocate it. It will be reclaimed upon deallocation of the VDP.

3.5.2 VDP's Code and Its Invocation

VDP is activated by PULSAR by calling the VDP function specified in a call to `prt_vdp_new()`. The VDP function has to be specified by the user and here is a simple implementation with relevant comments included in the code:

```
void
my_vdp_function(int *tuple, int counter,
  prt_channel_t **input, prt_channel_t **output,
  void *local_store, void *global_store) {
  int vdp_id = tuple[0]; // extract ID of VDP
```

```
  // extract pointer to my local data storage
  my_local_store *ls = (my_local_store *) local_store;

  // extract pointer to my global data storage
  my_global_store *gs = (my_global_store *) global_store;

  // compute on VDP's data based on value of 'counter'
  // communicate with other VDPs using input/output channels
}
```

3.5.3 Connecting VDPs Using Channels

Upon creation, VDPs are assigned a fixed number of input channels and output channels. They are used for one-directional communication between VDPs during execution. Channels are created with a call to `prt_channel_new()`:

```
prt_channel_t *my_channel = prt_channel_new(
  data_count, mpi_datatype,
  src_tuple, src_slot, dst_tuple, dst_slot );
```

The number of data elements in a packet is specified with `data_count` and `mpi_data` specifies the MPI data type of the elements. The source VDP of the data is given by its tuple identifier `src_tuple` and the slot number `src_slot`. Similarly, the identifier of the destination VDP and its slot number are given by `dst_tuple` and `dst_slot`, respectively. The ownership of the tuples' references is taken over by PULSAR and should not be deallocated by the user.

After successful creation, a channel has to be associated with a VDP which can be achieved with a call to the `prt_vdp_channel_insert` function:

```
prt_vdp_channel_insert(my_vdp, my_channel, direction, slot );
```

where direction can be either `PrtInputChannel` or `PrtOutputChannel` depending on the channel's direction of communication and `slot` designates the slot to use among the available slots allocated during the call to the `prt_vdp_new()` function. The ownership of the reference of a channel inserted into a VDP is taken over by PULSAR and should not be deallocated by the user. It will be deallocated upon deallocation of the VDP.

3.5.4 Communicating between VDPs through Channels

VDPs can communicate between each other using their channels.

To receive a message from `my_channel`, the `prt_channel_pop()` function has to be called:

```
prt_packet_t *my_packet = prt_channel_pop(my_channel);
compute(my_packet->data);
```

The payload of the packet can be accessed through the `data` member.

To send a packet to another VDP, we first have to allocate a packet through a call to the `prt_packet_new()`:

```
prt_packet_t *my_packet = prt_packet_new(size_in_bytes);
```

The size of the packets payload, `size_in_bytes`, has to be given in bytes. Before sending, the payload has to be initialized but since the `data` member of the packet structure is of type `void *`, it has to be cast to a type more useful for the user:

```
double *x = (double *)my_packet->data;
```

With payload data in place, the packet can be send through a call the `prt_channel_push()` function:

```
prt_channel_push(my_channel, my_packet);
```

Regardless of where the packet came from, either a call to the `prt_channel_pop()` or the `prt_packet_new()` function, it has to be deallocated with a call to the `prt_packet_release()` function:

```
prt_packet_release(my_packet);
```

3.5.5 Adding a VDP to a VSA

A VSA is a collection of VDP's and it is very easy to add a VDP to a VSA with the `prt_vsa_vdp_insert()` function after the VDP have been fully defined and connected to other VDPs:

```
prt_vsa_vdp_insert(my_vsa, my_vdp);
```

The ownership of the reference of the VDP is taken over by PULSAR and should not be deallocated by the user. It will be deallocated upon disposal of the VSA.

3.6 Executing the VSA Code

Once the VSA has been constructed out of VDPs and their connections, it be executed with a single call to the `prt_vsa_run()` function:

```
prt_vsa_run(my_vsa);
```

3.7 Auxiliary PULSAR Routines

3.7.1 Time of Day

The function `pulsar_time_of_day()` returns the current time of day in a much the same way as POSIX `gettimeofday()` does. The main difference is the fact that the return value is combined into a single floating point value that represents the number of seconds:

```
double time_of_day = pulsar_time_of_day();
```

The return value will not be a whole number for most invocations and the fraction part represents the micro-seconds obtained from the call to `gettimeofday()`.

3.7.2 Tuple Storage

For convenience, PULSAR provides a tuple data type that stores tuples of integer values. Each tuple is an array of integers terminated by the `INT_MAX` value, hence `INT_MAX` cannot be stored in a PULSAR's tuple.

A tuple may be created by a call to the `prt_tuple_new()` function:

```
int *tuple = prt_tuple_new(len, e1, e2, e3, ...);
```

where the first argument length specifies the length of the tuple and the tuple elements are `e1`, `e2`, `e3`, and so on.

Deallocation of the tuple is simply done by calling the standard C library function `free()`. If, however, a tuple is passed into any of the PULSAR functions the user looses the ownership of the reference to the tuple and PULSAR becomes responsible for deallocation the memory.

There are six macros provided in the PULSAR headers that are a safer alternative to the `prt_tuple_new()` function because they allow the compiler to catch the error of passing the wrong number of arguments after the tuple length. These macros are `prt_tuple_new1()`, `prt_tuple_new2()`, `prt_tuple_new3()`, `prt_tuple_new4()`, `prt_tuple_new5()`, and `prt_tuple_new6()`.

Caution: majority of PULSAR functions that accept tuples as parameters assume that the tuple was allocated with the `prt_tuple_new()` function. A common mistake is to pass to such a function a tuple allocated on the stack:

```
int *stack_tuple[4] = {1, 2, 3, INT_MAX};
prt_vdp_t *my_vdp = prt_vdp_new(stack_tuple, counter,
  function, size, ninputs, noutputs, color);
prt_vsa_vdp_insert(my_vsa, my_vdp);

// attempt to deallocate 'stack_tuple'
prt_vsa_delete(my_vsa); // ERROR
```

The variable `stack_tuple` resides on the function stack and is not suitable for use with PULSAR functions. In particular, it cannot be deallocated with a call to the `free` function. The error will manifest itself upon the call to `prt_vsa_delete()`.

Bibliography

[1] H. T. Kung and C. E. Leiserson. Systolic arrays (for VLSI). In *Sparse Matrix Proceedings*, pages 256–282. Society for Industrial and Applied Mathematics, 1978. ISBN: 0898711606.

[2] Jakub Kurzak, Piotr Luszczek, Mark Gates, Ichitaro Yamazaki, and Jack Dongarra. Virtual systolic array for QR decomposition. In *IPDPS'13, the 27st IEEE Int. Parallel and Distributed Processing Symposium*, Boston, MA, May 2013. IEEE Computer Society Press.

[3] Guillaume Aupy, Mathieu Faverge, Yves Robert, , Jakub Kurzak, Piotr Luszczek, and Jack Dongarra. Implementing a systolic algorithm for QR factorization on multicore clusters with PaRSEC. In *PROPER Workshop, in conjunction with EuroPar 2013*, Aachen, Germany, August 2013.

[4] G. M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485, Atlantic City, N.J., APR 18-20 1967. AFIPS Press, Reston, VA.

[5] J. L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[6] A. Buttari, J. Langou, J. Kurzak, and J. J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Comput. Syst. Appl.*, 35:38–53, 2009. DOI: 10.1016/j.parco.2008.10.002.