Advanced Programming on Hybrid Computers in PaR-SEC Using Symbolic Representation

How to program with PaRSEC

G. Bosilca, A. Bouteiller, D. Genet, T. Herault
With contributions from M. Faverge, A. Danalis, A. Guermouche, et al.

University of Tennessee - ICL
Bordeaux INP - Inria - CNRS - Univ. de Bordeaux

January 7, 2019

1

PaRSEC Framework

1.1

PaRSEC Framework

Setup a PaRSEC environment

PaRSEC

1. Clone the last version of the bitbucket repository (requires bitbucket login):
   `git clone git@bitbucket.org:icldistcomp/parsec.git`
2. Depends on: HwLoc, MPI, CUDA, Plasma
3. Configuration through cmake ($> 2.8.7$)
   `mkdir build; cd build; cmake .. [-DCMAKE_VARIABLE=VALUE]`
   - `CMAKE_INSTALL_PREFIX`: Installation prefix directory
   - `CMAKE_BUILD_TYPE`: Type of compilation (RelWithDebInfo, Debug . . .)
   - `BUILD_DPLASMA`: Enable/Disable the compilation of DPlasma library
   - `PARSEC_DIST_WITH_MPI`: Enable/Disable the compilation with MPI
   - `PARSEC_GPU_WITH_CUDA`: Enable/Disable the support for CUDA kernels
   - See INSTALL file and contrib directory for more information
4. `make -j 4 install`
5. Set $PATH and $LD_LIBRARY_PATH

More information https://bitbucket.org/icldistcomp/parsec/wiki/Home

Compiling and linking a program using PaRSEC

- `export PKG_CONFIG_PATH=$PKG_CONFIG_PATH:$PARSEC_DIR/lib/pkgconfig`
- `CFLAGS += ‘pkg-config --cflags parsec‘`
- `LDFLAGS += ‘pkg-config --libs parsec‘`
- `cc -o myprogram mybasename.c -I. ${CFLAGS} ${LDFLAGS}`

Dynamic Task Discovery: parsec_insert_task

- Does not require special steps

Parameterized Task Graph: From `.jdf` to `.c` file

- `$PARSEC_DIR/bin/parsec_ptgpp -i myfile.jdf -o mybasename`
- Generates the `mybasename.c` and `mybasename.h` files from the `.jdf`
- `.c` file can be compiled by any C compiler

Makefile rule

-
  ```
  %.c %.h: %.jdf
          $PARSEC_DIR/bin/parsec_ptgpp [--noline] -i $< -o $*
  ```

1.2

PaRSEC Framework

Initializing PaRSEC

The basics for a PaRSEC program

Ex00_StartStop.c

- How to get and compile PaRSEC?
- PaRSEC initialization and finalization
- How to compile a program using PaRSEC?
- How to wait for the end of an algorithm?

Skeleton PaRSEC application

```
parsec_context_t* parsec = parsec_init(-1, NULL, NULL);  /* start a PaRSEC instance */

parsec_taskpool_t* parsec_tp = parsec_dtd_taskpool_new();
rc = parsec_enqueue(parsec, parsec_tp);
rc = parsec_context_start(parsec);

parsec_vector_t dDATA;
parsec_vector_init( &dDATA, matrix_Integer, matrix_Tile,
                    nodes, rank,
                    1 /* tile_size*/,         N /* Global vector size*/,
                    0 /* starting point */,  1 /* block size */ );

/** Generate Tasks **/

rc = parsec_context_wait( parsec );
rc = parsec_fini( &parsec );
```

PaRSEC header must always be included

```
#include <parsec.h>
```

Main data structure

```
parsec_context_t* parsec_init( int ncores, int* pargc, char** pargv[]);
int parsec_fini( parsec_context_t** pcontext );
```

- parsec_context_t is an instance of the PaRSEC engine
- The new context will be passed as arguments of all the main functions
- **ncores** defines the number of cores on the current process
  if -1, the number of cores is set to the number of physical cores
- **pargc, pargv** forwards the program parameters to the runtime
  Takes into account all options after the first -- in the command line
- Multiple PaRSEC instances can coexist, but they might conflict on resources
- New contexts are not automatically started. This allows for creating and populating PaRSEC instances outside the critical path.

Main data structure

```
int parsec_context_start(parsec_context_t* context);
int parsec_context_test(parsec_context_t* context);
int parsec_context_wait(parsec_context_t* context);
```

- `start` starts the infrastructure associated with the context (threads, accelerators, communication engines, memory allocators). This must be paired with `test` or `wait`
- `test` checks the status of a context. Returns 1 if the context has no active work (no known tasks, no pending communications, ...), 0 otherwise.
- `wait` progresses the execution context until all known operations have been completed and no further operations are available. Upon return the context is disabled and the associated resources are released.
- `test` only investigate the local state of the context while `wait` waits for the distributed context (basically until all work on this context is completed on all nodes associated with the context).

1.3

PaRSEC Framework

Dynamic Task Discovery

```
typedef int (parsec_dtd_funcptr_t)(parsec_execution_stream_t*,
                                   parsec_task_t*);
void parsec_dtd_taskpool_insert_task( parsec_taskpool_t  *tp,
                    parsec_dtd_funcptr_t *fpointer, int priority,
                    char *name_of_kernel, ... );
```

- The ... are tuples of <size | PASSED_BY_REF, pointer | DATA, flags>:

VALUE	copy a well-defined variable into the task
SCRATCH	prepare space into the task structure using the size parameter
usage	INOUT/INPUT/OUTPUT: how the argument is manipulated by the task. Allows the correct tracking of dependencies by the runtime
AFFINITY	task affinity with this data
DONT_TRACK	ignore this dependency
data	DATA_OF or DATA_OF_KEY

Decode arguments

```
      void parsec_dtd_unpack_args( parsec_task_t *this_task, ... );
```

- should be called in order for each argument
- this is C: no type validation

The STF Hello World application

dtd_example_hello_world.c

```
int task_hello_world( parsec_execution_stream_t *es,
                      parsec_task_t *this_task )
{
  printf("Hello␣World␣my␣rank␣is:␣%d\n",
         this_task->taskpool->context->my_rank);
  return PARSEC_HOOK_RETURN_DONE;
}
```

```
parsec_dtd_taskpool_insert_task( dtd_tp, task_hello_world,
                                 0, "Hello_World_task",
                                 0 );  /* No more arguments */
```

The STF Hello World with arguments

dtd_example_hello_arg.c

```
int task_hello_with_arg( parsec_execution_stream_t *es,
                         parsec_task_t *this_task )
{
  int *i;
  parsec_dtd_unpack_args( this_task,
                          UNPACK_VALUE,  &i);
  printf("Hello␣World␣my␣index␣is:␣%d\n", *i);
  return PARSEC_HOOK_RETURN_DONE;
}
```

```
for( int i = 0; i < 10; i++ ) {
  parsec_dtd_taskpool_insert_task( dtd_tp, task_hello_with_arg,
                                   0, "Hello_World_task",
                                   sizeof(int), &i, VALUE,
                                   0 );  /* No more arguments */
```

What if

- we change the number of cores in the example ?
- we have too many tasks (not 10 but millions)

The STF Hello World untied with arguments

```
int task_to_insert_task_hello_world( parsec_execution_stream_t *es,
                                     parsec_task_t *this_task )
{
    parsec_taskpool_t *dtd_tp = this_task->taskpool;
    int *n;
    parsec_dtd_unpack_args( this_task,
                            UNPACK_VALUE,  &n);
    for( int i = 0; i < (*n); i++ )
        parsec_dtd_taskpool_insert_task( dtd_tp, task_hello_world, 0, "Hello_World_task",
                                         sizeof(int), &i, VALUE, 0);
    return PARSEC_HOOK_RETURN_DONE;
}
```

```
parsec_dtd_taskpool_insert_task( dtd_tp, task_to_insert_task_hello_world,
                                 0, "Untied␣Hellow␣World␣Generator",
                                 sizeof(int), &n, VALUE,
                                 0 );  /* No more arguments */
```

What if

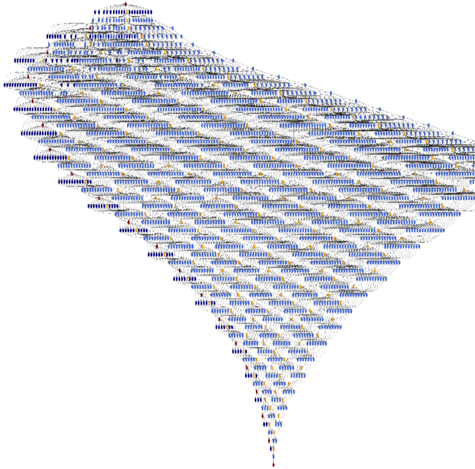
- we change the number of cores in the example ?
- ave too many tasks (not 10 but millions)

What about scalability?

- Too many tasks known, too much memory overhead, too much scheduling overhead
- Possible to reduce but then the used drive the execution by the way the sequential code is written
- In distributed every process needs to analyse the entire DAG to be able to track all data (so that it can infer the communications
- Almost impossible to detect collective patterns without explicit description

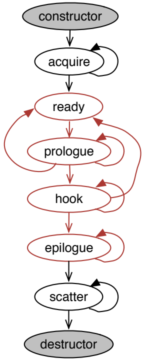
Domain Specific Language/Extension

- Ideally we only need to be able to know a small section of the DAG (the tasks with ready dependencies, and eventually a little bit further)
- Let computational scientists define how they want to interact with the runtime (!)



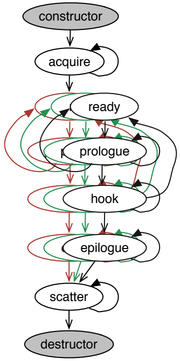
Deep-dive into PaRSEC task structure

- PaRSEC tasks are instances of task classes
- each task class is a state machine, changing state is the task of the DSL
- tasks are usually created only when all input dependencies are satisfied (prologue)
- before that the tasks are virtual and do not require any memory
- tasks leave traces until all successors have been identified (scatter)



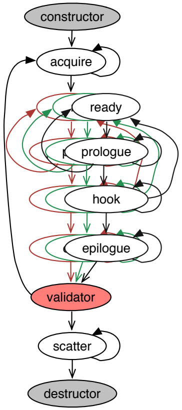
Deep-dive into PaRSEC task structure

- PaRSEC tasks are instances of task classes
- each task class is a state machine, changing state is the task of the DSL
- tasks are usually created only when all input dependencies are satisfied (prologue)
- before that the tasks are virtual and do not require any memory
- tasks leave traces until all successors have been identified (scatter)
- provide support for heterogeneous environments by using multiple sets of functions



Deep-dive into PaRSEC task structure

- PaRSEC tasks are instances of task classes
- each task class is a state machine, changing state is the task of the DSL
- tasks are usually created only when all input dependencies are satisfied (prologue)
- before that the tasks are virtual and do not require any memory
- tasks leave traces until all successors have been identified (scatter)
- provide support for heterogeneous environments by using multiple sets of functions
- for resilience: provide means to either validate the task's results or to checkpoint their outcome (asynchronously)



1.4

PaRSEC Framework

Playing with the Parameterized Task Graph (JDF)

A simple HelloWorld example with a jdf

Ex01_HelloWorld.jdf

- How to submit an algorithm/object to the runtime?
- How to write a JDF?
- A simple sequential, and embarrassingly parallel, example

JDF Objects

```
parsec_JDFName_taskpool_t *parsec_JDFName_new( ... );
int parsec_enqueue( parsec_context_t* , parsec_taskpool_t* );
void parsec_taskpool_free(parsec_taskpool_t *handle);
```

- This is the structure associated to each algorithm
- Each JDF has it's own handle structure that inherits from the main `parsec_taskpool_t`
- `parsec_JDFName_new` is the generated function from the jdf that will create the object. It takes the union of the used descriptors, and the non hidden private as parameters (see generated .h for the correct prototype)
- enqueue submits the handle to the give context. the execution will start only when `parsec_context_start` or `parsec_context_wait` is called
- `free` calls the object destructor that unregister the handle from the context and releases the memory

Main structure of the JDF language

```
Prologue

Private

TaskClass1

TaskClass2

...

Epilogue
```

Prologue/Epilogue (optional):

- Syntax:

  ```
  extern "C" %{
  // Content
  %}
  ```

- Optional in the syntax
- Not interpreted / Directly copy-paste in the generated .c file
- Allows to include personal headers
- Allows to define functions, macros, global variables
- ...

Main structure of the JDF language

```
Prologue

Private

TaskClass1

TaskClass2

...

Epilogue
```

Private (optional):

- Optional in the syntax
- Defines variables attached to one instance of the JDF handle
- Can be accessed from any tasks in this handle
- Parameters of the handle new() function

Main structure of the JDF language

```
Prologue

Private

TaskClass1

TaskClass2

...

Epilogue
```

TaskClass (required $>= 1$):

- A JDF file need at least **one** task class
- No limit in the number of tasks

TaskClass

```
TaskClassName( i, j, ... )

Locals

/* Partitioning */
: descriptor( x, y, ... )

Flow1
Flow2
...

/* Priority */
; priority

Body1
Body2
...
```

TaskClassName (required):

- Must be unique per JDF (similar to function name in a program)
- Parameters are integers. They identify each instance of the task.
- Each task class is currently limited to `MAX_PARAM_COUNT` parameters

TaskClass

```
TaskClassName( i, j, ... )

Locals

/* Partitioning */
: descriptor( x, y, ... )

Flow1
Flow2
...

/* Priority */
; priority

Body1
Body2
...
```

Locals (required $>= 1$):

- Contains the variables from the taskclass execution space
- Can be defined as a value or a range:

  `p1 = start [.. end [.. inc]]`
- Each local can be defined in function of the previously defined local variables

  `p2 = start .. p1 [.. inc]`
- Can be defined through a function

  `p3 = inline_c %{return f(a, b, p1 ); %}`
- Can be a range only if part of the execution space
- start and end bounds are both included in the execution space
- Maximum number of locals is defined by `MAX_LOCAL_COUNT`

TaskClass

```
TaskClassName( i, j, ... )

Locals

/* Partitioning */
: descriptor( x, y, ... )

Flow1
Flow2
...

/* Priority */
; priority

Body1
Body2
...
```

Partitioning (required):

- Defines where the task will be executed: MPI process, and possibly NUMA node
- Must be consistent on all nodes
- Given with a parsec_data_collection_t structure (shortcut parsec_dc_t)
- Takes parameters from the Private, or Local variables
- Not possible to give directly an integer (rank/vpid)
- Can be a different data collection than one used in the algorithm
- Can be dynamically changed **only** if everyone involved knows about the changes

TaskClass

```
TaskClassName( i, j, ... )

Locals

/* Partitioning */
: descriptor( x, y, ... )

Flow1
Flow2
...

/* Priority */
; priority

Body1
Body2
...
```

Flows (required $>= 1$):

- Defines a data used by the task
- Defines the type of access to each flow (R and/or W)
- Defines the incoming and outgoing dependencies associated to this each

TaskClass

```
TaskClassName( i, j, ... )

Locals

/* Partitioning */
: descriptor( x, y, ... )

Flow1
Flow2
...

/* Priority */
; priority

Body1
Body2
...
```

Priority (optional):

- Define the priority of the task as an integer
- Can be given as an integer or an expression

  `; prio`
- The higher the value, the higher the priority

TaskClass

```
TaskClassName( i, j, ... )

Locals

/* Partitioning */
: descriptor( x, y, ... )

Flow1
Flow2
...

/* Priority */
; priority

Body1
Body2
...
```

Body (required $>= 1$):

- Define the function of the task
- Must be pure: modify only local variables, and read-only on private ones
- One body per type of device
- Need at least one CPU body
- Body are prioritized by order of appearance (Ex: CUDA, RECURSIVE, CPU)

Body (CPU)

```
BODY
{
  /**
   * Code that will be executed on CPU
   * and that can use any private, local, or flow
   */
}
END
```

- Each body is delimited by the keywords BODY and END and must be valid C code
- The code is copy/paste in a function and has access privately to all parameters, locals and flows of the taskclass.
- Any thread in the system belonging to the PaRSEC context where the task is generated can execute this task
- The task is N.O.T. allowed to block the execution thread indefinitely (careful with mutexes/conditions/barriers/. . .)

1.5

PaRSEC Framework

Chain

Add an execution order to the tasks: Chain

Ex02_Chain.jdf

- How to add its own private variables?
- How to exchange data between tasks?
- How to define the type of a dependency?

More details about private variables

```
Name
Name [type = "CType"]
Name [type = "CType" default = "Value"]
Name [type = "CType" hidden = ON default = "Value"]
```

- Each private is part of the `parsec_JDFName_new()` function prototype
- Each private needs a name, and some optional properties

  type Defines the variable datatype (int by default).

  default Defines the default value of the variable (unset by default). If the variable has a default value, it is hidden from the `parsec_JDFName_New()` function prototype

- Implicitly includes all data collections used in the JDF for task partitioning

Flows

```
AccessType Name <- NULL
                <- ( m == 0 ) ? NEW
                <- ( m == 1 ) ? Name1 TaskA(m) : dataA( m )
                -> Name2 TaskA(n)
                -> Name1 TaskA(m)
```

- A flow must have an access type:
  READ, RW, WRITE, or CTL
- A flow has a unique name that:
  - defines a (void*) variable in the body to access the associated data
  - identifies the flow to connect them together
- A flow can have multiple input, and/or output dependencies defined by the direction of the arrow: input (<-) and output (->)
- A flow dependency can have multiple properties that helps define its datatype (See to go further)

Flows (Input)

```
WRITE A <- NEW
RW    B <- ( m == 0 ) ? NEW : dataA(m)
READ  C <- ( m == 1 ) ? A TaskA(m) : NULL
        <- B TaskA(m)
```

- An input dependency can be used on all types of flows
- **Only one single** dependency can be input, thus the first one to match cancels all the following ones.
  In C example, the first input dependency discards the second one.
- There must be an input for the **whole** execution space of the task
- A WRITE flow can only have NEW as input, but it is not mandatory
- A READ flow can not have NEW as an input

Flows (Output)

```
WRITE A -> A task(m)
READ  B -> ( m == 0 ) ? B taskA(m)
RW    C -> ( m == 1 ) ? A TaskA(m) : dataA(n)
        -> B TaskA(m)
```

- An output dependency can be used on all types of flows
- All matching outputs are performed
- The output dependencies does not need to cover the whole execution space
- A WRITE flow can only go toward another task
- NULL or NEW can **not** be associated to output dependencies
- A NULL data can **not** be forwarded to another task

Dependencies datatypes

```
#include <parsec/arena.h>
int parsec_arena_construct(parsec_arena_t* arena,
                           size_t elem_size,
                           size_t alignment,
                           parsec_datatype_t opaque_dtt);
void parsec_arena_destruct(parsec_arena_t* arena);
```

- Arena can be seen as an extension to MPI_Datatype
- They define the datatype of the dependencies (not the flows) in the JDF
- They also work as a freelist to have lists of pre-allocated spaces for each datatype. They are used by: the communication engine; the accelerators; or with the NEW keyword from RW or WRITE flows
- `construct` initializes a pre-allocated arena structure
- Each element in the arena is of type `opaque_dtt`, and is of size `elem_size`
- `elem_size` is equivalent to `MPI_Extent`
- Allocated spaces are memory aligned based on the `alignment` parameter: `PARSEC_ARENA_ALIGNMENT_[64b | INT | PTR | SSE | CL1 ]`
- `destruct` releases all the memory elements allocated

parsec_datatype_t

```
#include <parsec/datatype.h>
```

Map the datatype creation to the well designed and well known MPI datatype manipulation. However, right now we only provide the most basic types and functions to mix them together.

Types, parsec_datatype_xxx_t with xxx among
int, int8, int16, int32, int64, uint8, uint16, uint32, uint64, float, double, long_double, complex, double_complex

Functions:
parsec_type_size, parsec_type_create_contiguous, parsec_type_create_vector, parsec_type_create_hvector, parsec_type_create_indexed, parsec_type_create_indexed_block, parsec_type_create_struct, parsec_type_create_resized

1.6

PaRSEC Framework

Distributed Chain

A distributed chain of tasks

Ex03_ChainMPI.jdf

- How to create a data collection?
- How to specify the distribution, and the associated data?
- How to extend the existing collections structure?

Task and data collections

```
#include <parsec/data_distribution.h>

void parsec_data_collection_init(parsec_data_collection_t *d,
                                 int nodes, int myrank );
void parsec_data_collection_destroy(parsec_data_collection_t *d );
```

- This is the structure that is provided to any DSL to describe the task and data distribution, and the data location
- `init` initializes the fields of the data structure to default values
  - the data collection exists but it is not yet able to deliver data
- `destroy` cleans up all allocated data used by the data collection
- In most cases, this API is not to be used directly but instead wrapped in specialized data collections

rank_of / vpid_of / data_of functions

```
#include <parsec/data_distribution.h>

uint32_t (*rank_of)(parsec_data_collection_t *d, ...);
int32_t  (*vpid_of)(parsec_data_collection_t *d, ...);

parsec_data_t* (*data_of)(parsec_data_collection_t *d, ...);
```

- Each data collection contains a set of functions that describes how the data is distributed across the nodes and how to access it
- Each function takes as parameter the pointer to the data collection itself, and a variadic parameter, usually made of one or multiple integers
  - **rank_of** Returns the rank of the process associated to the given parameters
  - **vpid_of** Returns the virtual process (NUMA node) id of the process associated to the given parameters
  - **data_of** Returns the `parsec\_data\_t` structure that describes the piece of data associated to the given parameters
- A second version of these function exist (*_key) where the variadic parameters are replaced by an unsigned 64 bits key. You can obtain the key using `data_key` accessor associated to the data ction.

Deriving data collections from parsec_data_collection_t

```
#include <parsec/data_distribution.h>

typedef struct my_data_collection_s {
  parsec_data_collection_t super;
  ...;
} my_data_collection_t;
```

- If information need to be stored in the object, then a derived collection *inheriting* from the main structure should be defined.
- The parent type, `parsec_data_collection_t` must always be the first field
- Together with the `rank_of` and `data_of` you can defined any distributed data collections

1.7

PaRSEC Framework

Chain with data

Exploiting an application data

Ex04_ChainData.jdf

- How to give an application data to PaRSEC?
- How to implement the data_of function of a collection?

parsec_data_t

```
#include <parsec/data_distribution.h>
parsec_data_t *
parsec_data_create( parsec_data_t **holder,
                    parsec_data_collection_t *dc,
                    parsec_data_key_t key, void *ptr, size_t size );
void
parsec_data_destroy( parsec_data_t *holder );
```

- This structure stores information about each piece of data that will be provided to the engine
  - the location and size
  - the existing copies on the different devices
  - the versions of the copies
- `create` initializes the data structure `holder` associated to the piece of data `ptr` of `size` bytes. This data is associated to the collection `desc`, and `key` is its unique identifier in this data collection
- `key` is a unique identifier of the elementary piece of data
- `destroy` frees the existing copies of the data, and the structure

1.8

PaRSEC Framework

Broadcast

Broadcast an information

Ex05_Broadcast.jdf

- How to hide a private variable from the `parsec_JDFName_new()` protoype?
- How to broadcast an information from one task to many others?

1.9

PaRSEC Framework

Read After Write Anti-dependencies

The danger of the RAW dependencies

Ex06_RAW.jdf

- What happen if the JDF contains a Read After Write dependency?

1.10

PaRSEC Framework

Control Flows

CTL flows

Ex07_RAW_CTL.jdf

- How to prevent RAW dependencies?
- How to add some control flows? (for adding sequentiality, RAW problems, . . .)

CTL Flows

```
CTL  ctl1 <- ctl2 Task1( 0 .. k )
          <- ( m == 0 ) ? ctl1 Task2( k )
          <- ( m == 1 ) ?
          -> ctl1 Task1( m, n )
          -> ctl2 TaskA( m .. n )
```

- Is a flow of type CTL
- Has a name as a regular flow
- Does not have associated data (neither properties)
- Can only have other CTL flows as dependencies
- Can gather multiple inputs from one or multiple tasks

1.11

PaRSEC Framework

To go further with dependencies

Dependencies properties

```
Type Name <- NULL
          <- ( m == 0 ) ? NEW                        [ type=DEFAULT ]
          <- ( m == 1 ) ? A TaskA(m) : B TaskA(m)  [ type=ArenaType ]
          -> dataA( m, n )                   [ layout=CType count=nb ]
          -> A TaskA(m)                [ type=ArenaType displ=offset ]
```

type Gives the arena describing the data type of the dependency
By default, it is DEFAULT

layout Specifies the smallest atomic unit that can be sent

count Give the size of the dependency in multiple of the data layout
Used only with layout property

displ Specify a displacement in the data to send, or in the location to receive the information in bytes

- Different outputs can have different types
  - The input type must encompass the output types
  - In the case of a WRITE flow, if not NEW input dependency is defined, the first output defines the type that encompasses all others

2

PTG Cholesky

How to program a Cholesky with PTG programming model

Going from a sequential code to a parameterized task graph of the same application, with the example of a Cholesky decomposition.

1. Matlab code
2. Sequential code
3. Blocked code as in LAPACK/ScaLAPACK
4. Tiled algorithm as in PLASMA
5. Tiled algorithm with sequential task flow (STF) model
6. Tiled algorithm with parameterized task graph (PTG) model

Sequential code

```
for (k=0; k<N; k++) {
  a[k][k] = sqrt( a[k][k] )
  for (m=k+1; m<N; m++) {
    a[m][k] = a[m][k] / a[k][k]
  }
  for (n=k+1; n<N; n++) {
    a[n][n] = a[n][n] - a[n][k] * a[n][k]
    for (m=n+1; m<N; m++) {
       a[m][n] = a[m][n] - a[m][k] * a[n][k]
    }
  }
}
```

- Start with a sequential scalar code

Tiled algorithm (pseudo Matlab)

```
for (k=0; k<NT; k++) {
  A[k][k] = Cholesky( A[k][k] )
  for (m=k+1; m<NT; m++) {
    A[m][k] = A[m][k] / A[k][k]
  }
  for (n=k+1; n<NT; n++) {
    A[n][n] = A[n][n] - A[n][k] * A[n][k]
    for (m=n+1; m<NT; m++) {
       A[m][n] = A[m][n] - A[m][k] * A[n][k]
    }
  }
}
```

- Move from scalar to matrix operation ($a \to A$)
- Based on the algorithm, it might be: really simple (Cholesky, LU without pivoting), or more complex (QR)
- Here, each operation is independent and can be a function

Tiled algorithm

```
for (k=0; k<NT; k++) {
  POTRF( A[k][k] );
  for (m=k+1; m<NT; m++)
     TRSM( A[k][k], A[m][k] );
  for (n=k+1; n<NT; n++) {
    SYRK( A[n][k], A[n][n] );
    for (m=n+1; m<NT; m++)
        GEMM( A[m][k], A[n][k], A[m][n] );
  }
}
```

- How to move to a task based runtime from this?
  1. Runtime with STF model: Quark, StarPU, OmpSS, ...
  2. Runtime with PTG model: Intel CnC, PaRSEC

Tiled algorithm

```
for (k=0; k<NT; k++) {
  POTRF( A[k][k] );
  for (m=k+1; m<NT; m++)
     TRSM( A[k][k], A[m][k] );
  for (n=k+1; n<NT; n++) {
    SYRK( A[n][k], A[n][n] );
    for (m=n+1; m<NT; m++)
        GEMM( A[m][k], A[n][k], A[m][n] );
  }
}
```

- How to move to a task based runtime from this?
  1. Runtime with STF model: Quark, StarPU, OmpSS, . . .
  2. Runtime with PTG model: Intel CnC, PaRSEC

PTG programming: Need to think local

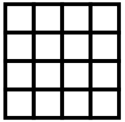
Need to think:

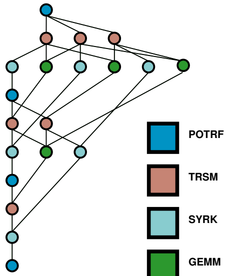
- With dependencies
- With data movements
- Without loops

Let's study the case of the TRSM task in the Cholesky example

PTG programming (DAG based)

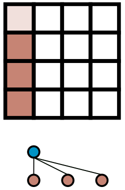
```
TRSM(k, m)
```





PTG programming (DAG based)

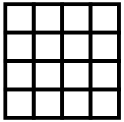
```
TRSM(k, m)





// Flows & their dependencies
READ  A <- A POTRF(k)
RW    C <- (k == 0) ? dataA(m, k)
```

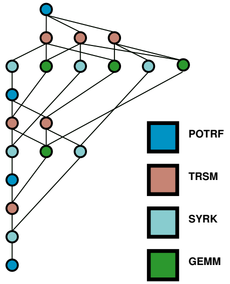


🟦	POTRF
🟫	TRSM
▨	SYRK
🟩	GEMM

PTG programming (DAG based)

```
TRSM(k, m)
```

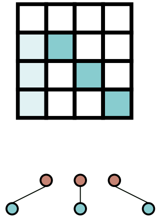




```
// Flows & their dependencies
READ  A <- A POTRF(k)
RW    C <- (k == 0) ? dataA(m, k)
        <- (k != 0) ? C GEMM(k-1, m, k)
```

PTG programming (DAG based)

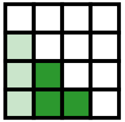
```
TRSM(k, m)




// Flows & their dependencies
READ  A <- A POTRF(k)
RW    C <- (k == 0) ? dataA(m, k)
        <- (k != 0) ? C GEMM(k-1, m, k)
        -> A SYRK(k, m)
```

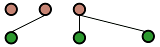


🟦	POTRF
🟫	TRSM
🟦	SYRK
🟩	GEMM

PTG programming (DAG based)

```
TRSM(k, m)




// Flows & their dependencies
READ  A <- A POTRF(k)
RW    C <- (k == 0) ? dataA(m, k)
        <- (k != 0) ? C GEMM(k-1, m, k)
        -> A SYRK(k, m)
        -> A GEMM(k, m, k+1..m-1)
```

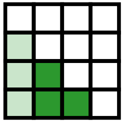


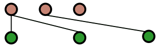


🟦	POTRF
🟫	TRSM
🟦	SYRK
🟩	GEMM

PTG programming (DAG based)

```
TRSM(k, m)




// Flows & their dependencies
READ  A <- A POTRF(k)
RW    C <- (k == 0) ? dataA(m, k)
        <- (k != 0) ? C GEMM(k-1, m, k)
        -> A SYRK(k, m)
        -> A GEMM(k, m, k+1..m-1)
        -> B GEMM(k, m+1..NT-1, m)
```

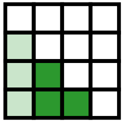




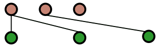
🟦	POTRF
🟫	TRSM
🟦	SYRK
🟩	GEMM

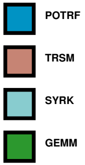
PTG programming (DAG based)

```
TRSM(k, m)
```



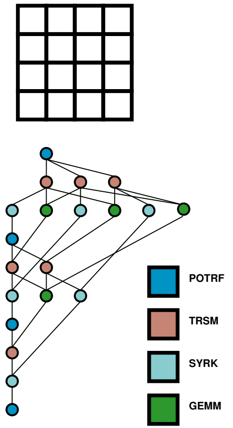
```
// Flows & their dependencies
READ   A <- A POTRF(k)
RW     C <- (k == 0) ? dataA(m, k)
         <- (k != 0) ? C GEMM(k-1, m, k)
         -> A SYRK(k, m)
         -> A GEMM(k, m, k+1..m-1)
         -> B GEMM(k, m+1..NT-1, m)
         -> dataA(m, k)
```





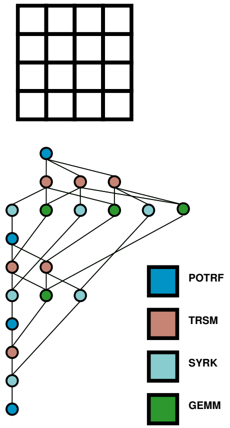
PTG programming (DAG based)

```
TRSM(k, m)

// Execution space
k = 0   .. NT-1
m = k+1 .. NT-1



// Flows & their dependencies
READ  A <- A POTRF(k)
RW    C <- (k == 0) ? dataA(m, k)
        <- (k != 0) ? C GEMM(k-1, m, k)
        -> A SYRK(k, m)
        -> A GEMM(k, m, k+1..m-1)
        -> B GEMM(k, m+1..NT-1, m)
        -> dataA(m, k)
```



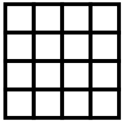
PTG programming (DAG based)

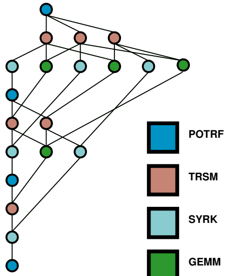
```
TRSM(k, m)

// Execution space
k = 0   .. NT-1
m = k+1 .. NT-1

// Partitioning
: dataA(m, k)

// Flows & their dependencies
READ  A <- A POTRF(k)
RW    C <- (k == 0) ? dataA(m, k)
        <- (k != 0) ? C GEMM(k-1, m, k)
        -> A SYRK(k, m)
        -> A GEMM(k, m, k+1..m-1)
        -> B GEMM(k, m+1..NT-1, m)
        -> dataA(m, k)
```



PTG programming (DAG based)

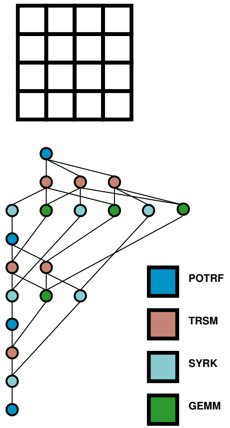
```
TRSM(k, m)

// Execution space
k = 0   .. NT-1
m = k+1 .. NT-1

// Partitioning
: dataA(m, k)

// Flows & their dependencies
READ  A <- A POTRF(k)
RW    C <- (k == 0) ? dataA(m, k)
        <- (k != 0) ? C GEMM(k-1, m, k)
        -> A SYRK(k, m)
        -> A GEMM(k, m, k+1..m-1)
        -> B GEMM(k, m+1..NT-1, m)
        -> dataA(m, k)

BODY
 trsm(A, C);
END
```





PTG programming (DAG based)

```
TRSM(k, m)

// Execution space
k = 0   .. NT-1
m = k+1 .. NT-1

// Partitioning
: dataA(m, k)

// Flows & their dependencies
READ  A <- A POTRF(k) [type = LOWER]
RW    C <- (k == 0) ? dataA(m, k)
        <- (k != 0) ? C GEMM(k-1, m, k)
        -> A SYRK(k, m)
        -> A GEMM(k, m, k+1..m-1)
        -> B GEMM(k, m+1..NT-1, m)
        -> dataA(m, k)

BODY
 trsm(A, C);
END
```



PTG programming (Code based)

```
TRSM(k, m)
```

```
for (k=0; k<N; k++) {
  POTRF(RW, A[k][k]);
  for (m=k+1; m<N; m++)
    TRSM(R,  A[k][k],
         RW, A[m][k]);
  for (n=k+1; n<N; n++) {
    SYRK(R,  A[n][k],
         RW, A[n][n]);
    for (m=n+1; m<N; m++)
      GEMM(R,  A[m][k],
           R,  A[n][k],
           RW, A[m][n]);
  }
}
```

PTG programming (Code based)

```
TRSM(k, m)




// Flows & their dependencies
READ   A <- A POTRF(k)
RW     C <- (k == 0) ? dataA(m, k)
```

```
for (k=0; k<N; k++) {
  POTRF(RW, A[k][k]);
  for (m=k+1; m<N; m++)
    TRSM(R,  A[k][k],
         RW, A[m][k]);
  for (n=k+1; n<N; n++) {
    SYRK(R,  A[n][k],
         RW, A[n][n]);
    for (m=n+1; m<N; m++)
      GEMM(R,  A[m][k],
           R,  A[n][k],
           RW, A[m][n]);
  }
}
```

PTG programming (Code based)

```
TRSM(k, m)




// Flows & their dependencies
READ  A <- A POTRF(k)
RW    C <- (k == 0) ? dataA(m, k)
        <- (k != 0) ? C GEMM(k-1, m, k)
```

```
for (k=0; k<N; k++) {
  POTRF(RW, A[k][k]);
  for (m=k+1; m<N; m++)
    TRSM(R,  A[k][k],
         RW, A[m][k]);
  for (n=k+1; n<N; n++) {
    SYRK(R,  A[n][k],
         RW, A[n][n]);
    for (m=n+1; m<N; m++)
      GEMM(R,  A[m][k],
           R,  A[n][k],
           RW, A[m][n]);
  }
}
```

PTG programming (Code based)

```
TRSM(k, m)




// Flows & their dependencies
READ  A <- A POTRF(k)
RW    C <- (k == 0) ? dataA(m, k)
        <- (k != 0) ? C GEMM(k-1, m, k)
        -> A SYRK(k, m)
```

```
for (k=0; k<N; k++) {
  POTRF(RW, A[k][k]);
  for (m=k+1; m<N; m++)
    TRSM(R,  A[k][k],
         RW, A[m][k]);
  for (n=k+1; n<N; n++) {
    SYRK(R,  A[n][k],
         RW, A[n][n]);
    for (m=n+1; m<N; m++)
      GEMM(R,  A[m][k],
           R,  A[n][k],
           RW, A[m][n]);
  }
}
```

PTG programming (Code based)

```
TRSM(k, m)




// Flows & their dependencies
READ  A <- A POTRF(k)
RW    C <- (k == 0) ? dataA(m, k)
        <- (k != 0) ? C GEMM(k-1, m, k)
        -> A SYRK(k, m)
        -> A GEMM(k, m, k+1..m-1)
```

```
for (k=0; k<N; k++) {
  POTRF(RW, A[k][k]);
  for (m=k+1; m<N; m++)
    TRSM(R,  A[k][k],
         RW, A[m][k]);
  for (n=k+1; n<N; n++) {
    SYRK(R,  A[n][k],
         RW, A[n][n]);
    for (m=n+1; m<N; m++)
      GEMM(R,  A[m][k],
           R,  A[n][k],
           RW, A[m][n]);
  }
}
```

PTG programming (Code based)

```
TRSM(k, m)




// Flows & their dependencies
READ  A <- A POTRF(k)
RW    C <- (k == 0) ? dataA(m, k)
        <- (k != 0) ? C GEMM(k-1, m, k)
        -> A SYRK(k, m)
        -> A GEMM(k, m, k+1..m-1)
        -> B GEMM(k, m+1..NT-1, m)
```

```
for (k=0; k<N; k++) {
  POTRF(RW, A[k][k]);
  for (m=k+1; m<N; m++)
    TRSM(R,  A[k][k],
         RW, A[m][k]);
  for (n=k+1; n<N; n++) {
    SYRK(R,  A[n][k],
         RW, A[n][n]);
    for (m=n+1; m<N; m++)
      GEMM(R,  A[m][k],
           R,  A[n][k],
           RW, A[m][n]);
  }
}
```

PTG programming (Code based)

```
TRSM(k, m)




// Flows & their dependencies
READ  A <- A POTRF(k)
RW    C <- (k == 0) ? dataA(m, k)
        <- (k != 0) ? C GEMM(k-1, m, k)
        -> A SYRK(k, m)
        -> A GEMM(k, m, k+1..m-1)
        -> B GEMM(k, m+1..NT-1, m)
        -> dataA(m, k)
```

```
for (k=0; k<N; k++) {
  POTRF(RW, A[k][k]);
  for (m=k+1; m<N; m++)
    TRSM(R,  A[k][k],
         RW, A[m][k]);
  for (n=k+1; n<N; n++) {
    SYRK(R,  A[n][k],
         RW, A[n][n]);
    for (m=n+1; m<N; m++)
      GEMM(R,  A[m][k],
           R,  A[n][k],
           RW, A[m][n]);
  }
}
```

PTG programming (Code based)

```
TRSM(k, m)

// Execution space
k = 0   .. NT-1
m = k+1 .. NT-1




// Flows & their dependencies
READ  A <- A POTRF(k)
RW    C <- (k == 0) ? dataA(m, k)
        <- (k != 0) ? C GEMM(k-1, m, k)
        -> A SYRK(k, m)
        -> A GEMM(k, m, k+1..m-1)
        -> B GEMM(k, m+1..NT-1, m)
        -> dataA(m, k)
```

```
for (k=0; k<N; k++) {
  POTRF(RW, A[k][k]);
  for (m=k+1; m<N; m++)
    TRSM(R,  A[k][k],
         RW, A[m][k]);
  for (n=k+1; n<N; n++) {
    SYRK(R,  A[n][k],
         RW, A[n][n]);
    for (m=n+1; m<N; m++)
      GEMM(R,  A[m][k],
           R,  A[n][k],
           RW, A[m][n]);
  }
}
```

PTG programming (Code based)

```
TRSM(k, m)

// Execution space
k = 0   .. NT-1
m = k+1 .. NT-1

// Partitioning
: dataA(m, k)

// Flows & their dependencies
READ  A <- A POTRF(k)
RW    C <- (k == 0) ? dataA(m, k)
        <- (k != 0) ? C GEMM(k-1, m, k)
        -> A SYRK(k, m)
        -> A GEMM(k, m, k+1..m-1)
        -> B GEMM(k, m+1..NT-1, m)
        -> dataA(m, k)
```

```
for (k=0; k<N; k++) {
  POTRF(RW, A[k][k]);
  for (m=k+1; m<N; m++)
    TRSM(R,  A[k][k],
         RW, A[m][k]);
  for (n=k+1; n<N; n++) {
    SYRK(R,  A[n][k],
         RW, A[n][n]);
    for (m=n+1; m<N; m++)
      GEMM(R,  A[m][k],
           R,  A[n][k],
           RW, A[m][n]);
  }
}
```

PTG programming (Code based)

```
TRSM(k, m)

// Execution space
k = 0   .. NT-1
m = k+1 .. NT-1

// Partitioning
: dataA(m, k)

// Flows & their dependencies
READ  A <- A POTRF(k)
RW    C <- (k == 0) ? dataA(m, k)
        <- (k != 0) ? C GEMM(k-1, m, k)
        -> A SYRK(k, m)
        -> A GEMM(k, m, k+1..m-1)
        -> B GEMM(k, m+1..NT-1, m)
        -> dataA(m, k)

BODY
  trsm(A, C);
END
```

```
for (k=0; k<N; k++) {
  POTRF(RW, A[k][k]);
  for (m=k+1; m<N; m++)
    TRSM(R,  A[k][k],
         RW, A[m][k]);
  for (n=k+1; n<N; n++) {
    SYRK(R,  A[n][k],
         RW, A[n][n]);
    for (m=n+1; m<N; m++)
      GEMM(R,  A[m][k],
           R,  A[n][k],
           RW, A[m][n]);
  }
}
```

PTG programming (Code based)

```
TRSM(k, m)

// Execution space
k = 0   .. NT-1
m = k+1 .. NT-1

// Partitioning
: dataA(m, k)

// Flows & their dependencies
READ  A <- A POTRF(k) [type = LOWER]
RW    C <- (k == 0) ? dataA(m, k)
        <- (k != 0) ? C GEMM(k-1, m, k)
        -> A SYRK(k, m)
        -> A GEMM(k, m, k+1..m-1)
        -> B GEMM(k, m+1..NT-1, m)
        -> dataA(m, k)

BODY
  trsm(A, C);
END
```

```
for (k=0; k<N; k++) {
  POTRF(RW, A[k][k]);
  for (m=k+1; m<N; m++)
    TRSM(R,  A[k][k],
         RW, A[m][k]);
  for (n=k+1; n<N; n++) {
    SYRK(R,  A[n][k],
         RW, A[n][n]);
    for (m=n+1; m<N; m++)
      GEMM(R,  A[m][k],
           R,  A[n][k],
           RW, A[m][n]);
  }
}
```

3

Advanced usage

Nvidia CUDA body

```
BODY [type=CUDA weight=expression device=expression
      dyld=fct_prefix dyld_type=fct_type]
{
   /**
    * Code that will be executed on a CUDA stream
    * parsec_body.stream, on the GPU parsec_body.index
    */
}
END
```

type	The type keyword for a GPU kernel is CUDA
weight	(Optional) Gives a hint to the static scheduler on the number of tasks that will be applied on the RW flow in a serie
device	(Optional) Gives a hint to the scheduler to decide on which device the kernel should be scheduled. (Default is -1)
dyld	(Optional) Specifies a function name prefix to look for in case of dynamic search. Allows for ld_preload the functions. If the function is not found, the body is disabled, otherwise the variable `parsec_body.dyld_fn` points to the function.
dyld_type	(Optional) Defines the function prototype

Nvidia CUDA body / device property

The device property of a CUDA body will help the scheduler to choose on which device, GPU or not, execute the kernel:

- < -1 The CUDA body will be skipped for this specific task, and the engine will try the next body in the list
- $>= 0$ This specifies a given GPU for this body. If device is larger than the number of GPU, then a modulo with the total number of CUDA devices is applied
- -1 This is default value. The runtime will automatically decides which GPU is the best fitted for this task, or to move forward to the next body.
  The actual policy is based: 1) on the locality of one the inout data; 2) on the less loaded device. The task weight is set accordingly to the device performance, and multiply by the optional weight of the task to take into account the following tasks that will be scheduled on the same device based on data locality.

The device property can be given by value, or through an inline function.

Recursive Body

```
BODY [type=RECURSIVE]
{
  /**
   * Code that will generate a new local DAG working on subparts of the
   * current flows.
   */
  parsec_taskpool_t tp = parsec_SmallDAG_New( ... );
  parsec_recursivecall( context, (parsec_task_t*)this_task,
                        handle, parsec_SmallDAG_Destruct, ... );

  return PARSEC_HOOK_RETURN_ASYNC;
}
END
```

- The type keyword for a recursive kernel is RECURSIVE
- The current task completes only when all the sub-tasks are completed
- The sub-DAG is only known by the current process
- `parsec_recursive_call` function is an helper function to set the callback that will complete the current task
- Must return `PARSEC_HOOK_RETURN_ASYNC` to notify asynchronous completion of the task, `RSEC_HOOK_RETURN_NEXT` to forward the computation to the next body

4

Miscellaneous

PaRSEC

Website	http://icl.cs.utk.edu/parsec
Git	https://bitbucket.org/icldistcomp/parsec, Open to external contributors via pull requests
Licence	BSD
Documentation	▪ Wiki: https://bitbucket.org/icldistcomp/parsec/wiki/Home Documentation for compilation and contributors ▪ Doxygen: Internal structure documentation (under redaction)
Contacts	▪ Mailing list: dplasma-users@icl.utk.edu ▪ BitBucket Issues/Request tracker https://bitbucket.org/icldistcomp/parsec/issues
Credits	University of Tennessee, ICL Bordeaux INP - Inria - CNRS - Univ. de Bordeaux

References

1. PaRSEC: Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P., Dongarra, J. **"DAGuE: A Generic Distributed DAG Engine for High Performance Computing,"** *Parallel Computing*, T. Hoefler eds. Elsevier, Vol. 38, No 1-2, 27-51, 2012.
2. DPLASMA: Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Luszczek, P., Dongarra, J. **"Dense Linear Algebra on Distributed Heterogeneous Hardware with a Symbolic DAG Approach,"** *Scalable Computing and Communications: Theory and Practice*, Khan, S., Wang, L., Zomaya, A. eds. John Wiley & Sons, 699-735, March, 2013.

Example of projects using it

- DPLASMA: Dense Linear Algebra
  Runs tile algorithms (PLASMA) on top of the PaRSEC Engine
  Distributed within PaRSEC (UTK/ICL, Bdx INP/Inria/CNRS/Univ Bdx)
- PaSTiX: Sparse direct solver (Bdx INP/Inria/CNRS/Univ Bdx)
- DOMINO: 3D sweep for Neutron Transport Simulation (EDF)
- ALTA: Rational & Non-linear fitting of BRDFs (LP2N/Univ Montreal/CNRS/Inria)
- DiP: (Total)
- Eigenvalue problems (KAUST)
- NWChem[Ex]
- MADNESS, TiledArray

Thank you