Introduction to PaRSEC

Innovative Computing Laboratory
University of Tennessee
March 2016

George Bosilca
Aurelien Bouteiller
Anthony Danalis
Mathieu Faverge
Damien Genet
Amina Guermouche
Thomas Herault
Jack Dongarra
Motivation

✧ Why PaRSEC?
  ✧ Dataflow-based execution

✧ Why Dataflow-based execution?
  ✧ Freedom from Control Flow
  ✧ Express Algorithmic Dataflow NOT Explicit data movement

✧ Why?
  ✧ Scalability
  ✧ Performance at every scale
Motivation

✧ Why PaRSEC?
  ✧ Dataflow-based execution
✧ Why Dataflow-based execution?
  ✧ Freedom from Control Flow
  ✧ Express Algorithmic Dataflow NOT Explicit data movement
Why PaRSEC?
- Dataflow-based execution

Why Dataflow-based execution?
- Freedom from Control Flow
- Express Algorithmic Dataflow NOT Explicit data movement

Why?
- Scalability
- Performance at every scale
Tile QR Algorithm

- GEQRT
- TSQRT
- UNMQR
- TSMQR
FOR $k = 0 .. \text{SIZE} - 1$

$A[k][k] := \text{GEQRT}( A[k][k] )$

FOR $m = k+1 .. \text{SIZE} - 1$

$(A[k][k]|\text{UP}, A[m][k]) := \text{TSQRT}( A[k][k]|\text{UP}, A[m][k] )$

FOR $n = k+1 .. \text{SIZE} - 1$


FOR $m = k+1 .. \text{SIZE} - 1$

$(A[k][n], A[m][n]) := \text{TSMQR}( A[m][k], A[k][n], A[m][n] )$
FOR $k = 0 .. \text{SIZE} - 1$

$A[k][k] := \text{GEQRT}( A[k][k] )$

FOR $m = k+1 .. \text{SIZE} - 1$

$( A[k][k] \text{UP}, A[m][k] ) := \text{TSQRT}( A[k][k] \text{UP}, A[m][k] )$

FOR $n = k+1 .. \text{SIZE} - 1$


FOR $m = k+1 .. \text{SIZE} - 1$

$( A[k][n], A[m][n] ) := \text{TSMQR}( A[m][k], A[k][n], A[m][n] )$
Tile QR pseudocode

FOR k = 0 .. SIZE - 1

A[k][k] := GEQRT( A[k][k] )

FOR m = k+1 .. SIZE - 1


FOR n = k+1 .. SIZE - 1


FOR m = k+1 .. SIZE - 1

( A[k][n], A[m][n] ) := TSMQR( A[m][k], A[k][n], A[m][n] )

Problem with serial execution and BSP: We tell the computer what to do and exactly in what order to do it.
### Load Balance, Idle time & Jitter

#### SPMD / MPI

![Graph showing load balance, idle time, and jitter for SPMD / MPI](image)

- Minimum: 0
- Maximum: 1.1919e+07

#### PaRSEC

![Graph showing load balance, idle time, and jitter for PaRSEC](image)

- Minimum: 0
- Maximum: 7.46814e+06
What’s wrong with the serial/BSP code?

✗ It has:
   ✗ Control Flow
   ✗ No Dataflow, only hints for runtime to infer Dataflow
       ✗ High memory requirements or reduced parallelism

✓ It should have:
   ✓ No Control Flow
   ✓ Explicit Dataflow
Tile QR Algorithm

GEQRT
TSQRT
UNMQR
TSMQR
Tile QR Algorithm

GEQRT(k)

TSQRT(k,m)

UNMQR(k,n)

TSMQR(k,m,n)
FOR \( k = 0 \ldots \text{SIZE} - 1 \)

\[ A[k][k] := \text{GEQRT}( A[k][k] ) \]

FOR \( m = k+1 \ldots \text{SIZE} - 1 \)

\[ ( A[k][k]\text{UP}, A[m][k] ) := \text{TSQRT}( A[k][k]\text{UP}, A[m][k] ) \]

FOR \( n = k+1 \ldots \text{SIZE} - 1 \)

\[ A[k][n] := \text{UNMQR}( A[k][k]\text{LOW}, A[k][n] ) \]

FOR \( m = k+1 \ldots \text{SIZE} - 1 \)

\[ ( A[k][n], A[m][n] ) := \text{TSMQR}( A[m][k], A[k][n], A[m][n] ) \]
Tile QR Algorithm

- GEQRT(k)
  - $k = 0 .. MT-1$

- TSQRT(k,m)
  - $k = 0 .. MT-1$
  - $m = k+1 .. MT-1$

- UNMQR(k,n)

- TSMQR(k,m,n)
Tile QR Algorithm

- **GEQRT(k)**
  - $k = 0 \ldots MT-1$
  - $\rightarrow$ **TSQRT(k,k+1)**

- **TSQRT(k,m)**
  - $k = 0 \ldots MT-1$
  - $m = k+1 \ldots MT-1$

- **UNMQR(k,n)**

- **TSMQR(k,m,n)**
Parameterized Task Graph (PTG)

✓ Task Classes w/ parameters
  ✓ geqrt(k), tsqrt(k,m), unmqr(k,n), tsmqr(k,n,m)

✓ Dataflow between Tasks

✓ Compressed form of the Execution DAG

✓ Fixed size (problem size independent)
Why not discover the whole DAG?
Why not discover the whole DAG?
Why not discover the whole DAG?
Why not discover the whole DAG?
Why not discover the whole DAG?

Memory Use Overhead (w/ window)
PaRSEC Tutorial

How to program with a PTG model?


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

March 7, 2016
Introduction
2

PaRSEC Examples
2.1 PaRSEC Examples

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?
1. Clone the last version of the bitbucket repository:
   git clone git@bitbucket.org:icldistcomp/parsec.git
2. Configuration through cmake
   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
   - DAGUE_DIST_WITH_MPI: Enable/Disable the compilation with MPI
   - DAGUE_GPU_WITH_CUDA: Enable/Disable the support for CUDA kernels
   - See INSTALL file and contrib directory for more information
3. Depends on: HwLoc, MPI, CUDA, Plasma
From .jdf to .c file

- `$PARSEC_DIR/bin/daguepp -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

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}`
Main data structure

dague_context_t

#include <dague.h>

dague_context_t* dague_init( int nb_cores, int* pargc, char** pargv[]);
int dague_fini( dague_context_t** pcontext );

- This is the main structure of the PaRSEC engine
- The new context will be passed as arguments of all the main functions
- **nb_cores** 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
Main data structure

dague_context_t

#include <dague.h>

int dague_context_start(dague_context_t *context);
int dague_context_test(dague_context_t *context);
int dague_context_wait(dague_context_t *context);

- **start** allows all the other threads to start executing. This call should be paired with test or wait.
- **test** checks the status of an ongoing execution, started with start, and returns 1 if the context has no more pending tasks, 0 otherwise.
- **wait** progresses the execution context until no further operations are available, and set the other threads to sleeping mode.
2.2

PaRSEC Examples

HelloWorld
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 then embarrassingly parallel, example
JDF Objects

```
#include <dague.h>

dague_JDFName_handle_t *dague_JDFName_new(...);
int dague_enqueue(dague_context_t*, dague_handle_t*);
void dague_handle_free(dague_handle_t *handle);
```

- This is the structure associated to each algorithm
- Each JDF has it’s own handle structure that inherits from the main `dague_handle_t`
- `dague_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 `dague_context_start` or `dague_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/Epilogue (optional):

- **Syntax:**
  - `extern "C" %{
    // Content
  }`

- Optional in the syntax

- Not compiled / Directly copy-paste in the generated `.c` file

- Allows to include personal headers

- Allows to define functions

- ...
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 $\geq 1$):
- A JDF file need at least one task
- No limit in the number of tasks
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.
TaskClass

Locals (required $\geq 1$):

- Contains the variables from the taskclass execution space
- Can be defined as a value or a range:
  
  $p1 = start \ldots end \ldots inc$

- Each local can be defined in function of the previously defined local variables

  $p2 = start \ldots p1 \ldots inc$

- Can be defined through a function

  $p3 = \text{inline}_{c} \%\{\text{return } f(a, b, p1 ); \%\}$

- Can be a range only if part of the execution space
  
  $\text{start}$ and $\text{end}$ bounds are both included in the execution space

- Maximum number of locals is defined by $\text{MAX\_LOCAL\_COUNT}$
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 dague_ddesc_t structure
- Takes parameters from the Private, or Local variables
- Not possible to give directly an integer (rank/vpid)
- Can be dynamically changed only if everyone involved knows about the changes
TaskClass

TaskClassName( i, j, ... )

Locals

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

Flows (required $\geq 1$):
- 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

Body1
Body2
...
TaskClass

TaskClassName( i, j, ... )

Locals

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

Flow1
Flow2
...

/* Priority */
; priority

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

Body1
Body2
...
TaskClass

TaskClassName( i, j, ... )

Locals

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

Flow1
Flow2
...

/* Priority */
; priority

Body (required $\geq 1$):

- 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
- The code is copy/paste in a function that makes all the private, locals and flows available to it
- Any thread in the system belonging to the process, and the NUMA node defined by the partitioning can execute this task
2.3
PaRSEC Examples
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 dague_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
dague_JDFName_New() function prototype
- Implicitly includes all descriptors 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)

```plaintext
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

```c
#include <dague/arena.h>
int dague_arena_construct(dague_arena_t* arena,
    size_t elem_size,
    size_t alignment,
    dague_datatype_t opaque_dtt);

void dague_arena_destruct(dague_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: `DAGUE_ARENA_ALIGNMENT_[64b | INT | PTR | SSE | CL1 ]`
- `destruct` releases all the memory elements allocated
#include <dague/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**, `dague_datatype_xxx_t` with `xxx` among `int, int8, int16, int32, int64, uint8, uint16, uint32, uint64, float, double, long_double, complex, double_complex`

**Functions:**
dague_type_size, dague_type_create_contiguous, dague_type_create_vector,
dague_type_create_hvector, dague_type_create_indexed,
dague_type_create_indexed_block, dague_type_create_struct,
dague_type_create_resized
2.4 PaRSEC Examples
Distributed Chain
A distributed chain of tasks

Ex03_ChainMPI.jdf

- How to create a descriptor?
- How to specify the distribution, and the associated data?
- How to extend the existing descriptor structure?
#include <dague/data_distribution.h>

void dague_ddesc_init(dague_ddesc_t *d, int nodes, int myrank);
void dague_ddesc_destroy(dague_ddesc_t *d);

- This is the structure that is provided to the JDFs to describe the task and data distribution, and the data location
- init initializes the fields of the data structure to default values
- destroy cleans up the allocated data of the descriptor
- In most cases, it can not be used directly
**rank_of / vpid_of / data_of functions**

```c
#include <dague/data_distribution.h>

uint32_t (*rank_of)(dague_ddesc_t *d, ...);
int32_t (*vpid_of)(dague_ddesc_t *d, ...);
dague_data_t* (*data_of)(dague_ddesc_t *d, ...);
```

- Each descriptor contains a set of functions that describes the data distribution and location
- Each function takes as parameter the pointer to the descriptor 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 `dague_data_t` structure that describes the piece of data associated to the given parameters
dague_ddesc_t / Own descriptor

```c
#include <dague/data_distribution.h>

typedef struct my_ddesc_s {
    dague_ddesc_t super;
    ...
} my_ddesc_t;
```

- If information need to be stored in the descriptor, then a personal descriptor *inheriting* from the main structure can be defined.
- The parent type, dague_ddesc_t must be the first field
2.5 PaRSEC Examples

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 descriptor?
#include <dague/data_distribution.h>
dague_data_t *
dague_data_create( dague_data_t **holder, 
                 dague_ddesc_t *desc, 
                 dague_data_key_t key, void *ptr, size_t size );

void
dague_data_destroy( dague_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 descriptor desc, and key is its unique identifier in this descriptor
- key is a unique identifier of the elementary piece of data
- destroy frees the existing copies of the data, and the structure
2.6
PaRSEC Examples
Broadcast
Broadcast an information

Ex05_Broadcast.jdf

- How to hide a private variable from the dague_JDFName_new() prototype?
- How to broadcast an information from one task to many others?
2.7
PaRSEC Examples
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?
2.8

PaRSEC Examples

Control Flows
CTL flows

Ex07_RAW_CTL.jdf

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

PaRSEC Team – JDF tutorial – 2. PaRSEC Examples
CTL Flows

CTL \( ct1 \leftarrow ct2 \text{ Task1}(0 .. k) \)
\( \leftarrow (m == 0) ? ct1 \text{ Task2}(k) \)
\( \leftarrow (m == 1) ? \)
\( \rightarrow ct1 \text{ Task1}(m, n) \)
\( \rightarrow ct2 \text{ TaskA}(m .. n) \)

- 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
2.9

PaRSEC Examples

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
3

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
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++) {
        for (m=n+1; m<NT; m++) {
        }
    }
}

- Move from scalar to matrix operation ($a \rightarrow A$)
- 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)

\[ \text{TRSM}(k, m) \]
PTG programming (DAG based)

TRSM(k, m)

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

TRSM in Cholesky

GEMM
SYRK
TRSM
POTRF
PTG programming (DAG based)

\textbf{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)

 BODY
trsm(A, C);
END
PTG programming (DAG based)

\textbf{TRSM}(k, m)

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

\textbf{TRSM in Cholesky}

\[ \begin{array}{c}
\text{READ} & A \leftarrow A \textbf{POTRF}(k) \\
\text{RW} & C \leftarrow (k \text{ == 0}) \ ? \ \text{dataA}(m, k) \\
& \leftarrow (k \text{ != 0}) \ ? \ C \ \textbf{GEMM}(k-1, m, k) \\
& \rightarrow A \ \textbf{SYRK}(k, m)
\end{array} \]
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)

TRSM in Cholesky
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

// 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)

\[ \text{TRSM}(k, m) \]

// Execution space
\[
\begin{align*}
    k &= 0 \ldots NT-1 \\
    m &= k+1 \ldots NT-1
\end{align*}
\]

// Flows & their dependencies
READ  \( A \leftarrow \text{A POTRF}(k) \)
RW    \( C \leftarrow (k == 0) \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}(m, k) \)
      \( \text{ ? dataA}
PTG programming (DAG based)

\( \text{TRSM}(k, m) \)

// Execution space
\( k = 0 \ldots NT-1 \)
\( m = k+1 \ldots NT-1 \)

// 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)

\[ \text{TRSM}(k, m) \]

// Execution space
\[ k = 0 \ldots NT-1 \]
\[ m = k+1 \ldots NT-1 \]

// Partitioning
: dataA(m, k)

// Flows & their dependencies
READ \[ A \leftarrow A \text{ POTRF}(k) \]
RW \[ C \leftarrow (k == 0) \ ? \text{ dataA}(m, k) \]
\[ \langle k \neq 0 \rangle \ ? \text{ GEMM}(k-1, m, k) \]
\[ \rightarrow A \text{ SYRK}(k, m) \]
\[ \rightarrow A \text{ GEMM}(k, m, k+1..m-1) \]
\[ \rightarrow B \text{ GEMM}(k, m+1..NT-1, m) \]
\[ \rightarrow \text{ dataA}(m, k) \]

BODY
trsm(A, C);
END
PTG programming (DAG based)

\textbf{TRSM}(k, m)

// Execution space
\begin{align*}
k &= 0 \ldots NT-1 \\
m &= k+1 \ldots NT-1
\end{align*}

// Partitioning
: dataA(m, k)

// Flows & their dependencies
\begin{align*}
\text{READ} &\quad A \leftarrow A \ \text{POTRF}(k) \ [\text{type = LOWER}] \\
\text{RW} &\quad C \leftarrow (k == 0) \ ? \ \text{dataA}(m, k) \\
&\quad \quad \leftarrow (k != 0) \ ? \ C \ \text{GEMM}(k-1, m, k) \\
&\quad \quad \quad \rightarrow A \ \text{SYRK}(k, m) \\
&\quad \quad \quad \rightarrow A \ \text{GEMM}(k, m, k+1..m-1) \\
&\quad \quad \quad \rightarrow B \ \text{GEMM}(k, m+1..NT-1, m) \\
&\quad \quad \quad \rightarrow \ \text{dataA}(m, k)
\end{align*}

\textbf{BODY}
\begin{align*}
\text{trsm}(A, C);
\end{align*}

\textbf{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]);
    }
}

TRSM in Cholesky
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]);
    }
}

TRSM in Cholesky

PTG programming (Code based)
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)
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**

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

```plaintext
// 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)

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]);
  }
}

TRSM in Cholesky
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]);
}
}

TRSM in Cholesky
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]);
    }
}
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]);
   }

PaRSEC Team – JDF tutorial – 3. PTG Cholesky
Advanced usage
**Nvidia CUDA body**

```c
BODY [type=CUDA weight=expression device=expression
dyld=fct_prefix dyld_type=fct_type]
{
    /**
     * Code that will be executed on a CUDA stream
     * dague_body.stream, on the GPU dague_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 dague_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.
     */
    dague_handle_t handle = dague_SmallDAG_New( ... );
    dague_recursivecall( context, (dague_execution_context_t*)this_task,
                         handle, dague_SmallDAG_Destruct, ... );

    return DAGUE_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
- dague_recursive_call function is an helper function to set the callback that will complete the current task
- Must return DAGUE_HOOK_RETURN_ASYNC to notify asynchronous completion of the task, or DAGUE_HOOK_RETURN_NEXT to forward the computation to the next body
5

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)
- This tutorial :)

Contacts
- Mailing list: dplasma-users@eecs.utk.edu
- BitBucket Issues/Request tracker
  https://bitbucket.org/icldistcomp/parsec/issues

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

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)
Thank you

PaRSEC

Web Site: http://icl.cs.utk.edu/parsec/
Git Repository: https://bitbucket.org/icldistcomp/parsec
BSD License