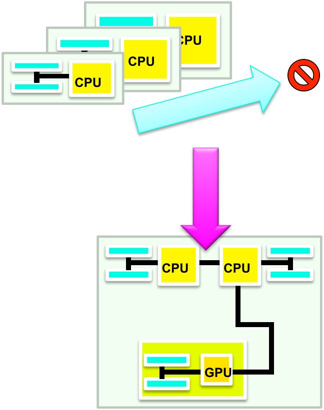
1

Introduction

Introduction

Parallel Multicore Architectures

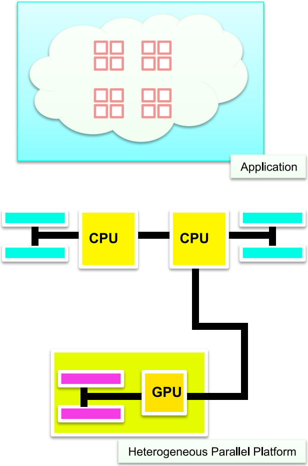
- Increasingly widespread
- Increasingly dense
- Increasingly diverse
  - Specialized cores
  - Heterogeneity



Heterogeneous Parallel Platforms

Heterogeneous Association

- General purpose processor
- Specialized accelerator



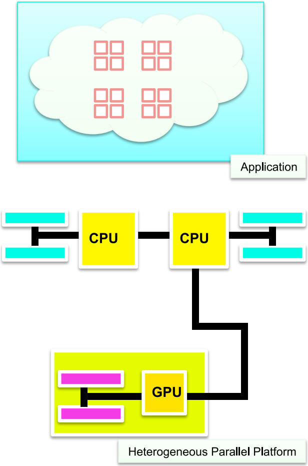
Heterogeneous Parallel Platforms

Heterogeneous Association

- General purpose processor
- Specialized accelerator

Generalization

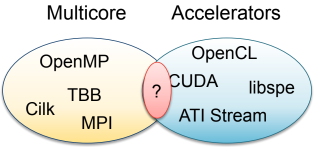
- Combination of various units
  - Latency-optimized cores
  - Throughput-optimized cores
  - Energy-optimized cores
- Distributed cores
  - Standalone GPUs
  - Intel Xeon Phi (MIC)
  - Intel Single-Chip Cloud (SCC)
- Integrated cores
  - Intel Haswell
  - AMD Fusion
  - nVidia Tegra
  - ARM big.LITTLE

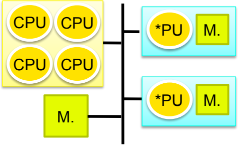


Programming Models

How to Program these architectures?

- Multicore programming
  - pthreads, OpenMP, TBB, ...
- Accelerator programming
  - Consensus on OpenCL?
  - (Often) Pure offloading model
- Hybrid models?
  - Take advantage of all resources
  - Complex interactions





Work Needed at Multiple Levels

- Applications
  - Programming paradigm
  - BLAS kernels, FFT, ...
- Compilers
  - Languages
  - Code generation/optimization
- **Runtime systems**
  - Resources management
  - Heterogeneous Task scheduling
- Architecture
  - Memory interconnect

Heterogeneous Task Scheduling

Scheduling on platform equipped with accelerators

- Adapting to heterogeneity
  - Decide about tasks to offload
  - Decide about tasks to keep on CPU

Heterogeneous Task Scheduling

Scheduling on platform equipped with accelerators

- Adapting to heterogeneity
  - Decide about tasks to offload
  - Decide about tasks to keep on CPU
- Communicate with discrete accelerator board(s)
  - Send computation requests
  - Send data to be processed
  - Fetch results back
  - Expensive

Heterogeneous Task Scheduling

Scheduling on platform equipped with accelerators

- Adapting to heterogeneity
  - Decide about tasks to offload
  - Decide about tasks to keep on CPU
- Communicate with discrete accelerator board(s)
  - Send computation requests
  - Send data to be processed
  - Fetch results back
  - Expensive
- Decide about worthiness

2

StarPU Programming/Execution Models

Task Parallelism

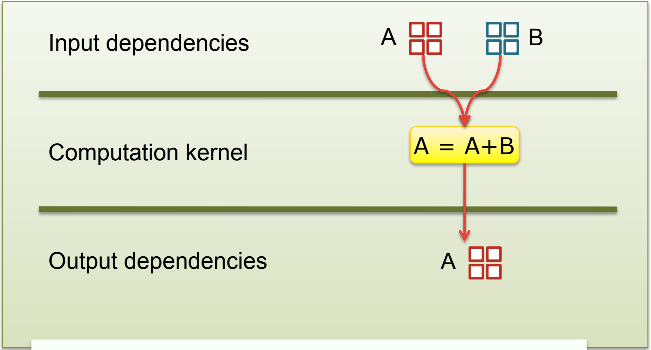
Principles

- Input dependencies
- Computation kernel
- Output dependencies

Task Parallelism

Principles

- Input dependencies
- Computation kernel
- Output dependencies



Task = an « elementary » computation + dependencies

StarPU Programming Model: Sequential Task Flow

- Express parallelism. . .

StarPU Programming Model: Sequential Task Flow

- Express parallelism. . .
- . . . using the natural program flow

StarPU Programming Model: Sequential Task Flow

- Express parallelism. . .
- . . . using the natural program flow

- **Submit** tasks in the sequential flow of the program. . .

StarPU Programming Model: Sequential Task Flow

- Express parallelism. . .
- . . . using the natural program flow

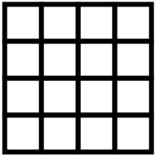
- **Submit** tasks in the sequential flow of the program. . .
- . . . then let the runtime schedule the tasks asynchronously

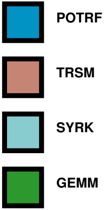
Ex.: The Sequential **Task-Based** Cholesky Decomposition

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

Ex.: The Sequential Task-Based Cholesky Decomposition

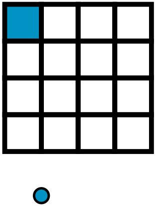
```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
  }
}
__wait__();
```

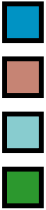




Ex.: The Sequential **Task-Based** Cholesky Decomposition

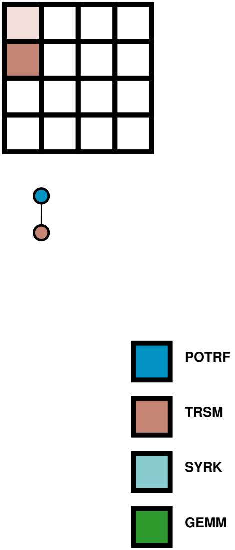
```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
                  R,A[i][j], R,A[k][j]);
  }
}
__wait__();
```





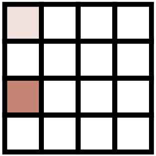
Ex.: The Sequential **Task-Based** Cholesky Decomposition

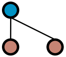
```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
  }
}
__wait__();
```

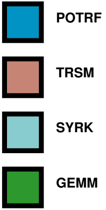


Ex.: The Sequential **Task-Based** Cholesky Decomposition

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

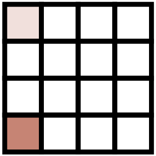


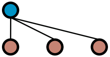


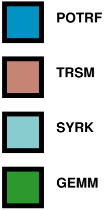


Ex.: The Sequential **Task-Based** Cholesky Decomposition

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

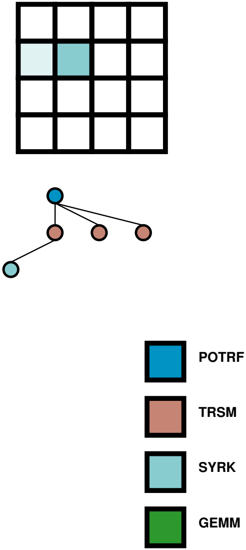






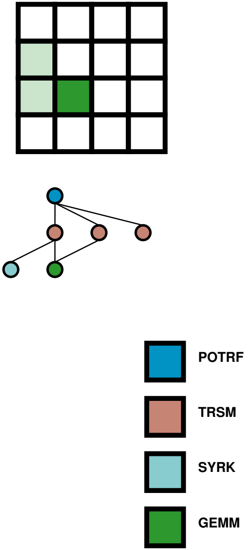
Ex.: The Sequential **Task-Based** Cholesky Decomposition

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



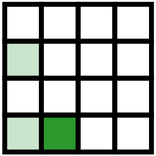
Ex.: The Sequential Task-Based Cholesky Decomposition

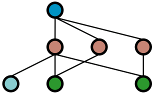
```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
  }
}
__wait__();
```

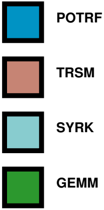


Ex.: The Sequential **Task-Based** Cholesky Decomposition

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

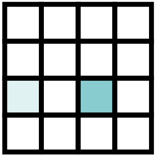


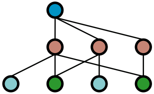


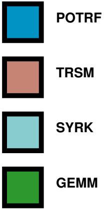


Ex.: The Sequential Task-Based Cholesky Decomposition

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

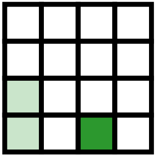


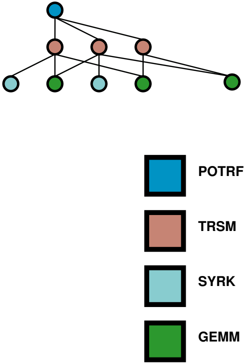




Ex.: The Sequential Task-Based Cholesky Decomposition

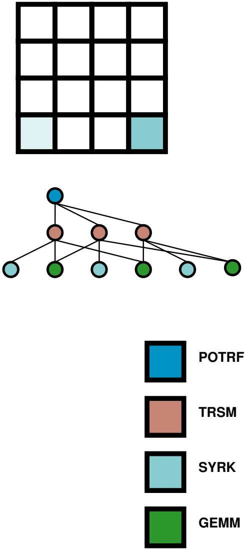
```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
  }
}
__wait__();
```





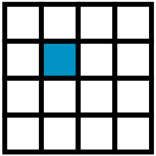
Ex.: The Sequential Task-Based Cholesky Decomposition

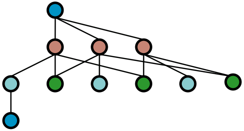
```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
               R,A[i][j], R,A[k][j]);
  }
}
__wait__();
```

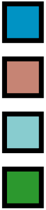


Ex.: The Sequential **Task-Based** Cholesky Decomposition

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

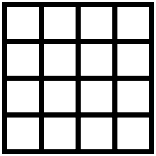


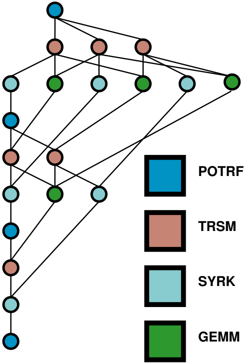




Ex.: The Sequential Task-Based Cholesky Decomposition

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



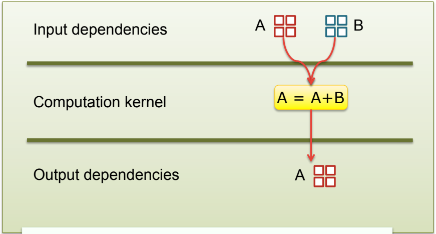


Task Relationships

Abstract Application Structure

Task Relationships

Abstract Application Structure

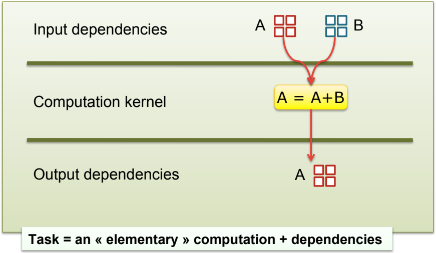


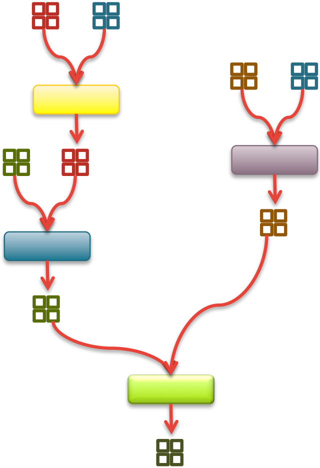
Task = an « elementary » computation + dependencies

Task Relationships

Abstract Application Structure

- Directed Acyclic Graph (DAG)





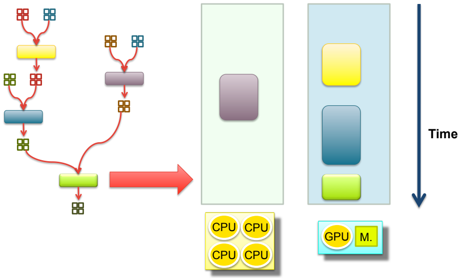
StarPU Execution Model: Task Scheduling

Mapping the graph of tasks (DAG) on the hardware

StarPU Execution Model: Task Scheduling

Mapping the graph of tasks (DAG) on the hardware

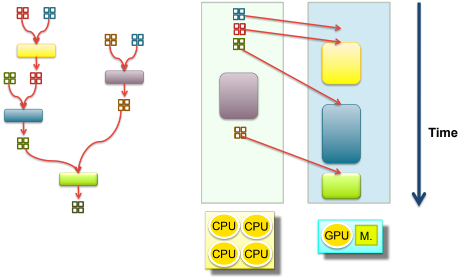
- Allocating computing resources



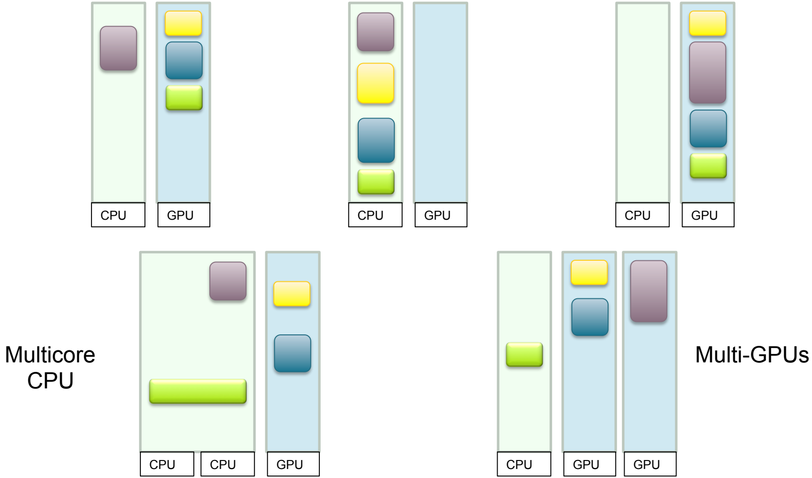
StarPU Execution Model: Task Scheduling

Mapping the graph of tasks (DAG) on the hardware

- Allocating computing resources
- Enforcing dependency constraints
- Handling data transfers



A Single DAG for Multiple Schedules, Platforms



StarPU in a Nutshell

Rationale

- Implement the sequential task flow programming model
- Map computations on heterogeneous computing units

StarPU in a Nutshell

Rationale

- Implement the sequential task flow programming model
- Map computations on heterogeneous computing units

Programming Model

- Task
- Data
- Relationships
  - Task $\leftrightarrow$ Task
  - Task $\leftrightarrow$ Data

StarPU in a Nutshell

Rationale

- Implement the sequential task flow programming model
- Map computations on heterogeneous computing units

Programming Model

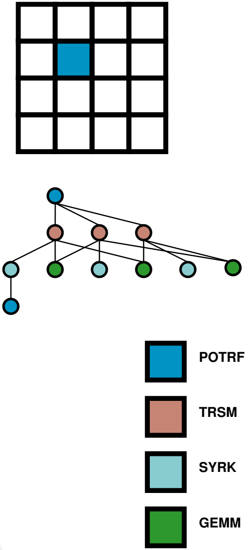
- Task
- Data
- Relationships
  - Task $\leftrightarrow$ Task
  - Task $\leftrightarrow$ Data

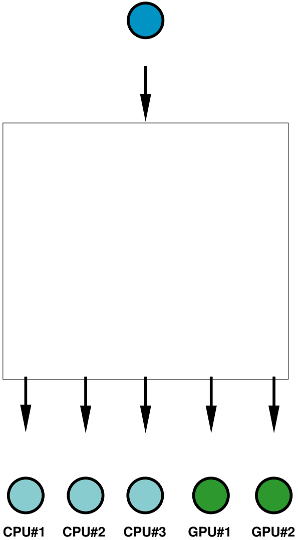
Runtime System

- Heterogeneous Task scheduling
- Application Programming Interface (Library)

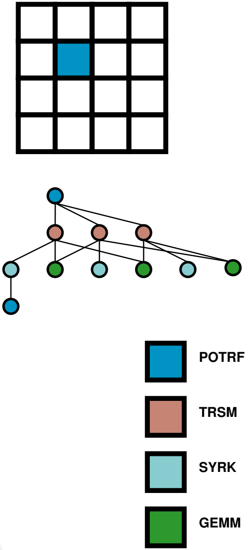
What StarPU can do for You?

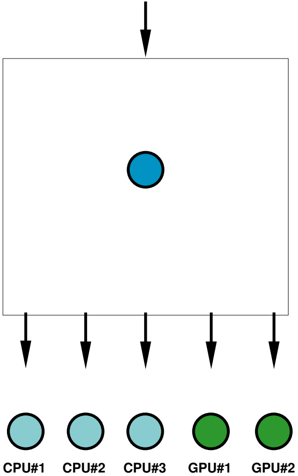
What StarPU does for You: Heterogeneous Task Scheduling



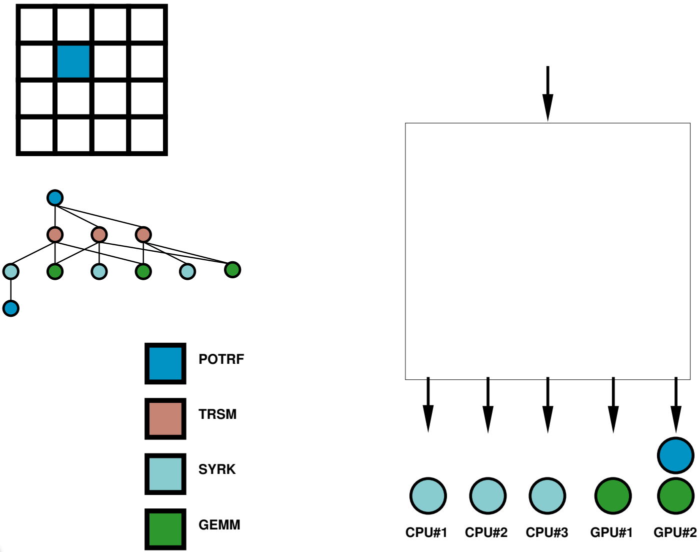


What StarPU does for You: Heterogeneous Task Scheduling





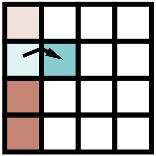
What StarPU does for You: Heterogeneous Task Scheduling

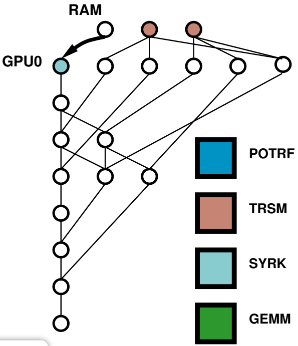


What StarPU does for You:

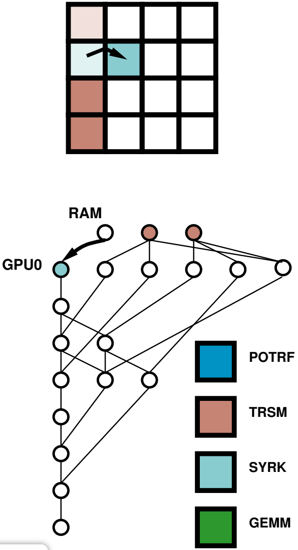
What StarPU does for You: Data Transfers

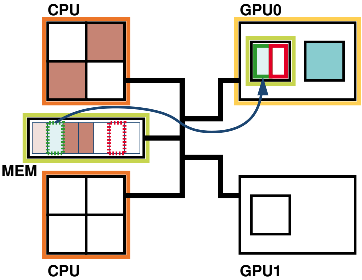
What StarPU does for You: Data Transfers





What StarPU does for You: Data Transfers





3

Programming with StarPU

Basic Example: Scaling a Vector

```c
float factor = 3.14;
float vector[NX];
```

Basic Example: Scaling a Vector

```
float factor = 3.14;
float vector[NX];
starpu_data_handle_t vector_handle;
```

Basic Example: Scaling a Vector

```
float factor = 3.14;
float vector[NX];
starpu_data_handle_t vector_handle;

/* ... fill vector ... */

starpu_vector_data_register(&vector_handle, 0,
                            (uintptr_t)vector, NX, sizeof(vector[0]));
```

Basic Example: Scaling a Vector

```c
float factor = 3.14;
float vector[NX];
starpu_data_handle_t vector_handle;

/* ... fill vector ... */

starpu_vector_data_register(&vector_handle, 0,
                            (uintptr_t)vector, NX, sizeof(vector[0]));

starpu_task_insert(
                   &scal_cl,
                   STARPU_RW, vector_handle,
                   STARPU_VALUE, &factor, sizeof(factor),
                   0);
```

Basic Example: Scaling a Vector

```
float factor = 3.14;
float vector[NX];
starpu_data_handle_t vector_handle;

/* ... fill vector ... */

starpu_vector_data_register(&vector_handle, 0,
                    (uintptr_t)vector, NX, sizeof(vector[0]));

starpu_task_insert(
                &scal_cl,
                STARPU_RW, vector_handle,
                STARPU_VALUE, &factor, sizeof(factor),
                0);

starpu_task_wait_for_all();
```

Basic Example: Scaling a Vector

```
float factor = 3.14;
float vector[NX];
starpu_data_handle_t vector_handle;

/* ... fill vector ... */

starpu_vector_data_register(&vector_handle, 0,
                            (uintptr_t)vector, NX, sizeof(vector[0]));

starpu_task_insert(
                    &scal_cl,
                    STARPU_RW, vector_handle,
                    STARPU_VALUE, &factor, sizeof(factor),
                    0);

starpu_task_wait_for_all();
starpu_data_unregister(vector_handle);

/* ... display vector ... */
```

Terminology

- Codelet
- Task
- Data handle

Definition: A Codelet

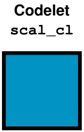
A **Codelet**. . .

- . . . relates an abstract computation kernel to its implementation(s)
- . . . can be instantiated into one or more **tasks**
- . . . defines characteristics common to a set of **task**s

Definition: A Codelet

A **Codelet**. . .

- . . . relates an abstract computation kernel to its implementation(s)
- . . . can be instantiated into one or more **tasks**
- . . . defines characteristics common to a set of **task**s



Definition: A Codelet

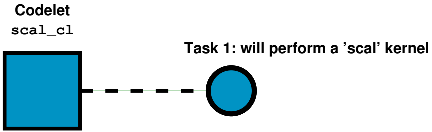
A **Codelet**. . .

- . . . relates an abstract computation kernel to its implementation(s)
- . . . can be instantiated into one or more **tasks**
- . . . defines characteristics common to a set of **task**s

Definition: A Codelet

A **Codelet**...

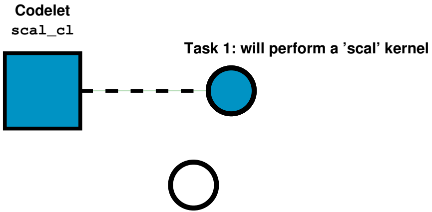
- ... relates an abstract computation kernel to its implementation(s)
- ... can be instantiated into one or more **tasks**
- ... defines characteristics common to a set of **task**s



Definition: A Codelet

A **Codelet**. . .

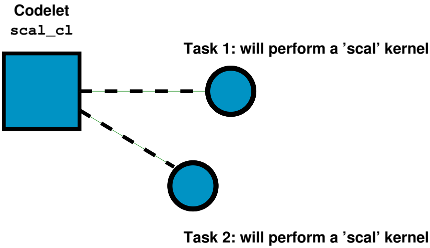
- . . . relates an abstract computation kernel to its implementation(s)
- . . . can be instantiated into one or more **tasks**
- . . . defines characteristics common to a set of **task**s



Definition: A Codelet

A **Codelet**. . .

- . . . relates an abstract computation kernel to its implementation(s)
- . . . can be instantiated into one or more **tasks**
- . . . defines characteristics common to a set of **task**s



Definition: A Task

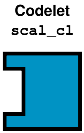
A **Task**. . .

- . . . is an instantiation of a **Codelet**
- . . . atomically executes a kernel from its beginning to its end
- . . . receives some input
- . . . produces some output

Definition: A Task

A **Task**. . .

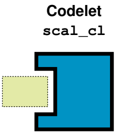
- . . . is an instantiation of a **Codelet**
- . . . atomically executes a kernel from its beginning to its end
- . . . receives some input
- . . . produces some output



Definition: A Task

A **Task**. . .

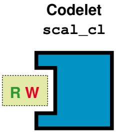
- . . . is an instantiation of a **Codelet**
- . . . atomically executes a kernel from its beginning to its end
- . . . receives some input
- . . . produces some output



Definition: A Task

A **Task**. . .

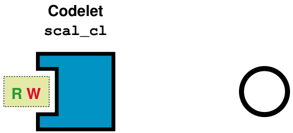
- . . . is an instantiation of a **Codelet**
- . . . atomically executes a kernel from its beginning to its end
- . . . receives some input
- . . . produces some output



Definition: A Task

A **Task**. . .

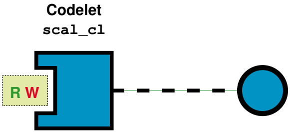
- . . . is an instantiation of a **Codelet**
- . . . atomically executes a kernel from its beginning to its end
- . . . receives some input
- . . . produces some output



Definition: A Task

A **Task**. . .

- . . . is an instantiation of a **Codelet**
- . . . atomically executes a kernel from its beginning to its end
- . . . receives some input
- . . . produces some output



Definition: A Task

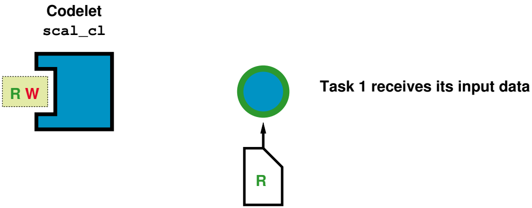
A **Task**. . .

- . . . is an instantiation of a **Codelet**
- . . . atomically executes a kernel from its beginning to its end
- . . . receives some input
- . . . produces some output

Definition: A Task

A **Task**. . .

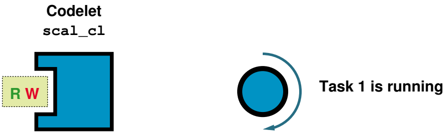
- . . . is an instantiation of a **Codelet**
- . . . atomically executes a kernel from its beginning to its end
- . . . receives some input
- . . . produces some output



Definition: A Task

A **Task**. . .

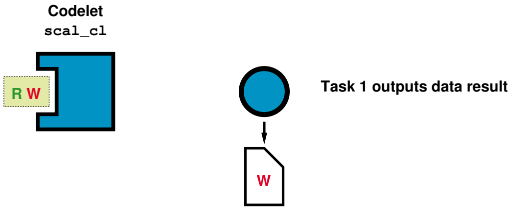
- . . . is an instantiation of a **Codelet**
- . . . atomically executes a kernel from its beginning to its end
- . . . receives some input
- . . . produces some output



Definition: A Task

A **Task**. . .

- . . . is an instantiation of a **Codelet**
- . . . atomically executes a kernel from its beginning to its end
- . . . receives some input
- . . . produces some output



Definition: A Data Handle

A **Data Handle**. . .

- . . . designates a piece of data managed by StarPU
- . . . is typed (vector, matrix, etc.)
- . . . can be passed as input/output for a **Task**

Elementary API

- Initializing/Ending a StarPU session
- Declaring a codelet
- Declaring and Managing Data
- Writing a Kernel Function
- Submitting a task
- Waiting for submitted tasks

Initializing a StarPU Session

- **starpu_init**(**struct starpu_conf** $*$configuration)

Initializing a StarPU Session

- **starpu_init**(**struct starpu_conf** $*$configuration)
  - The **struct starpu_conf** can be used to configure StarPU settings
  - Specify NULL for default settings

Initializing a StarPU Session

- **starpu_init**(**struct starpu_conf** ∗configuration)
  - The **struct starpu_conf** can be used to configure StarPU settings
  - Specify NULL for default settings

```c
#include <starpu.h>

int ret = starpu_init(NULL);

if (ret == 0) {
	printf("StarPU␣successfully␣initialized\n");
} else {
	fprintf(stderr, "StarPU␣initialization␣failed\n");
	exit(1);
}

/* StarPU is ready */
...
```

Ending a StarPU Session

- **starpu_shutdown**()

Ending a StarPU Session

- **starpu_shutdown**()

```
...
starpu_shutdown();

/* StarPU is terminated */
...
```

Declaring a Codelet

Define a **struct** starpu_codelet

```
struct starpu_codelet scal_cl = {
    ...
};
```

Declaring a Codelet

Define a **struct starpu_codelet**

- Plug the kernel function
  - Here: scal_cpu_func

```
struct starpu_codelet scal_cl = {
    .cpu_func  = { scal_cpu_func, NULL },
    ...
};
```

Declaring a Codelet

Define a **struct starpu_codelet**

- Plug the kernel function
  - Here: scal_cpu_func
- Declare the number of data pieces used by the kernel
  - Here: A single vector

```
struct starpu_codelet scal_cl = {
    .cpu_func  = { scal_cpu_func, NULL },
    .nbuffers = 1,
    ...
};
```

Declaring a Codelet

Define a **struct** **starpu_codelet**

- Plug the kernel function
  - Here: scal_cpu_func
- Declare the number of data pieces used by the kernel
  - Here: A single vector
- Declare how the kernel accesses the piece of data
  - Here: The vector is scaled in-place, thus R/W

```
struct starpu_codelet scal_cl = {
    .cpu_func  = { scal_cpu_func, NULL },
    .nbuffers = 1,
    .modes = { STARPU_RW },
};
```

Declaring and Managing Data

Put data under StarPU control

Declaring and Managing Data

Put data under StarPU control

- Initialize a piece of data

```
float vector[NX];
/* ... fill data ... */
```

Declaring and Managing Data

Put data under StarPU control

- Initialize a piece of data
- Register the piece of data and get a handle
  - The vector is now under StarPU control

```c
float vector[NX];
/* ... fill data ... */

starpu_data_handle_t vector_handle;
starpu_vector_data_register(&vector_handle, 0,
                        (uintptr_t)vector, NX, sizeof(vector[0]));
```

Declaring and Managing Data

Put data under StarPU control

- Initialize a piece of data
- Register the piece of data and get a handle
  - The vector is now under StarPU control
- Use data through the handle

```
float vector[NX];
/* ... fill data ... */

starpu_data_handle_t vector_handle;
starpu_vector_data_register(&vector_handle, 0,
                            (uintptr_t)vector, NX, sizeof(vector[0]));

/* ... use the vector through the handle ... */
```

Declaring and Managing Data

Put data under StarPU control

- Initialize a piece of data
- Register the piece of data and get a handle
  - The vector is now under StarPU control
- Use data through the handle
- Unregister the piece of data
  - The handle is destroyed
  - The vector is now back under user control

```c
float vector[NX];
/* ... fill data ... */

starpu_data_handle_t vector_handle;
starpu_vector_data_register(&vector_handle, 0,
                            (uintptr_t)vector, NX, sizeof(vector[0]));

/* ... use the vector through the handle ... */

starpu_data_unregister(vector_handle);
```

Writing a Kernel Function

- Every kernel function has the same C prototype

```c
void scal_cpu_func(void *buffers[], void *cl_arg) {
    ...
}
```

Writing a Kernel Function

- Every kernel function has the same C prototype
- Retrieve the vector's handle

```c
void scal_cpu_func(void *buffers[], void *cl_arg) {
    struct starpu_vector_interface *vector_handle = buffers[0];

    ...
}
```

Writing a Kernel Function

- Every kernel function has the same C prototype
- Retrieve the vector's handle
- Get vector's number of elements and base pointer

```c
void scal_cpu_func(void *buffers[], void *cl_arg) {
    struct starpu_vector_interface *vector_handle = buffers[0];

    unsigned n     = STARPU_VECTOR_GET_NX(vector_handle);
    float *vector = STARPU_VECTOR_GET_PTR(vector_handle);

    ...
}
```

Writing a Kernel Function

- Every kernel function has the same C prototype
- Retrieve the vector's handle
- Get vector's number of elements and base pointer
- Get the scaling factor as inline argument

```
void scal_cpu_func(void *buffers[], void *cl_arg) {
    struct starpu_vector_interface *vector_handle = buffers[0];

    unsigned n      = STARPU_VECTOR_GET_NX(vector_handle);
    float *vector = STARPU_VECTOR_GET_PTR(vector_handle);

    float *ptr_factor = cl_arg;

    ...
}
```

Writing a Kernel Function

- Every kernel function has the same C prototype
- Retrieve the vector's handle
- Get vector's number of elements and base pointer
- Get the scaling factor as inline argument
- Compute the vector scaling

```
void scal_cpu_func(void *buffers[], void *cl_arg) {
    struct starpu_vector_interface *vector_handle = buffers[0];

    unsigned n      = STARPU_VECTOR_GET_NX(vector_handle);
    float *vector = STARPU_VECTOR_GET_PTR(vector_handle);

    float *ptr_factor = cl_arg;

    unsigned i;
    for (i = 0; i < n; i++)
        vector[i] *= *ptr_factor;
}
```

Submitting a task

The **starpu_task_insert** call

- **Inserts** a task in the StarPU DAG

Submitting a task

The **starpu_task_insert** call

- **Inserts** a task in the StarPU DAG

Arguments

- The codelet structure

```
starpu_task_insert(&scal_cl
                   ...);
```

Submitting a task

The **starpu_task_insert** call

- **Inserts** a task in the StarPU DAG

Arguments

- The codelet structure
- The StarPU-managed data

```
starpu_task_insert(&scal_cl,
                    STARPU_RW, vector_handle,
                    ...);
```

Submitting a task

The **starpu_task_insert** call

- **Inserts** a task in the StarPU DAG

Arguments

- The codelet structure
- The StarPU-managed data
- The small-size inline data

```
starpu_task_insert(&scal_cl,
                   STARPU_RW, vector_handle,
                   STARPU_VALUE, &factor, sizeof(factor),
                   ...);
```

Submitting a task

The **starpu_task_insert** call

- **Inserts** a task in the StarPU DAG

Arguments

- The codelet structure
- The StarPU-managed data
- The small-size inline data
- 0 to mark the end of arguments

```
starpu_task_insert(&scal_cl,
                   STARPU_RW, vector_handle,
                   STARPU_VALUE, &factor, sizeof(factor),
                   0);
```

Submitting a task

The **starpu_task_insert** call

- **Inserts** a task in the StarPU DAG

Arguments

- The codelet structure
- The StarPU-managed data
- The small-size inline data
- 0 to mark the end of arguments

Notes

- The task is submitted non-blockingly

Submitting a task

The **starpu_task_insert** call

- **Inserts** a task in the StarPU DAG

Arguments

- The codelet structure
- The StarPU-managed data
- The small-size inline data
- 0 to mark the end of arguments

Notes

- The task is submitted non-blockingly
- Dependencies are enforced with previously submitted tasks' data. . .

Submitting a task

The **starpu_task_insert** call

- **Inserts** a task in the StarPU DAG

Arguments

- The codelet structure
- The StarPU-managed data
- The small-size inline data
- 0 to mark the end of arguments

Notes

- The task is submitted non-blockingly
- Dependencies are enforced with previously submitted tasks' data. . .
- . . . following the natural order of the program

Submitting a task

The **starpu_task_insert** call

- **Inserts** a task in the StarPU DAG

Arguments

- The codelet structure
- The StarPU-managed data
- The small-size inline data
- 0 to mark the end of arguments

Notes

- The task is submitted non-blockingly
- Dependencies are enforced with previously submitted tasks' data. . .
- . . . following the natural order of the program
- This is the **Sequential Task Flow Paradigm**

Waiting for Submitted Task Completion

- Tasks are submitted non-blockingly

Waiting for Submitted Task Completion

- Tasks are submitted non-blockingly

```
/* non-blocking task submits */
starpu_task_insert(...);
starpu_task_insert(...);
starpu_task_insert(...);
...
```

Waiting for Submitted Task Completion

- Tasks are submitted non-blockingly
- Wait for all submitted tasks to complete their work

```
/* non-blocking task submits */
starpu_task_insert(...);
starpu_task_insert(...);
starpu_task_insert(...);
...
```

Waiting for Submitted Task Completion

- Tasks are submitted non-blockingly
- Wait for all submitted tasks to complete their work

```
/* non-blocking task submits */
starpu_task_insert(...);
starpu_task_insert(...);
starpu_task_insert(...);
...

/* wait for all task submitted so far */
starpu_task_wait_for_all();
```

Basic Example: Scaling a Vector

```c
float factor = 3.14;
float vector[NX];
starpu_data_handle_t vector_handle;

/* ... fill vector ... */

starpu_vector_data_register(&vector_handle, 0,
                            (uintptr_t)vector, NX, sizeof(vector[0]));

starpu_task_insert(
                &scal_cl,
                STARPU_RW, vector_handle,
                STARPU_VALUE, &factor, sizeof(factor),
                0);

starpu_task_wait_for_all();
starpu_data_unregister(vector_handle);

/* ... display vector ... */
```

Heterogeneity: Device Kernels

Extending a codelet to handle heterogeneous platforms

Heterogeneity: Device Kernels

Extending a codelet to handle heterogeneous platforms

- Multiple kernel implementations for a CPU
  - SSE, AVX, ... optimized kernels

```
struct starpu_codelet scal_cl = {
    .cpu_func  = { scal_cpu_func,
            scal_sse_cpu_func, scal_avx_cpu_func, NULL },
    .nbuffers = 1,
    .modes = { STARPU_RW },
};
```

Heterogeneity: Device Kernels

Extending a codelet to handle heterogeneous platforms

- Multiple kernel implementations for a CPU
  - SSE, AVX, ... optimized kernels
- Kernels implementations for accelerator devices
  - OpenCL, NVidia Cuda kernels

```
struct starpu_codelet scal_cl = {
    .cpu_func  = { scal_cpu_func,
                   scal_sse_cpu_func, scal_avx_cpu_func, NULL },
    .opencl_func  = { scal_cpu_opencl, NULL },
    .cuda_func  = { scal_cpu_cuda, NULL },
    .nbuffers = 1,
    .modes = { STARPU_RW },
};
```

Writing a Kernel Function for CUDA

Writing a Kernel Function for CUDA

```c
extern "C" void scal_cuda_func(void *buffers[], void *cl_arg){
    struct starpu_vector_interface *vector_handle = buffers[0];
    unsigned n     = STARPU_VECTOR_GET_NX(vector_handle);
    float *vector = STARPU_VECTOR_GET_PTR(vector_handle);
    float *ptr_factor = cl_arg;

    ...




}
```

Writing a Kernel Function for CUDA

```
extern "C" void scal_cuda_func(void *buffers[], void *cl_arg){
    struct starpu_vector_interface *vector_handle = buffers[0];
    unsigned n      = STARPU_VECTOR_GET_NX(vector_handle);
    float *vector = STARPU_VECTOR_GET_PTR(vector_handle);
    float *ptr_factor = cl_arg;

    unsigned threads_per_block = 64;
    unsigned nblocks = (n+threads_per_block-1)/threads_per_block;

    ...

}
```

Writing a Kernel Function for CUDA

```
extern "C" void scal_cuda_func(void *buffers[], void *cl_arg){
    struct starpu_vector_interface *vector_handle = buffers[0];
    unsigned n      = STARPU_VECTOR_GET_NX(vector_handle);
    float *vector = STARPU_VECTOR_GET_PTR(vector_handle);
    float *ptr_factor = cl_arg;

    unsigned threads_per_block = 64;
    unsigned nblocks = (n+threads_per_block-1)/threads_per_block;

    vector_mult_cuda<<<nblocks,threads_per_block,0,
        starpu_cuda_get_local_stream()>>>(n,vector,*ptr_factor);
}
```

Writing a Kernel Function for CUDA

```
static __global__ void vector_mult_cuda(unsigned n,
                                        float *vector, float factor){
    unsigned i = blockIdx.x*blockDim.x + threadIdx.x;

    ...
}

extern "C" void scal_cuda_func(void *buffers[], void *cl_arg){
    struct starpu_vector_interface *vector_handle = buffers[0];
    unsigned n     = STARPU_VECTOR_GET_NX(vector_handle);
    float *vector = STARPU_VECTOR_GET_PTR(vector_handle);
    float *ptr_factor = cl_arg;

    unsigned threads_per_block = 64;
    unsigned nblocks = (n+threads_per_block-1)/threads_per_block;

    vector_mult_cuda<<<nblocks,threads_per_block,0,
        starpu_cuda_get_local_stream()>>>(n,vector,*ptr_factor);
}
```

Writing a Kernel Function for CUDA

```
static __global__ void vector_mult_cuda(unsigned n,
                                        float *vector, float factor){
    unsigned i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n)
        vector[i] *= factor;
}

extern "C" void scal_cuda_func(void *buffers[], void *cl_arg){
    struct starpu_vector_interface *vector_handle = buffers[0];
    unsigned n     = STARPU_VECTOR_GET_NX(vector_handle);
    float *vector = STARPU_VECTOR_GET_PTR(vector_handle);
    float *ptr_factor = cl_arg;

    unsigned threads_per_block = 64;
    unsigned nblocks = (n+threads_per_block-1)/threads_per_block;

    vector_mult_cuda<<<nblocks,threads_per_block,0,
        starpu_cuda_get_local_stream()>>>(n,vector,*ptr_factor);
}
```