Multicore Asynchronous Runtime Environment

Documentation and Interface Specification

HT80-NG608-1 E

April 28, 2014

Submit technical questions at:

mare@qti.qualcomm.com

Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

Contents

List of Figures

List of Tables

Revision History

Revision	Date	Description
A	May 2013	Version 0.8
B	August 2013	Version 0.81
C	Nov 2013	Version 0.90
D	Jan 2014	Version 0.9.3
E	Feb 2014	Version 0.11.0

1 Introduction

1.1 Purpose

This document describes the Multicore Asynchronous Runtime Environment (MARE) programming model and API.

1.2 Scope

This document is for system developers using the MARE API to develop domain-specific libraries for high-performance applications. MARE handles core management, providing the ability to port an application across multiple cores. Speed is determined by the number of processors on the device.

This document provides the public interfaces necessary to use the features provided by the MARE API. A functional overview and information on leveraging the interface functionality are also provided.

1.3 Conventions

Function declarations, function names, type declarations, and code samples appear in a different font. For example, `#include`.

Code variables appear in angle brackets. For example, `<number>`.

Commands and command variables appear in a different font. For example, `{copy a:*.* b:}`.

1.4 References

The following table lists reference documents, which may include Qualcomm documents and non-Qualcomm standards and resources. Reference documents that are no longer applicable are deleted from this table; therefore, reference numbers might not be sequential. This document also includes a Bibliography at the end of this document with linkable citations throughout.

Table 1-1 Reference documents and standards

Ref.	Document	
Qualcomm		
Q1	Application Note: Software Glossary for Customers	CL93-V3077-1

1.5 Technical Assistance

For assistance or clarification on information in this guide, send email to Qualcomm Technologies, Inc. at mare@qti.qualcomm.com.

1.6 Acronyms

For definitions of commonly used terms and abbreviations, refer to Q1 The following terms are specific to this document.

Table 1-2 Acronyms

Acronym	Definition
API	application programming interface
CTP	cordless telephony profile
DAG	directed acyclic graph
GCC	Globalstar control center
GPGPU	general purpose GPU
LTS	Long Term Support
MARE	Multicore Asynchronous Runtime Environment
MIMD	multiple instruction, multiple data
PPA	personal package archive
MSM	mobile station modem
NDEBUG	C/C++ preprocessor macro for NO DEBUG
NDK	Native Development Kit
SAXPY	scalar vector multiply
SIMD	single instruction, multiple data
SoC	system on chip

2 Installing MARE

This section explains how to configure an application to use MARE given the binary distribution. The installer package available from the Qualcomm Developer Network contains precompiled libraries for Android, GNU/Linux, Microsoft Windows, and Mac OS X. Please install the distribution on your system following the installer prompts, and then see the appropriate section below on how to use it on your particular platform.

2.1 Verifying your installation

By default, the binary installer places the MARE library, headers, and examples in the following directory: `$HOME/Qualcomm/MARE/<version>/<platform>/<build_type>`. We call this directory MARE_DIR in the rest of the document. Substitute *platform* with the name of the directory for native architecture on which you installed MARE, e.g., for Linux, x86_64-linux-gnu. Substitute *build_type* with the desired type, e.g., *debug* or *release*. Additional information on platforms and build types is discussed for each supported architecture below. You may choose to install in a different location, in which case, that location becomes MARE_DIR.

To verify the installation, please do the following:

Android:

```
cd $MARE_DIR/examples
$ANDROID_NDK/ndk-build
$ANDROID_SDK/adb shell mkdir /data/mare
$ANDROID_SDK/adb push bin/helloworld1 /data/mare
$ANDROID_SDK/adb shell /data/mare/helloworld1
```

Linux or Mac OS X:

```
cd $MARE_DIR/examples
cmake .
make
./helloworld1
```

On Windows 32bit:

```
cd %MARE_DIR%/examples
cmake . -G "Visual Studio 12"
cmake --build . --config <Release|Debug> -- /m
helloworld1.exe
```

On Windows 64bit, use the generator for cmake to be "Visual Studio 12 Win64".

The output of this program should be:

```
examples> ./helloworld1
```

```
Hello World!
examples>
```

2.2 Integrating MARE with Android NDK Applications

The precompiled MARE libraries can be easily integrated with an existing native Android application. These libraries have been compiled with the Google NDK r9d that includes GCC 4.8. We recommend using the same NDK and compiler.

Requirements

You must have a working Google NDK r9 or later installed on your system.

You will need to make changes to your project files to use the MARE libraries.

For Heterogeneous Compute support (e.g., GPU offload), please refer to section Heterogeneous Compute.

Modify Application.mk

Edit your project's Application.mk file to include the following entries:

```
APP_STL := gnustl_static
APP_ABI := armeabi-v7a
NDK_TOOLCHAIN_VERSION := 4.8
#set the APP_PLATFORM to match your platform version.
#APP_PLATFORM := android-19
LOCAL_ARM_NEON := true
```

GCC 4.8.x is necessary to link against the precompiled library.

gnustl_static must be used to provide the necessary C++11 library support which is not available in other APP_STL configurations.

Modify Android.mk

Edit your project's Android.mk to define the location of the MARE libraries and headers, MARE_DIR. For Android, MARE_DIR may be an absolute path, or a path relative to your current project. Include the MARE.mk at that location file as follows:

```
MARE_DIR := $(HOME)/Qualcomm/MARE/0.11.0/arm-linux-androideabi/<build_type>
include ${MARE_DIR}/lib/MARE.mk
```

Substitute build_type with one of the following provided build types: release, debug, release-gpu, debug-gpu, and release-ftrace-logging. Substituting ${APP_OPTIM} instead will automatically choose between release and debug builds.

Edit your project's Android.mk file for each build target to include the following:

```
LOCAL_ARM_MODE := arm
LOCAL_CPP_FEATURES := rtti exceptions
LOCAL_STATIC_LIBRARIES := libmare
```

Note that unlike other platforms, $(MARE_CXX_FLAGS) are automatically picked by the android ndk build system from MARE.mk, so it need not be set explicitly.

Build using ndk-build

To build your application using the ndk-build run:

```
$ANDROID_NDK/ndk-build
```

Shared libraries

In most cases, linking statically with libmare will be the easiest approach. For applications with large numbers of libraries, you must link dynamically so that only one copy of MARE exists in memory.

To use MARE as a shared library, you will need to ensure that you include libmare_shared.so into your APK file, and load it using loadLibrary("mare_shared") from Java before loading any other libraries. Next, you need to change LOCAL_STATIC_LIBRARIES to

```
LOCAL_SHARED_LIBRARIES := libmare_shared
```

and add the following to the end of your Android.mk:

```
include $(CLEAR_VARS)
LOCAL_MODULE := libmare_shared
LOCAL_LDFLAGS := -llog
LOCAL_WHOLE_STATIC_LIBRARIES := libmare
include $(BUILD_SHARED_LIBRARY)
```

You will also need to update APP_STL in Application.mk to use gnustl_shared, so that both your application and all the libraries use dynamic linking. You may experience strange crashes if you mix static and dynamic linking.

If you are using the GPU version, please see Heterogeneous Compute for additional requirements.

2.3 GNU/Linux and Mac OS X

MARE can also be compiled on most GNU/Linux and Mac OS X systems. This is useful if you have a portable application that you want to test on a standard PC before running it on an Android device.

Requirements

MARE requires GCC 4.7.1 or later or clang 4.2 (LLVM 3.2) or later, which provide the necessary C++11 functionality.

The recommended MARE development environment is an Ubuntu Linux 12.04 LTS or later system.

If you are starting with a basic Ubuntu system, the necessary compilers and build tools may not be installed by default (the versions for 12.04LTS are not). In this case, you need to install the required versions manually using apt-get.

Please refer to section Heterogeneous Compute for requirements on heterogeneous compute.

Using MARE in your application

The binary distribution includes a `cmake` configuration option that allows you to configure your application to use MARE using the find_package command. Edit your application's CMakeLists.txt to include the following fragment:

```
find_package(MARE PATHS ${MARE_DIR}/lib/CMake/mare} NO_DEFAULT_PATH)
```

```
if(NOT MARE_FOUND)
  message(FATAL_ERROR "MARE not found")
endif()

include_directories(${MARE_INCLUDE_DIRS})
set(CMAKE_CXX_FLAGS "$CMAKE_CXX_FLAGS} ${MARE_CXX_FLAGS}")
add_executable(your_application ${your_application_SRCS})
target_link_libraries(your_application ${MARE_LIBRARIES})
```

Then configure your application using:

```
cmake . -DMARE_DIR=${HOME}/Qualcomm/MARE/0.11.0/<platform>/<build_type>
```

Substitute platform with one of: x86_64-apple-darwin or x86_64-linux-gnu.

Substitute build_type with one of the following provided build types: release, debug, release-gpu, or debug-gpu.

If your application is using a different build system, please look at the definitions in `$MARE_DIR/lib/CMake/lib/MAREConfig.cmake`, and:

Add to your C++ compiler flags the flags listed in ${MARE_CXX_FLAGS}.

Add to your C++ include files the directories listed in ${MARE_INCLUDE_DIRS}.

Add to your libraries flags, the libraries listed in ${MARE_LIBRARIES}.

Note that the MARE API changes based on the definitions listed in ${MARE_CXX_FLAGS}, including with -DNDEBUG. MARE includes a link time check that verifies that the correct flags were used for the application and the included library. Should you not use the correct flags, you will receive a link error, similar to:

```
mare::runtime::__mare_with_api_no_apid_int_debug_init_count", referenced from:
      mare::runtime::init()    in your_application.o
ld: symbol(s) not found
```

Please check your flags before raising an issue.

Additional requirement for using Heterogeneous Compute API

OpenCL 1.2 or later has to be installed. (see section Heterogeneous Compute).

Mac OS X specific notes

Currently, clang is the only compiler included with Xcode that supports C++11 and can compile MARE. When using -std=c++11, you should also use -stdlib=libc++ to include the proper library support.

2.4 Microsoft Windows

The MARE binaries have been compiled for Microsoft Visual Studio 2013. This is useful to develop for Microsoft Windows, and in particular, developing a portable concurrent application that can be easily

targeted to Android as well.

Requirements

MARE requires Microsoft Visual Studio 2013. We make extensive use of C++11, and this is not supported properly in previous versions of Microsoft Visual Studio. MARE is supported only for 32-bit and 64-bit Intel platforms on Microsoft Visual Studio.

The Heterogeneous Compute API is not supported on Windows.

Adding MARE to your application

Precompiled libraries have been provided for debug and release builds, on both Win32 and x64 architectures. Your project will need some changes to support compiling with the MARE headers and libraries. If you are using cmake, please see the instructions for Linux and Mac OS X. Otherwise, make sure that your solution is compatible with VS2013, and right-click on your project and go to the properties editor. Make the following changes in the "Configuration Properties" section:

C/C++ - General - Additional Include Directives

Add the following include paths to the end of your current include list, and replace %(MARE_DIR) to point to the location of your MARE distribution relative to the current project file:

```
;%(MARE_DIR)\include
```

C/C++ - Preprocessor - Preprocessor Definitions

Add the following #define if you are working with a Release build:

```
;NDEBUG
```

Add the following #define for a Debug build:

```
;MARE_ENABLE_CHECK_API_DETAILED
```

If you do not do this correctly, you will see errors involving an unresolved symbol "__mare_with_debug__taskqueue" or "__mare_with_ndebug__taskqueue". This is designed to protect you against incorrectly linking debug and non-debug libraries.

C/C++ - Code Generation - Runtime Library

Change to use "Multi-threaded (/MT)" in Release mode, or "Multi-threaded Debug (/MTd)" in Debug mode, which is what MARE is compiled with. This needs to be consistent across the solution.

Linker - Input - Additional Dependencies

Add the following libraries to the end of your current library list:

```
;%(MARE_DIR)/lib/libmare.lib
```

Substitute %(MARE_DIR) to point to the location of your MARE distribution, relative to the current project file, including the <platform>/<build_type> folders, where.

Substitute *platform* with one of: i386-windows or x86_64-windows.

Substutute *build_type* with one of: release or debug.

If you do not do this correctly, you will see errors involving an unresolved symbol "__mare_with_debug__taskqueue" or "__mare_with_ndebug__taskqueue". This is designed to protect you against incorrectly linking debug and non-debug libraries.

Linker - All Options - Image Has Safe Exception Handlers

Set this feature to no /SAFESH:NO since the inline assembler used inside MARE is incompatible with this feature.

Configure build types

You need to make sure the above changes are made for both Debug and Release, and also for Win32 and x64 builds.

Debugging output

MARE prints debugging information when NDEBUG is not defined to stdout and stderr. These will not be available when compiling some types of Microsoft Windows applications that do not have a console. You need to configure your application correctly if you wish to see the MARE debugging output.

Building

At this point, you should be able to build the entire solution, and your executable is now enabled for use with MARE. You need to initialize the runtime during the startup of your application, and then you can create and launch MARE tasks.

2.5 Heterogeneous Compute

MARE Heterogeneous Compute (e.g., GPU offload) is currently supported through OpenCL. OpenCL 1.2 or later is required on your platform. An example of a Qualcomm-based platform that meets this requirement is the `DragonBoard APQ8074`.

For using MARE's Heterogeneous Compute APIs, please refer to section Heterogeneous Compute API.

OpenCL C++ Support

Using MARE with OpenCL as GPU backend requires presence of the OpenCL C++ header file from Khronos (version must match to your OpenCL driver installation):
`http://www.khronos.org/registry/cl/`
The header file should be installed in a subdirectory ${includedir}/OpenCL/ that is searched by the compiler (e.g., /usr/local/include). Alternatively, additional options would need to be passed as compilation flags (e.g., -I${includedir}).

Unfortunately, the Khronos distributed header file requires small modifications to work on some platforms (see below). In these cases, a patch has to be applied (typically with `patch -p1`). To that end, the MARE binary distribution contains a patch file
`$MARE_DIR/lib/CMake/mare/patches/opencl-headers.patch`.

Android

Using MARE with OpenCL as GPU backend requires the above mentioned OpenCL C++ header file, which has to be patched per above instructions. In your Android.mk file, please set the ANDROID_CL_DIR variable, **before** including MARE.mk, as follows:

```
ANDROID_CL_DIR := <path to directory containing include(with patched cl.hpp) and lib folder conta
```

ANDROID_CL_DIR/include will be used as include path. ANDROID_CL_DIR/lib will be used as lib path.

Linux

On Linux, OpenCL is generally well-supported, OpenCL header files (including OpenCL C++ headers) are packaged by various Linux distributions (e.g., `opencl-headers` on Ubuntu/Debian).

Currently, NVIDIA OpenCL drivers on Linux do not support OpenCL 1.2. They can be used to some degree for testing, provided the above mentioned patch has been applied to the OpenCL C++ header file. The patch prevents linker errors related to `clRetainDevice` and `clReleaseDevice`, but consequently also leaks memory when using OpenCL devices.

As further consequence, AMD OpenCL drivers, which would work fine with OpenCL 1.2 out of the box, will also have to be used against the patched OpenCL C++ header files, and consequently they will also leak memory when accessing OpenCL devices.

Mac OS X

On Mac OS X, the OpenCL framework is pre-installed. However, the OpenCL C++ headers will have to be installed (no patching necessary).

Microsoft Windows

MARE does not currently offer GPU support on Windows. Please contact the MARE developers for inquiries about future support.

3 Getting Started

3.1 Hello World!

Let's explore a simple "Hello World!" example using MARE:

```
1 #ifdef HAVE_CONFIG_H
2 #include <config.h>
3 #endif
4
5 #include <mare/mare.h>
6 #include <stdio.h>
7
8 using namespace std;
9
10 int main() {
11
12   // Initialize the MARE runtime.
13   mare::runtime::init();
14
15   // Create task that prints "Hello "
16   auto hello = mare::create_task([]{
17       printf("Hello ");
18     });
19
20   // Create task that prints " World!"
21   auto world = mare::create_task([]{
22       printf("World!\n");
23     });
24
25   // Make sure that "World!" prints after "Hello"
26   mare::after(hello, world);
27
28   // Launch both tasks
29   mare::launch(hello);
30   mare::launch(world);
31
32   // Wait for world to complete
33   mare::wait_for(world);
34
35   // Shutdown the MARE runtime.
36   mare::runtime::shutdown();
37
38   return 0;
39 }
```

In line 5, we include the `mare.h` header, which is needed for any MARE program. All the MARE classes and functions are declared in the `mare` namespace.

Line 14 initializes the MARE runtime. There needs to be one invocation of `mare::runtime::init()` in the program, before any other MARE APIs are called. Repeated invocations of this function are allowed, however, they are not recommended. The function initializesall internal data structures, scheduler and starting of underlying threads. Pairwise with `init`, there is an equivalent `mare::runtime::shutdown()` (line 37). Again, at least one invocation of shutdown is necessary to tear down the runtime.

In this example, we use `mare::create_task` to create two tasks, `hello` (line 17) and `world` (line 22). `create_task` takes as a parameter a `lambda` expression, really just an anonymous function, which defines the work that the tasks execute. In our case, `hello` prints "Hello" and `world` prints "World!".

When a new task is created, the programmer has a chance to set up dependencies with other tasks. In the example, we use `mare::after` to ensure that `world` is printed after `hello`. Without this dependency, the MARE runtime could execute `world` first, or concurrently. Once the dependency between the tasks is set up, we launch them in lines 30 and 31. Launching tells the MARE runtime that the tasks are ready for execution. Finally, in line 34, we wait for `world` to finish. We do not need to explicitly wait for `hello` because the MARE runtime ensures that `hello` executes before `world`.

This simple example illustrates two basic MARE abstractions: tasks and dependencies. In MARE, programmers think about algorithms in terms of concurrent tasks and let the MARE runtime schedule them onto available resources in the system. Programmers can create dynamic task graphs by setting dependencies between tasks that the runtime enforces. Another key MARE abstraction —not shown in the example— are groups. Groups allow the programmer to easily manage sets of tasks. In the example, we could have used a group to wait for both tasks.

3.1.1 Compiling Hello World!

To compile the above example, we can use a standalone `Makefile`:

```
1 # -*- Makefile -*-
2 # Example standalone makefile for helloworld1
3
4 helloworld1_SOURCES = helloworld1.cc
5 helloworld1_OBJECTS = $(helloworld1_SOURCES:%.cc=%.o)
6
7 # Same value as provided to MARE for the invocation of ./configure --prefix=...
8 MARE_PREFIX = /path/to/mare/dir
9
10 CXX      = g++ -std=gnu++0x -pthread
11 CPPFLAGS = -I$(MARE_PREFIX)/include
12 CXXFLAGS = -g -O2 -Wall -Wextra
13 LDFLAGS  = -L$(MARE_PREFIX)/lib
14 LIBS     = -lmare
15
16
17 all: helloworld1
18
19 helloworld1: $(helloworld1_OBJECTS)
20     $(CXX) $(CXXFLAGS) $(LDFLAGS) $^ $(LIBS) -o $@
21
22 .PHONY: clean
23 clean:
24     $(RM) $(helloworld1_OBJECTS) helloworld1
25
26 .DELETE_ON_ERROR:
```

Note that the C++ code needs to be compiled with support for at least GNU++0x extensions (option `-std=gnu++0x`), preferably C++11 (option `-std=c++11`). We also need to ensure that the MARE install path is properly set; otherwise this results in compilation and/or link errors.

We assume the current directory contains two files:

```
Makefile
helloworld1.cc
```

The example can then be built and run by typing the following at the command prompt:

```
$ make
$ ./helloworld1
Hello World!
```

4 User Guide

4.1 Overview

All current hardware platforms, from desktops to tablets and smartphones, are built around multicore and heterogeneous systems on a chip (SoC). MARE addresses the question of how to enable full utilization of the hardware at the user application level, in the following way:

- By providing a concurrent programming model that allows developers to express the concurrency in their applications. MARE's powerful abstractions ease the burden of parallel programming since it was designed with dynamic concurrency from the ground up.
- By embedding the programming model in C++ and providing a C++ library API. C++ is a familiar language for a large number of performance oriented developers, thus making it easy for programmers to pick up the abstractions quickly. C++ embedding also allows incremental development of existing applications, since MARE interoperates with existing libraries, such as pthreads and OpenGL.

MARE runs on top of a runtime system that will execute the concurrent applications on all the available computational resources on the SoC. Our current implementation is focused on multicore with future releases supporting heterogeneous SoCs.

4.1.1 Writing a MARE Application

The fundamental abstraction in MARE is a task – an independent unit of work that can execute concurrently with other tasks. To build a MARE application, the programmers express the computation as a partially ordered task graph. The nodes in the graph are tasks, and the edges are dependencies. The directed acyclic graph (DAG) is scheduled by the runtime system based on resource availability as dependencies are satisfied. The reason this makes it easier to write parallel applications than using thread libraries is because the expression of concurrency is decoupled from the hardware thread abstraction. This has two advantages: first, programs are portable between different multicore platforms, since the application does not rely on a predefined number of cores or threads; second, tasks are light-weight, and therefore allow the expression of finer-grained concurrency, thus giving programmers the ability to parallelize a much larger set of algorithms and applications. In addition, MARE provides a set of attributes (annotations on tasks) that allow programmers to provide additional information about the semantics of the application, thus giving the

runtime system more insights into the application behavior. This allows the runtime system to make better scheduling decisions, increasing the efficiency of the parallel application.

Figure below illustrates how to write an application using MARE:

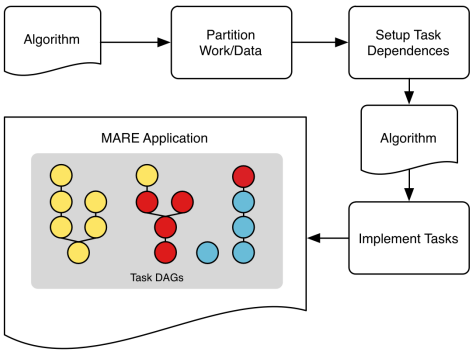


Figure 4-1 MARE workflow

The workflow from start to completion of a MARE application is as follows:

- Identify the algorithm to be parallelized and design a parallel version of the algorithm.
- Encode the algorithm using MARE abstractions. This includes partitioning the algorithm into tasks and partitioning the data for concurrent access.
- Building the DAG by setting up dependencies between tasks. MARE decouples the task creation from the task launch to allow dynamically building the DAG.
- The programmer has now developed a skeleton of the parallel algorithm and should implement each task. MARE is based on a shared memory programming paradigm. As tasks access memory, the programmer must protect shared data accesses with appropriate synchronization primitives (locks or atomic operations).
- The MARE application consists of a forest of DAGs. The runtime system schedules the tasks once their dependencies are satisfied.

Of course, there are a number of housekeeping functions provided in the MARE API that allow an application programmer to manage the runtime and the concurrency of the application. These are explained in later sections of the manual.

4.1.2 Executing a MARE Application

The MARE runtime fundamentally implements a thread pool over which tasks are scheduled at the user level. When the application starts running, the thread pool is initialized such that it makes optimal use of the existing hardware contexts on the device. The scheduler is a throughput-oriented scheduler. Tasks are scheduled in a non-preemptive manner as they are ready for execution (dependencies are satisfied). The task attributes mechanism in MARE allows developers to provide additional application information, which enables the runtime system to make better decisions and schedule tasks on the appropriate resources.

The figure below illustrates how the MARE runtime executes MARE applications.

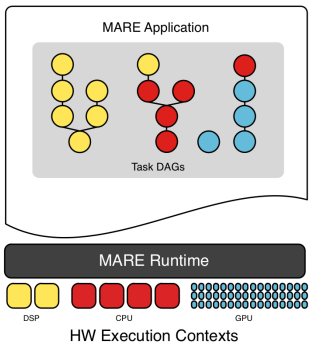


Figure 4-2 MARE execution

4.2 Parallel Programming Patterns

One of MARE's main goals is to simplify parallel programming. To this end, MARE provides several constructs that encapsulate commonly used parallel programming patterns. Examples include: parallel iteration (parallel loop), parallel scan, and parallel transform. See "patterns.hh" for a detailed description of the parallel patterns. In this section we provide a high-level overview of what to expect from the parallel patterns.

The recommended workflow for using MARE to parallelize an algorithm is: check whether the parallelization can be expressed using one of the existing patterns. If so, use the pattern, measure the performance and efficiency, and refine the implementation using other MARE constructs only if the pattern implementation does not satisfy the performance criteria.

4.2.1 Parallel Iteration

We currently support two parallel iterators (parallel loop type of constructs): `mare::pfor_each_async` and `mare::pfor_each`. These iterators concurrently execute the body of the loop for each instance of the element in the collection returned by the input iterator taken as an argument. The iterator can be expressed as a pair of integers, (lower bound, upper bound), or an iterator (begin, end).

```
// Parallel for-loop using indices
// vout[i] := 2*vin[i]
vector<size_t> vout(vin.size());
pfor_each(size_t(0), vin.size(), [=,&vin,&vout] (size_t i) {
       vout[i] = 2*vin[i];
});
```

Our current implementation divides the range of the iterator into chunks, and executes each set of iterations in a chunk as a task. This exploits locality and reduces the overhead of launching tasks. The chunking

heuristic is simple, but effective for most array-based operations. More complex heuristics may be provided in the future.

The difference between `mare::pfor_each_async` and `mare::pfor_each` is that the `pfor_each` construct returns only when all the tasks have completed, while the `pfor_each_async` does not wait for termination.

4.2.2 Parallel Map and Parallel Prefix

Another common pattern is `mare::ptransform`. This pattern applies a function object in parallel to every element of the input range collection(s), and stores the value in the output range. We provide both unary and binary versions of the operator; the latter case uses two input range collections. The ptransform pattern is blocking, waiting for the operations to terminate before returning.

MARE also implements `mare::pscan_inclusive`, a Sklansky-style, in-place parallel prefix operation. The applied operation must be associative, as the order in which the operations are applied is unpredictable. pscan_inclusive is also blocking, returning only when all the operations have completed.

The existing patterns are built using the basic MARE constructs of tasks and groups. However, they are optimized using knowledge about the pattern structure and the operations in the runtime to minimize the amount of synchronization, and to avoid other bookkeeping operations that are needed for more generic use. Because these patterns are layered on top of the MARE abstractions, programmers are welcome to add to the library of patterns.

Synchronous Data-Flow (SDF)

4.2.3 Synchronous Data-Flow (SDF)

The MARE SDF API supports the Synchronous Data-Flow graph (SDF) programming model. The SDF API allows the programmer to describe a graph of compute nodes that are connected with streaming data channels. The programmer associates a C++ node-function with each node, and specifies a basic C++ type or a user-defined data-type for each channel. Once launched, the graph executes synchronously, i.e., the nodes execute in lock-step, and at every step each channel has exactly one element pushed and one element popped (in FIFO order). Hence, SDF executes as a sequence of graph iterations. Note the contrast with the standard synchronous dataflow model where a node needs to pop/push a constant number of elements from a channel in each graph iteration, not exactly one element as required by MARE SDF. With MARE SDF, the user can create a channel that carries multiple data items wrapped as a single data item, to gain equivalency with the general synchronous dataflow case.

4.2.3.1 Is MARE SDF for me?

Like the MARE API for tasks, the SDF API is intended for easy integration into C++ applications. It is meant to ease the programmer's burden with regards to the correct and efficient orchestration of a collection of work specified by the programmer.

Application work that is repeated many times with the same control-flow or data-flow structure is a good candidate for execution via SDF. Applications from Computer Vision and Gaming tend to structure computation into frames, where the high-level compute structure every frame is often fixed. Additionally, an application component within a single frame may execute by applying the same structure of computation to a sequence of data, such as processing image data in segments. SDF can be applied at multiple levels in a such a situation: a graph for the frame and a graph implementing a component of the frame graph. The component graph would run multiple iterations for each iteration of the frame graph. In this situation, the

SDF API allows the creation of all the graphs upfront, avoiding the creation overheads of the component graph every frame. The component graph can simply be paused at the end of each frame, and resumed at the start of the next frame.

Algorithms from signal processing can also be expected to map in a straightforward manner to SDF.

4.2.3.2 Features

The MARE SDF API is designed with the following intent.

1. The SDF graph is created dynamically by a C++ program. Though once launched the graph can no longer be modified.

2. The SDF graph allows arbitrary C++ code in a node-function, though the parameter list of a node-function is dictated by the channels connected to the corresponding node.

3. The streaming data channels can transport any data-type that is shallow-copyable, i.e., a `memcpy()` should suffice for transporting the data from a producer node to a consumer.

4. The SDF semantics are automatically enforced via the API in the following ways.

- *Iteration ordering*: a node executes graph iteration i+1 only after the node has completed iteration i.
- *Implicit channel access*: the input channels of a node are each popped once before the node executes, and the output channels are each pushed once after the node completes. The node-function parameter list provides access by-reference to the inputs and outputs. The user-code cannot perform any explicit pop or push operations.
- *Precise pause-resume and cancel semantics*, built around the notion of SDF graph iterations. The user can specify if the interruptions (pause or cancel) occur at graph iteration boundaries, or should occur ASAP even if different nodes execute a different number of iterations. Here the user can choose between precise semantics or the low latency application of an interruption request. Additionally, the user can specify the minimum number of iterations that must be completed before the interruption is applied.

5. The SDF runtime performs automatic optimizations, including pipeline parallelization and lowered scheduling overheads, based on the fixed structure of the graph.

6. SDF does not restrict the user from mixing in other C++ paradigms, including other parallel programming constructs, such as mutexes on global shared data. Mixing other C++ paradigms with SDF describes how some common paradigms mix with SDF for greater benefit.

4.2.3.3 Example 1: A simple pipeline

The following code implements a graph with the pipeline structure shown below.

```
    dc1         dc2
A -------> B -------> C
```

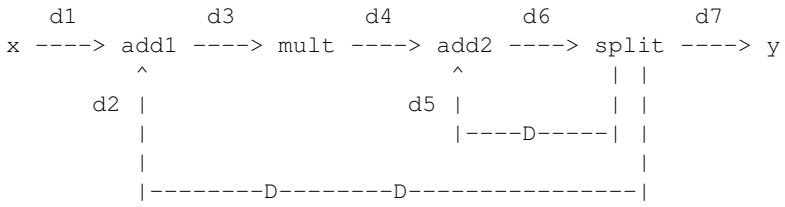
Nodes A, B, C are connected using channels dc1 and dc2. dc1 and dc2 carry values of type `int`. The graph will execute 10 iterations, resulting in the data structure `vresults` containing 10 integers. The nodes A, B and C execute the node-function assigned to them when each node was created. In this example, all the node-functions are C++11 lambdas, but they could be regular C/C++ functions, among other options. Note that the parameter list of each lambda matches the number and data-types of the channels connected to the node corresponding to that lambda. The MARE runtime invokes each node-function 10 times during the

graph execution, corresponding to the 10 graph iterations requested. Values are streamed from A to B via dc1 and from B to C via dc2.

```
std::vector<int> vresults;

mare::data_channel<int>  dc1, dc2;

mare::sdf_graph_ptr g = mare::create_sdf_graph();

int counter = 0;
// Node A
mare::create_sdf_node(g,
                      [&](int& output)
                      {
                        output = counter++;
                      },
                      mare::with_outputs(dc1));

// Node B
mare::create_sdf_node(g,
                      [](int& input, int& output)
                      {
                        output = input * 2;
                      },
                      mare::with_inputs(dc1),
                      mare::with_outputs(dc2));

// Node C
mare::create_sdf_node(g,
                      [&](int& input)
                      {
                        vresults.push_back(input);
                      },
                      mare::with_inputs(dc2));

const std::size_t num_iterations = 10;
mare::launch_and_wait(g, num_iterations);

mare::destroy_sdf_graph(g);
```

4.2.3.4 Example 2: Filter graph with feedback edges

The following graph illustrates a filter with feedback edges implemented using MARE SDF.



The "D"'s represent iteration delays along edges. For example, edge `d2` takes the value `y` produced at end of iteration `i` and presents it to be added to the value of `x` at the start of iteration `i+2` (since there are two "D"'s along the edge `d2`).

`filter()` applies an IIR filter to a sequence of values in `x_arr` and produces a sequence of filtered results in `y_arr`. The edges in the graph become streaming channels in MARE SDF.

```
void filter(float x_arr[], float y_arr[], int N, float coeff) {
  using namespace mare;

  data_channel<float> d1, d2, d3, d4, d5, d6, d7;

  auto g = create_sdf_graph();

  int x_index = 0;
```

```
  int y_index = 0;

  // Reads an input data item x in each iteration
  create_sdf_node(g, [&](float& x) { x = x_arr[x_index++]; },
                  with_outputs(d1));

  // add1: adds x to the filter output from two iterations ago
  create_sdf_node(g, [](float& in1, float& in2, float& out) { out = in1 + in2; },
                  with_inputs(d1, d2), with_outputs(d3));

  // mult: multiply by a fixed coefficient
  create_sdf_node(g, [=](float& in, float& out) { out = coeff * in; },
                  with_inputs(d3), with_outputs(d4));

  // add2: adds the intermediate data to the filter output from one iteration ago
  create_sdf_node(g, [](float& in1, float& in2, float& out) { out = in1 + in2; },
                  with_inputs(d4, d5), with_outputs(d6));

  // splits the adder output into y and the two feedback edges
  create_sdf_node(g,
                  [](float& in, float& out1, float& out2, float& out3)
                  {
                    out1 = out2 = out3 = in;
                  },
                  with_inputs(d6), with_outputs(d2, d5, d7));

  // Writes a filtered output in each iteration
  create_sdf_node(g, [&](float& in) { y_arr[y_index++] = in; },
                  with_inputs(d7));

  // Add two delays to d2
  preload_channel(d2, std::vector<float>{0.0, 0.0});

  // Add one delay to d5
  preload_channel(d5, std::vector<float>{0.0});

  // Stream N data items through graph
  launch_and_wait(g, N);

  destroy_sdf_graph(g);
}
```

All channels in MARE SDF are single-source and single-destination. Note the use of an explicit splitter node when one output value is consumed by multiple source nodes along multiple channels. Here, the values carried by `dc6` are split by a node by repetition along channels `dc2`, `dc5` and `dc7`. Instead of having the add followed by a split, the user could have alternatively defined a combined "add-and-split" node that replicated the sum on three output channels.

The `preload_channel()` function adds delays to channels. A certain number of initial values are pre-loaded into the channel. The number of initial values introduce that many delays along the channel. The initial values will be consumed first (by channel pops in the early graph iterations) while new values get pushed in FIFO order.

Any cycle in a synchronous dataflow graph must have at least one delay on an edge in the cycle. Otherwise, the graph is not a valid synchronous dataflow graph. At launch time, MARE SDF analyzes the graph for optimal execution. Improper SDF graphs are detected during this analysis at launch, producing a runtime error before the graph commences execution. Examples of improper graphs include:

- a channel connected to a node at only one end (dangling channel)
- a graph cycle with no delays along any of its channels

4.2.3.5 Mixing other C++ paradigms with SDF

The following constructs provide functionality that SDF does not attempt to duplicate. These constructs can be inter-mixed with SDF with the usual care associated with such constructs and a minimal understanding of the SDF semantics provided above in Features.

- *Global variables*: node-functions can access global variables. Global variables might provide shared read-only data for the node-functions, though the data could be freely modified by the application when the graph is paused. Or, a node-function may exclusively update global data not shared with other node-functions (e.g., a source node reading from a global file-pointer and feeding data into the graph).

- *Mutexes, conditional-wait signalling*: A node-function may access shared state protected by mutexes. The user must be cognizant of the SDF semantics about the serialization imposed on a node by the iteration ordering (node n may execute iteration i+1 only after node n has executed iteration i). The dependence structure of the graph may further constrain node n2 to execute iteration i only after a predecessor node n has executed iteration i. Apart from these restrictions, SDF may execute nodes concurrently for the same iteration i, and may have executions of multiple iterations (iterations ...,i-1, i, i+1, i+2, ...) in flight at the same time for different parts of the graph. Care must be taken if the user intends to perform conditional-waits across SDF nodes in the same graph, as deadlocks can occur due to the order of execution chosen by SDF. The user should ensure that the wait-signalling nodes are chosen so that they are guaranteed to be executed by SDF in a safe order based on the SDF semantics. Note: consider using `mare::mutex` and `mare::condition_variable` instead of the C++ Standard library variants, as these allow the MARE runtime to become aware of blocked work and better schedule work across cores.

- *Callable objects to manage state*: Callable objects can impart state to node-functions, avoiding a reliance on global state. Additionally, the callable object may implement methods that provide consistent interfaces for both the node-function and the external application to access the object data (say, locking a mutex to guard data that may be concurrently accessed).

4.3 Tasks

MARE programmers must partition their application into independent units of work that can be executed asynchronously. Those units of work are called ∗tasks∗. Tasks can be of arbitrary size, although the task granularity affects performance (see Choosing the Right Granularity for a Task). In MARE, each task can have predecessors and successors. The predecessors of a task `T` are the tasks that must complete before `T` can execute. Conversely, the sucessors `T` are the set of tasks that will execute only after `T` has completed its execution.

Tasks do not execute when they are created. Instead, they must be launched to be executed. Launching a task means that the programmer has finished adding predecessors to it and that she wants the MARE runtime to execute this task as soon as possible. Tasks can be launched into groups, which are sets of tasks that can be waited on, or canceled at once. For information about groups, see Groups.

Managing the lifetime of an object in a parallel application can be challenging. This is why MARE automatically destroys tasks once they execute and all references are no longer in scope.

4.3.1 Task Creation

MARE offers two templated methods to create tasks:

1. `template<typename Body> mare::task_ptr mare::create_task(Body&&)`
2. `template<typename Body> void mare::launch(mare::group_ptr const&, Body&&)`

The first one creates a task and returns a pointer to it. The second one creates a task and launches it into a group. In both cases, the tasks execute the `Body` passed as a parameter. The preferred type of `<typename Body>` in both template methods is a lambda expression, although it is possible to use other types such as function objects and function pointers.

Using `mare::launch(mare::group_ptr const&, Body&&)` is the fastest way to create tasks in MARE and should be used as often as possible. For more information about groups, see Groups.

4.3.1.1 Create Tasks Using Lambda Expressions

Lambda expressions are a new feature in C++11, and the preferred argument type to `template<typename Body> mare::task_ptr mare::create_task(Body&&)` and `template<typename Body> void mare::launch(mare::group_ptr const&, Body&&)`.

Lambda expressions are unnamed function objects that are able to capture variables from the enclosing scopes. A description of this C++11 feature is outside the scope of this document. Find detailed information about lambda expressions in the following links:

- C++11 Tutorial: Lambda Expressions -- The Nuts and Bolts of Functional Programming
- Lambda functions
- Lambda Functions in C++11 - the Definitive Guide
- Michael Caisse: Lambda Functions

The following code uses a lambda expression to create a task `t1` that prints 'Hello World!':

```
1 // Create task that prints Hello World!
2 mare::task_ptr t1 = mare::create_task([]{
3     printf("Hello World!\n");
4   });
```

Alternatively, you could create the task using `template<typename Body> void mare::launch(mare::group_ptr const&, Body&&)`:

```
1 // Create and launch a task into group g that prints Hello World!
2 mare::launch(g, [] {
3     printf("Hello World!\n");
4   });
```

The lambda expression in the previous example is very simple as it does not capture any variables. Let's suppose that you want to capture a string with the user name to do a proper greeting:

```
1 // Create and launch task into group g that prints Hello World!
2 {
3   std::string name = get_user_name();
4   mare::launch(g, [name] {
5       printf("Hello World, %s!\n", name.c_str());
6     });
7 }
```

By capturing `name` in the lambda expression, we make sure that we can use it when the task executes, which happens outside the scope where the task is created. Make sure that, if you capture variables by reference, the original object still exists when the task executes. For example, consider the following code:

```
1 // Create and launch a task into group g that prints Hello World!
2 {
3   std::string name = get_user_name();
4   mare::launch (g, [&name] {
5       printf("Hello World, %s!\n", name.c_str());
6     });
7 }
```

The string `name` goes out-of-scope in line 7, and its destructor is called then. If the scheduler executes the task after that happens, the program will most likely crash.

Please refer to The task_ptr and unsafe_task_ptr pointers for information about capturing `mare::task_ptr` by reference and why you should never do it.

Warning

> Using ∗default capture∗ by copy (`[=]`) or by reference (`[&]`) will capture all variables from the enclosing scope, which may increase the size of your tasks considerably if the compiler cannot figure out that many of them are not used and do not need to be captured. We recommend that you capture only the variables that your lambda expression uses.

4.3.1.2 Create Tasks Using Classes

You can use any custom class as `<typename Body>` by overloading the class's `operator()`. Notice that the `operator()` cannot have any arguments. The following code shows how to create a task from a class instance. When the MARE scheduler executes the task, the `operator()` method is called.

```
1
2 class user_class {
3 public:
4   user_class(int value)
5     :x(value) {
6   }
7
8   void operator()() {
9     printf("x = %d\n", x);
10   }
11
12   void set_x(int value) {
13     x = value;
14   }
15
16 private:
17   int x;
18 };
24 mare::launch(g, user_class(42));
```

It is also possible to create an object from `user_class` and then create a task using that object:

```
1 // Create task using a class
2 user_class obj(42);
3 auto t = mare::create_task(obj);
4 mare::launch(g, t);
```

The previous example raises an interesting question: What would the task print if we called `obj.set_x(100)` between lines 3 and 4?

```
1 // Create task using a class
2 user_class obj(42);
3 auto t = mare::create_task(obj);
4 obj.set_x(100);
```

```
5 mare::launch(g, t);
```

As always, the answer to the ultimate question of life, the universe, and everything is `42`. The reason is that MARE makes a copy of `obj` when it creates the task in line 3. Otherwise, users would need to keep track of the lifetime of the objects used to create tasks. However, if you were to construct the object in-place, no copies would be made:

```
1 // Create task using a class
2 auto t = mare::create_task(user_class(42));
3 mare::launch(g, t);
```

4.3.1.3 Create Tasks Using Function Pointers

The last way to create a task is by using a function pointer. As with lambda expressions and classes, the function passed as parameter cannot take any arguments:

```
1 void foo() {
2   printf("Hello World!\n");
3 };
4
5 // Create task that executes foo()
6 auto t = mare::create_task(foo);
```

Warning

Due to limitations in the Visual Studio C++ compiler, this does not work on Visual Studio. You can get around it by using a lambda function:

```
1 void foo() {
2   printf("Hello World!\n");
3 };
4
5 // Create task that executes foo()
6 auto t = mare::create_task([] {
7     foo();
8   });
```

4.3.1.4 The task_ptr and unsafe_task_ptr pointers

The method `mare::create_task(Body&&)` returns an object of type `mare::task_ptr`, which is a custom smart pointer to the task object.

Tasks are reference-counted, so they are automatically destroyed when no more `mare::task_ptr` pointers reference them. When a task is launched, MARE runtime increases the reference counter of the task. This prevents the task from being destroyed, even if all pointers referencing the task are reset. The MARE runtime decrements the reference counter of the task after it completes execution.

The task reference counter requires atomic operations. Copying a `mare::task_ptr` causes an atomic increment and the new copy of the `mare::task_ptr` causes an atomic decrement when it goes out of scope. For best results, minimize the times your application copies `mare::task_ptr` pointers.

Some algorithms require constantly passing `mare::task_ptr` pointers. To prevent a decrease in performance, MARE provides another task pointer type that does not perform reference counting: `mare::unsafe_task_ptr`.

The following example demonstrates how to point `mare::unsafe_task_ptr` to a task:

```
1 // Create task that prints Hello World!
2 mare::task_ptr t1 = mare::create_task([]{
3     printf("Hello World!\n");
4   });
5
```

```
6 // Get unsafe_task_ptr pointing to task
7 mare::unsafe_task_ptr unsafe_t1 = t1.get();
```

Note: Task lifetime is determined by the number of `mare::task_ptr` referencing it. Programmers must ensure that there is always a valid `mare::task_ptr` while using a `mare::unsafe_task_ptr`, otherwise it will lead to memory corruption and/or segmentation fault.

You can use a `mare::unsafe_task_ptr` in any API method in which you can use a `mare::task_ptr`, with the exception of `mare::launch_and_reset(mare::task_ptr&)`.

It is incorrect to reference a MARE `mare::task_ptr` like the following example:

```
1 mare::task_ptr t1 = mare::create_task([] {
2     printf("Hello World from t1!\n");
3   });
4 mare::task_ptr t2 = mare::create_task( [&t1] {
5     printf("Hello World from t2!\n");
6   });
```

Instead, copy the pointer

```
1 mare::task_ptr t1 = mare::create_task([] {
2     printf("Hello World from t1!\n");
3   });
4 mare::task_ptr t2 = mare::create_task( [t1] {
5     printf("Hello World from t2!\n");
6   });
```

Or use a `mare::unsafe_task_ptr` (of course, make sure that `t1` does not go out of scope):

```
1 mare::task_ptr t1 = mare::create_task([] {
2     printf("Hello World from t1!\n");
3   });
4
5 mare::unsafe_task_ptr unsafe_t1= t1.get();
6
7 mare::task_ptr t2 = mare::create_task([unsafe_t1] {
8     printf("Hello World from t2!\n");
9   });
10
11 mare::task_ptr t3 = mare::create_task([&unsafe_t1] {
12     printf("Hello World from t3!\n");
13   });
```

4.3.2 Create Dependencies Between Tasks

Use the following methods to specify the order of task execution:

- `void mare::before(mare::task_ptr const&, mare::task_ptr const&)`
- `void mare::after(mare::task_ptr const&, mare::task_ptr const&)`

The following example shows how to use `mare::after` to ensure that task `t1` executes before task `t2`. The MARE runtime guarantees that `t2` does not begin execution until `t1` completes execution, regardless of how many hardware execution contexts are available in the system.

```
1 mare::task_ptr t1 = mare::create_task([] {
2     printf("Hello ");
3   });
4
5 mare::task_ptr t2 = mare::create_task([] {
6     printf("World!\n");
7   });
8
9 // Make sure that t1 executes before t2
```

```
10 mare::after(t1, t2);
```

The previous example is equivalent to:

```
1 mare::task_ptr t1 = mare::create_task([] {
2     printf("Hello ");
3   });
4
5 mare::task_ptr t2 = mare::create_task([] {
6     printf("World!\n");
7   });
8
9 // Make sure that t1 executes before t2
10 mare::before(t2, t1);
```

Note

Alternatively, you can use `operator>>` instead of `mare::after` and `operator<<` instead of `mare::before`:

```
1 mare::task_ptr t1 = mare::create_task([] {
2     printf("Hello ");
3   });
4
5 mare::task_ptr t2 = mare::create_task([] {
6     printf("World!\n");
7   });
8
9 // Make sure that t1 executes before t2
10 t1 >> t2;
11
```

Because tasks can have multiple successors and predecessors, you can use `mare::before` and `mare::after` to create DAGs:

```
1 mare::task_ptr t1 = mare::create_task([]{ printf("Task t1\n");});
2 mare::task_ptr t2 = mare::create_task([]{ printf("Task t2\n");});
3 mare::task_ptr t3 = mare::create_task([]{ printf("Task t3\n");});
4 mare::task_ptr t4 = mare::create_task([]{ printf("Task t4\n");});
5
6 // Make sure that t1 executes before t2 and t3.
7 // t2 and t3 may execute concurrently
8 // t4 executes only after t2 and t3 have completed.
9 mare::after(t1, t2);
10 mare::after(t1, t3);
11 mare::after(t2, t4);
12 mare::after(t3, t4);
```

Warning

A cycle in the DAG may cause deadlock. For performance reasons, MARE does not check whether there are cycles in the DAG. The programmer is responsible for avoiding them.

```
1 mare::task_ptr t1 = mare::create_task([] {
2     printf("Hello ");
3   });
4 mare::task_ptr t2 = mare::create_task([] {
5     printf("World!\n ");
6   });
7
8 // Create a cycle in the DAG:
9 mare::after(t1, t2);
10 mare::after(t2, t1);
11
12 mare::launch(t1);
13 mare::launch(t2);
14
15 // It will never return
16 mare::wait_for(t1);
17
```

In the previous example, `mare::wait_for(t1)` will never return because `t1` needs `t2` to complete before it can execute. However, `t2` will never execute because it needs `t1` to execute first.

4.3.3 Task Storage

4.3.3.1 Task-Local Storage

Tasks, much like threads, can be associated with task-local storage, via `mare::task_storage_ptr`. The usage pattern consists of declaring a global variable, say `storage`, which holds a pointer to the actual task-local data. Then, within a task `t`, that variable is assigned a pointer to a (usually) local variable, or a chunk of freshly allocated memory. After that, `storage` can be used within the dynamic extent of task `t`:

```
1 namespace {
2 task_storage_ptr<int> storage;
3 }; // namespace
4
5 void func() {
6   printf("%d\n", *storage);
7   ++*storage;
8 }
9
10 void run() {
11   auto g = create_group("test");
12   for (int i = 0; i < N; ++i) {
13     launch(g, [i] {
14         int v = i;
15         storage = &v;
16         func();
17         assert(v == i+1);
18         func();
19         assert(v == i+2);
20       });
21   }
22   wait_for(g);
23 }
24
```

Note that accessing the value of `storage` affects only the current task. Attempting to modify the value of a `task_storage_ptr` outside of a task yields undefined behaviour.

Optionally, a destructor (or rather: finalizer), can be employed to dispose resources. The destructor will run within each task, which has a value assigned to the global variable.

4.3.3.2 Scheduler-Local Storage

Another use case are scratchpads: data that is persistent across task boundaries, usually to avoid per-task memory allocation or initialization. We can avoid synchronizing access to scratchpads if each scheduler creates its own scratchpad (which can then be used like task-local storage). As further optimization, ‘scheduler_storage_ptr‘ are created lazily when they are written to inside of a task. Note that variable initialization and destruction happen through the constructor and destructor of ‘T‘:

```
1 #include <mare/schedulerstorage.hh>
2
3 namespace {
4 const scheduler_storage_ptr<size_t> s_sls_state;
5 }; // namespace
6
7 void run() {
8   auto g = create_group("test");
9
10   for (size_t i = 0; i < 200; ++i) {
11     launch(g, [i] {
12         size_t c = ++*s_sls_state;
```

```
13         // values for c are consecutive on a per-scheduler basis
14       });
15   }
16
17   wait_for(g);
18 }
```

Scheduler-local storage is unaffected by context switches (e.g., via `wait_for`, or `yield`).

```
1 #include <mare/schedulerstorage.hh>
2
3 namespace {
4 const scheduler_storage_ptr<size_t> s_sls_state;
5 }; // namespace
6
7 void run() {
8   auto g = create_group("test");
9   auto t = create_task([] {});
10
11   for (size_t i = 0; i < 200; ++i) {
12     launch(g, [=] {
13         size_t c1 = ++*s_sls_state;
14         launch(t);
15         wait_for(t);
16         size_t c2 = ++*s_sls_state;
17         assert(c1 + 1 == c2);
18       });
19   }
20
21   wait_for(g);
22 }
```

A complete example

```
1 #include <assert.h>
2
3 #include <algorithm>
4 #include <iterator>
5
6 #include <mare/mare.h>
7 #include <mare/schedulerstorage.hh>
8
9 using namespace mare;
10
11 template <size_t N>
12 struct image_scratchpad {
13   image_scratchpad() {
14     std::fill(begin(edge_image), end(edge_image), 0);
15   }
16   char edge_image[N];
17 };
18
19 namespace {
20 const scheduler_storage_ptr<image_scratchpad<4096> >
       image_buffers;
21 }; // namespace
22
23 void run() {
24   int const N = 200;
25
26   auto g = create_group("test");
27   for (int i = 1; i < N; ++i) {
28     launch(g, [i] {
29         // fill image buffer, which is reused across tasks
30         for (auto& slot : image_buffers->edge_image)
31           slot = i & 0xff;
32         internal::yield(); // context-switch, we expect SLS to survive this
33         // check contents
34         for (auto const& slot : image_buffers->edge_image)
35           assert(slot == char(i & 0xff));
36       });
```

```
37   }
38   wait_for(g);
39 }
```

4.3.3.3 Thread-Local Storage

If a group of tasks needs scratchpads, but does not require that data persists across context switching, `thread_storage_ptr` is a viable alternative to `scheduler_storage_ptr`. Since MARE Thread-Local Storage is tied to MARE's device thread, we allocate fewer instances of 'T' (compared to `scheduler_storage_ptr`, see earlier example), at most one per device thread.

```
1 #include <mare/threadstorage.hh>
2
3 namespace {
4 const thread_storage_ptr<size_t> s_tls_state;
5 }; // namespace
6
7 void run() {
8   auto g = create_group("test");
9   auto t = create_task([] {});
10
11   for (size_t i = 0; i < 200; ++i) {
12     launch(g, [=] {
13         size_t* p1 = s_tls_state.get();
14         launch(t);
15         wait_for(t);
16         size_t* p2 = s_tls_state.get();
17         // cannot assume that p1 == p2
18       });
19   }
20
21   wait_for(g);
22 }
```

4.3.4 Launching Tasks

Tasks do not execute unless they are launched. In Groups we describe how to create and launch tasks using `template<typename Body> void mare::launch(mare::group_ptr const&, Body&&)`. Tasks launched in this way execute as soon as hardware contexts are available.

Note: `mare::launch(mare::group_ptr const&, Body&&)` does not provide a `mare::task_ptr` and therefore become anonymous and cannot be part of a DAG.

Tasks that are part of a DAG must be created using `template<typename Body> mare::task_ptr mare::create_task(Body&&)`. This template method returns a task pointer that is used to set up dependencies between tasks. Once a the dependencies of a task are set, `void mare::launch(mare::task_ptr const&)` launches it:

```
1 mare::task_ptr t1 = mare::create_task([] {
2     printf("Hello World!\n");
3   });
4
5 //...
6 // Set up dependencies
7 // ..
8 // t1 is ready, launch it
9
10 mare::launch(t1);
```

The method `void mare::launch(mare::task_ptr const&)` informs the MARE runtime that the task is ready to execute as soon as a hardware context is available ∗and∗ after all its predecessors have executed. In the following example, task `t2` launches, but it will never execute because its predecessor `t1`

has not executed, and therefore, this task will not execute:

```
1 mare::task_ptr t1 = mare::create_task([]{
2     printf("Hello World from t1!");
3   });
4
5 mare::task_ptr t2 = mare::create_task([]{
6     printf("Hello World from t2!");
7   });
8
9 mare::after(t1, t2);
10 mare::launch(t2);
11
12 //wait_for will never return because t2 won't execute until t1
13 //does. However, t1 hasn't been launched.
14 mare::wait_for(t2);
15
```

Notice that launching a task means that it is not possible to add any new predecessors, although you can add successors. The reason is that, by launching the task, the programmer is asking the MARE runtime to execute the task as soon as possible. By the time the programmer tries to add a new predecessor to the task, the task might have already executed, and adding a predecessor to an already-executed task is not allowed.

Tasks can launch only once. Any subsequent calls to `mare::launch()` do not cause the task to execute again. The calls might, however, cause the task to be added to new groups. See Groups.

4.3.4.1 Launch and Reset

If `mare::task_ptr` will not be used after launching a task, consider using `launch_and_reset(mare::task_ptr)` instead of `launch(mare::task_ptr)`. The `launch_and_reset(mare::task_ptr)` function launches the task and resets the `mare::task_ptr` pointer in a single step. In this case, if there is only one `mare::task_ptr`, the MARE runtime assumes it is the sole owner of the task and does not need to protect access to it, thereby improving overall performance.

```
1 auto t = mare::create_task([] {
2     printf("Hello World!\n");
3   });
4
5 //assert will not fire
6 assert(t != nullptr);
7
8 mare::launch_and_reset(t);
9
10 //assert will not fire
11 assert(t == nullptr);
```

4.3.5 Waiting For a Task

The method `wait_for(mare::task_ptr const&)` does not return until the task that passed as an argument completes execution. It returns immediately once the task completes or cancels.

```
1 mare::task_ptr t1 = mare::create_task([] {
2     printf("Hello World!\n");
3   });
4 mare::launch(t1);
5 mare::wait_for(t1); //won't return until t1 has executed
```

Note: If `wait_for(mare::task_ptr const&)` is called from within a task, MARE context-switches the task and finds another one to run. If called from outside a task (i.e., the main thread), MARE blocks the thread until `wait_for(mare::task_ptr const&)` returns (see Interoperability).

Both `mare::wait_for(mare::task_ptr const&)` and `mare::wait_for(mare::unsafe_task_ptr)` are safe points. For information about safe points, see Interoperability.

4.3.6 Canceling a Task

There are three main ways to cancel an individual task. First, if you have a pointer to the task, you can use `mare::cancel(mare::task_ptr)`. Second, to cancel a running task from within the task body, call `mare::abort_task()`. And third, an unlaunched task is canceled when every `mare::task_ptr` pointing to the task goes out-of-scope. In this section, we examine each of these cancelation methods in detail.

4.3.6.1 mare::cancel

Use `mare::cancel(mare::task_ptr const&)` — or `mare::cancel(mare::unsafe_task_ptr)` — to cancel a task and its successors. What happens to the task when the programmer calls `mare::cancel(mare::task_ptr const&)` depends on the status of task.

4.3.6.1.1 Canceling a Task Before It Executes

If a task is canceled before it is launched, it never executes, even if it is launched later. In addition, the runtime will then cancel all successors and descendents, this is called "cancelation propagation". In the following example, we create two tasks `t1` and `t2` and create a dependency between them. Notice that, if any of the tasks execute, it will raise an assertion. In line 13, we cancel `t1`, which causes `t2` to be canceled as well. In line 16, we launch `t2`, but this has no effect because the task will not execute, as it was canceled when `t1` propagated its cancelation.

```
1 auto t1 = mare::create_task([]{
2     assert(false);
3   });
4
5 auto t2 = mare::create_task([]{
6     assert(false);
7   });
8
9 // Create dependencies
10 t1 >> t2;
11
12 // Cancel t1, which propagates cancelation to t2
13 mare::cancel(t1);
14
15 // Launch t2. Does nothing, t2 got canceleded via cancelation propagation
16 mare::launch(t2);
17
18 // Returns immediately, t2 is canceled.
19 mare::wait_for(t2);
```

Similarly, if a task is canceled after it is launched, but before it starts executing, it never executes and will propagate the cancelation request to its successors. In the following example, we create and chain three tasks, `t1`, `t2` and `t3`. In line 18, we launch `t2`, but it cannot execute because its predecessor has not yet executed. In line 21, we cancel `t2`, which means that it will never execute. Because `t3` is `t2`'s successor, it is also canceled - if `t3` had a successor, it would also be canceled.

```
1 auto t1 = mare::create_task([]{
2     printf("Hello World from t1!\n");
3   });
4
5 auto t2 = mare::create_task([]{
```

```
6     assert(false);
7   });
8
9 auto t3 = mare::create_task([]{
10     assert(false);
11   });
12
13 // Create dependencies
14 t1 >> t2 >> t3;
15
16 // Launch t2. It can't execute yet because
17 // t1 hasn't been launched.
18 mare::launch(t2);
19
20 // Cancel t2, which propagates cancelation to t3
21 mare::cancel(t2);
22
23 // Launch t1. It will execute because nobody
24 // canceled it.
25 mare::launch(t1);
26
27 // Returns after t1 completes execution
28 mare::wait_for(t1);
```

4.3.6.1.2 Canceling a Task While It Executes

Canceling a task that is executing is more involved because MARE uses `cooperative multitasking` . This means that, once a task is executing, it is not pre-empted unless it voluntarily cedes the processor (i.e., by calling `mare::wait_for(mare::task_ptr const&)`). Thus, it is up to the task to check periodically whether or not it has been canceled. Use `mare::abort_on_cancel()` inside a task body to abort the task immediately if the task, or any of the groups to which it belongs, have been canceled.

```
1 mare::task_ptr t = mare::create_task([]{
2     while(1) {
3       mare::abort_on_cancel();
4       printf("Waiting to be canceled.\n");
5       usleep(10);
6     }
7     assert(false); // This will never fire
8   }
9   );
10
11 //Launch t
12 mare::launch(t);
13
14 // Wait for 2 seconds.
15 sleep(2);
16
17 // Cancel task. Returns immediately.
18 mare::cancel(t);
19
20 // Wait for the task.
21 mare::wait_for(t);
22
```

In the example above, task `t` will never finish unless it is canceled. Task `t` is launched in line 12. After launching the task, we block for 2 seconds in line 15 to make sure that `t` is scheduled and prints its messages. In line 18, we ask MARE to cancel the task, which should be running by now. The method `mare::cancel()` returns immediately after it marks the task as "pending for cancelation". This means that `t` might still be executing after `mare::cancel(t)` returns. That is why we call `mare::wait_for(t)` in line 21, to makes sure we wait for `t` to complete its execution.

Note

We should stress that a task does not know whether someone has requested its cancelation unless it calls `mare::abort_on_cancel()` during its execution.

The method `mare::abort_on_cancel()` never returns if the task has indeed been canceled because it throws an exception that the MARE runtime catches. For this reason, we recommend that you use `Resource Acquisition Is Initialization (RAII)` to allocate and deallocate the resources used inside a task. If using RAII in your code is not an option, surround `mare::abort_on_cancel()` with `try - catch`, and call `throw` from within the `catch` block after the cleanup code:

```
1 mare::task_ptr t = mare::create_task([]{
2     while(1) {
3       try{
4         mare::abort_on_cancel();
5       }catch(mare::abort_task_exception const& e) {
6         //..do cleanup
7         throw;
8       }
9       printf("Waiting to be canceled.\n");
10       usleep(10);
11     }
12     assert(false); // This will never fire
13   }
14   );
15
16 //Launch t
17 mare::launch(t);
18
19 // Wait for 2 seconds.
20 sleep(2);
21
22 // Cancel task. Returns immediately.
23 mare::cancel(t);
24
25 // Wait for the task to complete.
26 mare::wait_for(t);
```

Warning

If we replace `throw` in line 7 of the previous example with `return`, the exception would not propagate to the runtime, MARE would not consider the task as canceled, and, therefore, its successors (if any) would not be canceled.

4.3.6.1.3 Canceling a Task After It Completes Execution

Canceling a task after it has been executed has no effect on the task, nor on its successors. In the following example, we launch `t1` and `t2` after we set up a dependency between them. On line 25, we cancel `t1` after it has completed. By then, `t1` has finished execution (we wait for it in line 21) so `cancel(t1)` has no effect. Thus, nobody cancels `t2` and `wait_for(t2)` in line 28 never returns.

```
1 auto t1 = mare::create_task([]{
2     printf("Hello World from t1!\n");
3   });
4
5 auto t2 = mare::create_task([]{
6     while (1){
7       mare::abort_on_cancel();
8       printf("Hello World from t1!\n");
9       usleep(100);
10     };
11   });
12
13 // Create dependencies
```

```
14 t1 >> t2;
15
16 // Launch tasks
17 mare::launch(t1);
18 mare::launch(t2);
19
20 // Wait for t1 to complete
21 mare::wait_for(t1);
22
23 // Cancel t1. Because it has already completed, it does not do
24 // cancelation propagation.
25 mare::cancel(t1);
26
27 // Will never return
28 mare::wait_for(t2);
```

4.3.6.2 mare::abort_task()

Running tasks call `mare::abort_task()` to cancel themselves and their successors. Consider the following example. We create two tasks, `t1` and `t2`, and create a dependency between them. The body of `t1` is very simple: it prints a message 10 times and then it aborts. We launch both and wait for `t1` to complete its execution in line 26. Because `t1` calls `mare::abort_task()`, it is canceled and propagates its cancelation to its successor,`t2`.

```
1 auto t1 = mare::create_task([]{
2     int i = 0;
3     while(true) {
4       printf("Hello World %d\n", i);
5       sleep(1);
6       i++;
7       if(i == 10)
8         mare::abort_task();
9     }
10     // This will never fire
11     assert(false);
12   });
13
14 auto t2 = mare::create_task([] {
15     // This will never fire
16     assert(false);
17   });
18
19 t1 >> t2;
20
21 //Launch tasks
22 mare::launch(t1);
23 mare::launch(t2);
24
25 // Wait for t1 to complete.
26 mare::wait_for(t1);
27
28 // Returns immediately, t2 is canceled.
29 mare::wait_for(t2);
```

4.3.6.3 Cancelation by Abandonment

When all the `mare::task_ptr`s referencing an unlaunched task go out of scope, the task is canceled and it propagates the cancelation to its successors. The reasoning is simple: a task `t` cannot launch without a task pointer, and none of its successors will ever be able to execute because `t` never executed.

```
1 void foo()
2 {
3   auto t1 = mare::create_task([] {
4       printf("Hello World from t1\n");
5       // This will never fire
6       assert(false);
```

```
7     });
8
9   auto t2 = mare::create_task([] {
10        printf("Hello World from t2\n");
11        // This will never fire
12        assert(false);
13      });
14
15    auto t3 = mare::create_task([] {
16        int i = 0;
17        while(i++ < 10)
18        {
19          printf("Hello World from t3\n");
20          sleep(1);
21        };
22      });
23
24    t1 >> t2;
25
26    mare::launch(t2);
27    mare::launch(t3);
28
29    //t1, t2 and t3 go out of scope
30  }
```

In the snippet above, we create three tasks `t1`, `t2` and `t3`, and create a dependency between the first two. We launch `t2` and `t3` in lines 26 and 27. `t2` cannot run because `t1` has not yet executed. In line 30, foo() ends and the three pointers go out-of-scope. `t1` is canceled because it is not yet launched. `t2` is canceled because `t1` propagated its cancelation. `t3` does not get canceled and will run even after `foo()` goes out-of-scope.

4.3.7 Task Attributes

Programmers can decorate tasks with attributes that help MARE make better scheduling decisions. The current MARE release supports just one attribute, but there are plans to include others in future releases.

4.3.7.1 Using Task Attributes

Use the template method `template <typename Attribute, typename ...Attributes> mare::task_attrs mare::create_task_attrs(Attribute const&, Attributes const& ...)` to create an object of type `mare::task_attrs` that summarizes all of the attributes of the task. In the following code snippet, we create a `task_attrs` object that includes the `mare::blocking` attribute.

```
1 mare::task_attrs attrs = mare::create_task_attrs(
      mare::attr::blocking);
```

Task attributes can be applied only when the task is created using the template method `mare::with_attrs(...)`:

```
1 auto body = []{ printf("Hello World from t's task body!\n");};
2 auto cancel_handler = [] {printf(" Hellow World from t's cancel handler!\n");};
3
4 mare::task_attrs attrs = mare::create_task_attrs(
      mare::attr::blocking);
5 mare::task_ptr t = mare::create_task(
      mare::with_attrs(attrs, body,
6                                                    cancel_handler));
```

In the example above we create two lambda methods: `body` and `cancel_handler`. The reason we need the latter is explained below.

4.3.7.2 Attribute: Blocking Task

A blocking task is defined as a task that depends on external (non-MARE) synchronization to make guaranteed forward progress. Typically, this includes completing I/O requests and other OS syscalls with indefinite run-time, but also busy-waiting. It does not include waiting on MARE tasks or groups using `mare::wait_for`.

There are two problems with blocking tasks. The first is that once a task executes, it will take over a thread in a MARE thread pool, thus preventing other tasks from executing in the same thread. Because a blocking task spends most of its time blocking on an event, we are essentially wasting one of the threads in the thread pool. When the programmer decorates a task with the `mare::attr::blocking` attribute, MARE makes sure that the thread pool does not waste a thread to the task.

The second problem has to do with cancelation. If a blocking task is canceled while it is waiting on an external event, we need to be able to wake the task up so that it can execute `mare::abort_on_cancel`. This is why blocking tasks require an extra lambda function, called the "cancel handler". A task's cancel handler is executed only once, and only if the task is running. In the following example, we create a blocking task using two lambdas, `body` and `cancel_handler`. After launching `t` and sleeping for a couple of seconds, we cancel it (line 30). Most likely, by the time we cancel `t`, it will be waiting on the condition variable. The cancel handler function calls wakes up the task body so that it can abort (line 10).

```
1 static std::mutex mutex;
2 static std::condition_variable cv;
3
4 auto attrs = mare::create_task_attrs(mare::attr::blocking);
5
6 auto body = [] {
7   printf("START  blocking task\n");
8   std::unique_lock<std::mutex> lock(mutex);
9   for (;;) {
10     mare::abort_on_cancel();
11     cv.wait(lock);
12   }
13   printf("STOP blocking task\n");
14 };
15
16 auto cancel_handler = [] {
17   printf("CANCEL blocking task\n");
18   std::lock_guard<std::mutex> lock(mutex);
19   cv.notify_all();
20 };
21
22 auto t = mare::create_task(mare::with_attrs(attrs, body, cancel_handler));
23
24 mare::launch(t);
25
26 // Wait for task to block
27 sleep(2);
28
29 // Cancel task. It will call t's cancel_handler
30 mare::cancel(t);
31
32 //Wait for t to complete
33 mare::wait_for(t);
34
```

4.3.8 Choosing the Right Granularity for a Task

The performance of MARE applications is subject to all the issues discussed in Parallel Processing Tutorial. In this section, we discuss one of these particular issues, the task granularity. The MARE runtime has certain overheads introduced by the need to manage and schedule tasks properly, while respecting all dependencies, and ensuring that tasks can be executed concurrently. The programmer can amortize the overheads of the MARE runtime by following the simple guidelines presented here.

Programmers can control two key aspects that significantly affect application performance: task granularity (the amount of work in a task) and number of tasks in the application. Obviously, these two aspects are correlated, since the amount of work that the application performs should be the same, regardless of partitioning. Programmers must make the following tradeoff: the larger the number of tasks and the finer the granularity, the more overhead in the runtime to amortize. The smaller the number of tasks (and thus the coarser the granularity), the less concurrency available for the runtime to exploit.

To maximize performance, programmers should follow three basic guidelines:

1. Consider the total amount of work in the application: If the work is trivial, the effort of parallelization may not pay off from a performance standpoint. Other metrics, such as energy consumption may benefit from multicore utilization. However, the total amount of work should at least compensate for the overhead.
2. Consider the degree of parallelism: The more independent tasks the runtime has to schedule, the better the utilization of the multicore system. At a minimum, having at least double the number of cores is a good first approximation.
3. Consider the granularity of the tasks: As discussed earlier, coarser tasks amortize the overhead of parallelization. Therefore, we recommend tasks to be larger than few microseconds (the actual granularity will depend on the particular platform on which the application is deployed). For tasks larger than 50 microseconds, the overhead is completely amortized even when creating tens of thousands of tasks.

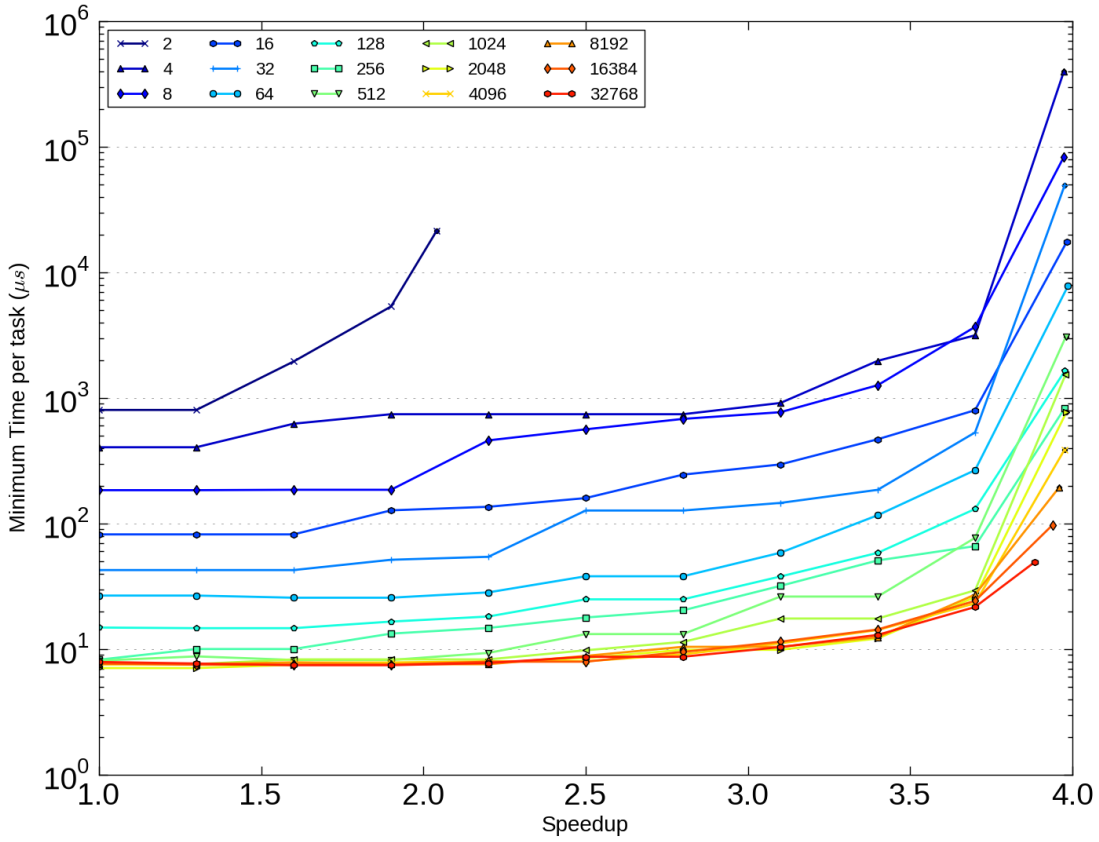


Figure 4-3 Speedup as a function of task granularity and total number of tasks.

To determine if an application can benefit from parallelization, the figure above plots the speedup that can be obtained as a function of the task granularity. We plot a family of curves, each for a different number of tasks in the application, between 2 and 32768. The tasks defined here are totally independent, representing iterations of an integer kernel with two nested loops. The tasks are launched simultaneously to maximize the pressure on the MARE scheduler. For the plot, we used a Snapdragon 800 platform with MARE running on four cores. On this configuration, the recommended minimum work per task is 10 microseconds. Other platforms will have different breakeven points. Dependencies and group membership will add to the overhead; however, tasks around 10 microseconds will be able to amortize the overhead. This is also an area where we are putting significant effort in optimizing the performance. Therefore, future releases of MARE should expect even better performance at finer granularity.

4.4 Groups

One of the most common parallel programming patterns is fork and join. The idea is quite intuitive. At the fork, the application splits the job-to-be-done into many tasks. At the join, the application waits for all of them to complete before continuing with its execution.

An example of fork and join could be styling a website. During the styling phase, a parallel web browser could traverse the DOM tree and spawn one task per DOM element. Each of the tasks would be responsible for styling an element. The browser can only render the page once all the styling tasks complete.

Calling `mare::wait_for` for each task is cumbersome: the programmer would have to store the task pointers into some data structure, and call `mare::wait_for` for each of them after visiting all nodes. A group is a MARE abstraction that allows the programmer to wait for a set of tasks to complete, relieving the programmer from having to wait for each task separately.

It is easy to imagine that, in addition to the styling tasks, the parallel browser would have launched other tasks to do HTML parsing, Javascript execution, etc... A logical design decision would be to have all the tasks working on the same page to belong to the same group. Notice that the styling tasks would need to belong to two groups: the "Page XYZ" group, and the "Page XYZ-styling" group. MARE supports tasks tasks that belong to multiple groups.

`mare::wait_for(mare::group_ptr const&)` is not the only operation that can be done with groups. The method `mare::cancel(mare::group_ptr const&)` cancels all the tasks in a group. In our parallel browser example, this would allow the browser to easily cancel all the tasks in the "Page XYZ" group when the user decides to navigate to a new page before the current one displays.

In summary, ∗groups are sets of tasks that can be canceled or waited for as a unit∗.

4.4.1 Group Creation

Use `mare::create_group()` to create a new group.

```
1 // Create group
2 mare::group_ptr g = mare::create_group();
```

Use `mare::create_group(std::string const& name)` to create a new group called <name>. Group names are only used for debugging applications, MARE does not check for duplicate group names.

```
1 // Create group named "Example"
2 mare::group_ptr g = mare::create_group("Example");
```

Use `mare::group_name(mare::group_ptr const&)` to get the group name:

```
1 // Create group named "Example"
2 mare::group_ptr g = mare::create_group("Example");
3 std::string name = mare::group_name(g); // returns "Example"
```

Warning

In the current MARE release, there can only exist 32 groups at a given time. Trying to create a 33rd one will result in an exception.

4.4.1.1 The group_ptr pointer

The method `mare::group_ptr mare::create_group(std::string const& name)` returns an object of type `mare::group_ptr`, which is a custom smart pointer to the group object. `mare::group_ptr` pointers behave similarly to `mare::task_ptr`. Therefore, groups are reference counted, and they are automatically destroyed when there are no more `mare::group_ptr` pointers pointing to them. This means that even if the user has no pointers to the group, MARE will not destroy the group until all its tasks complete.

The current MARE release does not support `mare::unsafe_group_ptr`, but a future one will.

4.4.2 Adding Tasks to Groups

There are three ways to add a task into a group:

Create and Launching

By creating a new task and immediately launching it using `template<typename Body> void mare::launch(mare::group_ptr const&, Body&&)`. Use this method when the task does not have predecessor or successors. This is the most performant way to create and launch a task in MARE. This is because the MARE runtime knows that the programer has no pointer to the task and it can perform aggressive optimizations. It is for this reason that `template<typename Body> void mare::launch(mare::group_ptr const&, Body&&)` does not return a `mare::task_ptr`. Use this method as much as you can:

```
1 // Create group named "Example"
2 mare::group_ptr g = mare::create_group("Example");
3
4 // Create and launch tasks into g
5 for (int i=0; i < 1000; i++)
6   mare::launch(g, []{ printf("Hello World!\n"); });
7
8 // Wait for all the tasks in group g to complete
9 mare::wait_for(g);
```

Launching

By launching an existing task using `void mare::launch_and_reset(mare::group_ptr const&, mare::task_ptr&)` or`void mare::launch(mare::group_ptr const&, mare::task_ptr const&)`. Use these methods when the task is part of a DAG. The former is more performant than the latter because the MARE runtime can infer that the programmer has no pointer to the task. Thus, we recommend that you use `void mare::launch_and_reset(mare::group_ptr const &, mare::task_ptr&)` if you do not need the `mare::task_ptr` pointer after the method call.

```
1 // Create Example group
2 mare::group_ptr g = mare::create_group("Example");
3
4 // Create tasks
5 mare::task_ptr t1 = mare::create_task([]{ printf("Hello World from t1!\n");}
      );
6 mare::task_ptr t2 = mare::create_task([]{ printf("Hello World from t2!\n");}
      );
7
8 // Launch t1 into g,
9 mare::launch(g, t1);
10
11 // Use t1
12 mare::after(t1, t2);
13
14 // t1 pointer no longer needed, reset it
15 t1.reset();
16
17 // Launch t2 into g and reset task pointer
18 mare::launch_and_reset(g, t2);
19
20 // Wait for tasks to complete
21 mare::wait_for(g);
```

Joining

By explicitly adding it into the group using: `void mare::join_group(mare::group_ptr const &, mare::task_ptr const &)`. This method is slow, and it should be employed scarcely. Use it only when you absolutely need the task to belong to a group (perhaps you want to prevent the group from being empty so you can wait on it somewhere else), but you're not ready to launch the task just yet.

```
1 // Create group g
2 mare::group_ptr g = mare::create_group("Example");
3
4 // Create task t1
5 mare::task_ptr t1 = mare::create_task([]{ printf("Hello World from t1!\n");}
      );
6
7 // Add t1 to g. Don't do this often
8 mare::join_group(g, t1);
9
10 // Launch t1. Because it belongs to group g,
11 // there is no reason to launch it into the
12 // same group again.
13 mare::launch(t1);
14
15 // t1 no longer needed, reset pointer
16 t1.reset();
17
18 // Wait for tasks in group g to complete
19 mare::wait_for(g);
```

Regardless of the method you use, the following rules always apply:

- Tasks stay in the group until they complete execution. Once a task is added to a group, there is no way to remove it from the group.
- Once a task belonging to multiple groups completes execution, MARE removes it from all the groups it belongs.
- Neither completed nor canceled tasks can join groups.
- Tasks can not be added to a canceled group.

4.4.2.1 Adding a Task to Multiple Groups

There are two ways to add a task to more than group:

- By launching it into each of the groups. For example, to add task `t` to groups `g1` and `g2`, we would do the following:

```
1 // Create task
2 mare::task_ptr t = mare::create_task([]{ printf("Hello World!\n");});
3
4 // Create groups
5 mare::group_ptr g1 = mare::create_group("Example 1");
6 mare::group_ptr g2 = mare::create_group("Example 2");
7
8 // Launch t into g1 and g2
9 mare::launch(g1, t);
10 mare::launch(g2, t);
11
12 mare::wait_for(t);
13
```

Notice that, in the example above, `t` joins both `g1` and `g2`, but it only executes once. Therefore, the code snippet outputs a single 'Hello World!'. However, You must understand that `t` might never join `g2` because it might complete execution before the first launch returns. Remember that completed tasks can never join groups.

- By creating a new group that is the intersection of all the groups where the task needs to launch, and then launch the task into it. `mare::group_ptr mare::intersect(mare::group_ptr const &, mare::group_ptr const &)` returns a group pointer to a group that represents the intersection of the two groups passed as arguments. This method is more performant than repeatedly launching the same task into different groups.

```
1 // Create task
2 mare::task_ptr t = mare::create_task([]{ printf("Hello World!\n");});
3
4 // Create groups
5 mare::group_ptr g1 = mare::create_group("Example 1");
6 mare::group_ptr g2 = mare::create_group("Example 2");
7
8 mare::group_ptr g12 = mare::intersect(g1, g2);
9
10 // Launch t into g1 and g2
11 mare::launch(g12, t);
```

4.4.2.1.1 Group Intersection

It is important to understand what group intersection really means, because it might appear counterintuitive. `mare::intersect` returns a pointer to a group that represents the intersection of two or more groups. Launching a task into the intersection group means simultaneously launching it into all the groups that are part of the intersection. Intersection groups do not count towards the 31 maximum number of simultaneous groups in the application.

For example, the following code snippet shows an application with two groups, `g1` and `g2`, with 100 and 200 tasks in each, respectively.

```
1 // Create groups
2
3 mare::group_ptr g1 = mare::create_group("Group 1");
4 mare::group_ptr g2 = mare::create_group("Group 2");
5
6 for(int i=0; i<100; i++)
7   mare::launch(g1, []{
8       //... Do something
9     });
10
11 for(int i=0; i<200; i++)
12   mare::launch(g2, []{
13       //... Do something
14     });
15
16 mare::group_ptr g12 = mare::intersect(g1, g2);
17
18 // Returns immediately. g12 is empty
19 mare::wait_for(g12);
20
21 // Return only after tasks in g1 and g2 complete
22 mare::wait_for(g1);
23 mare::wait_for(g2);
24
25 mare::task_ptr t = mare::create_task([] {
26     //... Calculate the Ultimate Question of Life, the Universe, and Everything
27     printf("42\n");
28   });
29
```

```
30 mare::launch(g12, t);
31
32 // All will return after the task prints 42
33 mare::wait_for(g1);
34 mare::wait_for(g2);
35
```

In line 16, we intersect `g1` and `g2` into `g12`. The returned pointer, `g12`, points to an empty group because no task belongs to both `g1` and `g2` yet. Therefore, `mare::wait_for(g12)` in line 19 returns immediately. The `wait_for` calls in lines 22 and 23 only return when their tasks complete. In line 30, we launch `t` into `g12`. The `wait_for` calls in lines 31 and 32 only return after `t` completes execution because `t` belongs to both `g1` and `g2` (and, of course, `g12`).

Note: You can use the `&` operator instead of `mare::intersect`:

```
1 // Create groups
2 mare::group_ptr g1 = mare::create_group("Group 1");
3 mare::group_ptr g2 = mare::create_group("Group 2");
4
5 // Get pointer to intersection groups:
6 mare::group_ptr g12 = g1 & g2;
7 mare::group_ptr g21 = g2 & g1;
8
9 // This assert will never fire
10 assert(g12 == g21);
11
```

You must keep in mind that group intersection is a somewhat expensive operation. If you need to intersect groups repeatedly, just do it once and keep the pointer to the group intersection alive.

```
1 // Create groups
2 mare::group_ptr g1 = mare::create_group("Example 1");
3 mare::group_ptr g2 = mare::create_group("Example 2");
4 mare::group_ptr g12 = g1 & g2;
5
6 // Launch 1000 tasks into g1 and g2
7 for (int i = 0; i < 1000; i++) {
8   mare::launch(g12, []{ printf("Hello World!\n");});
9  }
```

Therefore, the code snippet above is much faster than the one below:

```
1 // Create groups
2 mare::group_ptr g1 = mare::create_group("Example 1");
3 mare::group_ptr g2 = mare::create_group("Example 2");
4
5 // Launch 1000 tasks into g1 and g2
6 for (int i = 0; i < 1000; i++){
7   mare::launch(g1 & g2, []{ printf("Hello World!\n");});
8  }
9
```

Consecutive calls to `mare::intersect` with the same groups pointer as arguments return a pointer to the same group. In addition, group intersection is commutative:

```
1 // Create groups
2 mare::group_ptr g1 = mare::create_group("Group 1");
3 mare::group_ptr g2 = mare::create_group("Group 2");
4
5 // Get pointer to intersection groups:
6 mare::group_ptr g12 = g1 & g2;
7 mare::group_ptr g21 = g2 & g1;
8
9 // This assert will never fire
10 assert(g12 == g21);
11
```

and associative:

```
1 // Create groups
2 mare::group_ptr g1 = mare::create_group("Group 1");
3 mare::group_ptr g2 = mare::create_group("Group 2");
4 mare::group_ptr g3 = mare::create_group("Group 3");
5
6 // Get pointers to intersection groups:
7 mare::group_ptr g12_3 = (g1 & g2) & g3;
8 mare::group_ptr g1_23 = g1 & (g2 & g3);
9 mare::group_ptr g2_13 = g2 & (g1 & g3);
10
11 // These asserts will never fire
12 assert(g12_3 == g1_23);
13 assert(g12_3 == g2_13);
```

4.4.2.2 Waiting For a Group

`mare::wait_for(mare::group_ptr const &)` does not return until all the tasks in it have completed execution or have been canceled.

```
1 mare::group_ptr g = mare::create_group("Example");
2
3 // Launch 1000 tasks into g
4 for (int i = 0; i < 1000; i++){
5   mare::launch(g, []{ printf("Hello World!\n");});
6  }
7
8 // Wait for tasks to complete
9 mare::wait_for(g);
10
```

Note: As in `mare::wait_for(mare::task_ptr const &)`, if `mare::wait_for(mare::group_ptr const &)` is called from within a task, MARE context switches the task and finds another task to run. If called from outside a task, it blocks the calling thread until it returns.

Waiting for an intersection group means that MARE returns once the tasks in the intersection group have completed or canceled.

For example, `mare::wait_for(g12)` in the following code returns immediately, because there are no tasks in `g12`. Neither `mare::wait_for(g1)` nor `mare::wait_for(g2)` would return.

```
1 // Create groups
2 mare::group_ptr g1 = mare::create_group("Example 1");
3 mare::group_ptr g2 = mare::create_group("Example 2");
4 mare::group_ptr g12 = g1 & g2;
5
6 // Create and launch two tasks that never end
7 mare::launch(g1, []{
8     while(1) {}
9   });
10
11 mare::launch(g2, []{
12     while(1) {}
13   });
14
15 // Returns immediately because there are no
16 // tasks that belong to both g1 and g2
17 mare::wait_for(g12);
18
19 // Never returns
20 mare::wait_for(g1);
21
```

`mare::wait_for(mare::group_ptr const &)` and is a safe point. For information about safe points, see Interoperability.

4.4.3 Group Cancelation

Use `mare::cancel(mare::group_ptr const &)` to cancel all the tasks in a group. Canceling a group means that:

- The group tasks that have not started execution will never execute.
- The group tasks that are executing will be canceled only when they call `mare::abort_on_cancel`. If any of these executing tasks is a blocking task, MARE will execute its cancel handler if they had not executed it before.
- Any tasks added to the group after the group is canceled will also be canceled.

In the following example, we launch 10000 tasks and then sleep for some time so that a few of those 10000 tasks are done, a few others are executing and a large majority are waiting to be executed. In line 24 we cancel the group. This means that next time the running tasks execute `mare::abort_on_cancel()` they will see that their group has been canceled and will abort. `wait_for(g)` will not return before the running tasks end their execution — either because they call `mare::abort_on_cancel()` or because they finish writing all the messages.

```
1 // Create group
2 auto g = mare::create_group("example");
3
4 // Create lambda for task body.
5 auto body = [] () {
6   for (int i = 0; i < 5; ++i) {
7     mare::abort_on_cancel();
8     printf("Hello world %d\n", i);
9     sleep(1);
10   }
11 };
12
13 // Launch many tasks
14 for (int j = 0; j < 10000; ++j) {
15   mare::launch(g, body);
16  }
17 // Wait for a little bit, to give a
18 // few tasks time to complete their
19 // execution
20 sleep(10);
21
22 // Cancel group and wait for the
23 // running tasks to complete
24 mare::cancel(g);
25 mare::wait_for(g);
```

Like `mare::cancel(mare::task_ptr const &)`, `mare::cancel(mare::group_ptr const &)` returns immediately. Use `mare::wait_for(mare::group_ptr const &)` after `mare::cancel(mare::group_ptr const &)` to block execution until the group is empty.

Warning

Once a group is canceled, it cannot be "uncanceled".

4.5 Heterogeneous Compute Overview

Most of the current hardware platforms, from desktops to tablets to smartphones, are built around multicore and heterogeneous systems on a chip (SoC). As the first step towards exploiting this heterogeneity, GPU (Graphics Processing Unit) support has been added to MARE. OpenCL C (an open and royalty free standard) is used as the kernel language for writing code which runs on the GPU. Users can continue to use

the MARE abstractions of tasks, groups and pfor_each pattern to launch the OpenCL C kernels and need not worry about execution model details. The MARE runtime picks the best GPU device available on the platform, manages the underlying execution model for the device and provides an important abstraction of Shared Virtual Memory (SVM) through `mare::buffer`. The use of `mare::buffer` relieves the programmer from having to manage data movement explicitly. It also future proofs applications by transparently leveraging future hardware SVM support, relieving the programmer of the burden of changing the application. Lets look at a simple example below which does addition of two vectors on the GPU, and highlights MARE's heterogeneous compute capabilities.

```
1 #ifdef HAVE_CONFIG_H
2 #include <config.h>
3 #endif
4
5 #include <mare/mare.h>
6 #include <mare/alignedallocator.hh>
7 #include <mare/patterns.hh>
8
9 //Create a page aligned allocator.
10 template<class T, size_t Alignment>
11 using aligned_vector = std::vector<T, mare::aligned_allocator<T, Alignment> >;
12 template<class T>
13 using page_aligned_vector = aligned_vector<T, 4096>;
14
15 //Create a string containing OpenCL C kernel code.
16 #define OCL_KERNEL(name, k) std::string const name##_string = #k
17
18 OCL_KERNEL(vadd_kernel,
19   __kernel void vadd(__global float* A, __global float* B, __global float* C,
20                      unsigned int size) {
21     unsigned int i = get_global_id(0);
22     if(i < size)
23        C[i] = A[i] + B[i];
24   });
25
26 int
27 main(void)
28 {
29   //Initialize the MARE runtime.
30   mare::runtime::init();
31
32   //Create input vectors
33   page_aligned_vector<float> a(1024);
34   page_aligned_vector<float> b(a.size());
35
36   //Initialize the input vectors
37   for (size_t i = 0; i < a.size(); ++i) {
38     a[i] = i;
39     b[i] = a.size() - i;
40   }
41
42   //Create mare::buffer with initialization data.
43   auto buf_a = mare::create_buffer(a.data(), a.size());
44   auto buf_b = mare::create_buffer(b.data(), b.size());
45
46   //Create mare::buffer with no host ptr & initialization data.
47   auto buf_c = mare::create_buffer<float>(a.size());
48
49   //Name of the OpenCL C kernel.
50   std::string kernel_name("vadd");
51
52   //Create a kernel object,
53   auto gpu_vadd = mare::create_kernel<mare::buffer_ptr<const float>,
54                                       mare::buffer_ptr<const float>,
55                                       mare::buffer_ptr<mare::out<float>>,
56                                       unsigned int>(vadd_kernel_string,
57                                                     kernel_name);
58
59   //Create a task attribute to mark the task as a gpu task.
```

```
60   auto attrs = mare::create_task_attrs(mare::attr::gpu);
61
62   unsigned int size = a.size();
63
64   //Create a mare::range object, 1D in this case.
65   mare::range<1> range_1d(a.size());
66
67   //Create a ndrange_task
68   auto gpu_task =
69     mare::create_ndrange_task(range_1d,
70                               mare::with_attrs(attrs, gpu_vadd,
71                                                buf_a, buf_b, buf_c, size));
72
73
74   //Launch the task
75   mare::launch(gpu_task);
76
77   //Wait for task completion.
78   mare::wait_for(gpu_task);
79
80   //compare the results.
81   for(size_t i = 0; i < a.size(); ++i) {
82     MARE_INTERNAL_ASSERT(a[i] + b[i] == buf_c[i] && buf_c[i] == a.size(),
83                          "comparison failed at ix %zu: %f + %f == %f == %zu",
84                          i, a[i], b[i], buf_c[i], a.size());
85   }
86
87   //shutdown the mare runtime
88   mare::runtime::shutdown();
89 }
90
```

Lines 10-13 Create vectors using a page aligned allocator; creating host memory using an aligned allocator may optimize some of the copies done by the runtime. Although not mandatory, it is recommended to do all host allocations using an aligned allocator.

Lines 16-24 Create an OpenCL C kernel string.

Lines 33-40 Allocate host memory using the page aligned allocator for `buf_a` and `buf_b` and initializes them.

Lines 43-47 Create `mare::buffer`(s) `buf_a` and `buf_b` using host memory allocated earlier, it also creates `buf_c` without any host memory as backing store. If `buf_c` is not used before kernel launch then the first time it's used in a kernel, it will be allocated in device-accessible host memory. Hence subsequent accesses to `buf_c` can use the same host memory. However if `buf_c` is used before the kernel launch then MARE runtime allocates host memory as backing store and `buf_c` behaves similarly to `buf_a`.

Lines 53-58 Create a kernel object whose signature matches the OpenCL C kernel string created earlier. The template parameters dictates if buffer data is copied to device before launch or if buffer data is copied back from device when accessed on host. Here `buf_a` and `buf_b` are specified as const buffers i.e. they are not modified by the kernel, hence when they are accessed on host after kernel launch, data is not copied back from the device. Similarly buf_c is specified as `out<float>` which indicates it's an output only buffer and hence data is not copied to the device before kernel launch, however when `buf_c` is accessed after kernel launch, it's copied back from the device.

Lines 60-69 Create a gpu task attribute, a 1D range object and a ndrange_task with task attribute set to gpu. The kernel object passed to `mare::create_ndrange_task` is used for compile time type checking of actual kernel arguments passed.

Lines 73-76 Launch the gpu task and waits for it's completion.

Lines 79-83 Compare the results; note that the first time `buf_c` is accessed on the host after kernel launch, it is copied back from the device.

4.5.1 Buffers Overview

Different parts of an application may execute best on different devices. Orchestrating data movement for such an application is challenging and error prone. `mare::buffer` provides an important abstraction of Shared Virtual Memory(SVM) that relieves the programmer from having to manage data movement explicitly. On hardware that supports SVM directly, some of the data movements can be avoided. Currently only GPU devices are supported and all further references to device refer to GPU. Buffers can be created in different ways as shown below:

Creating a `mare::buffer` without any initialization (host) data.

```
// Create a buffer with 1k elements.
auto buf_a = mare::create_buffer<float>(1024);
```

Creating a `mare::buffer` with existing host data.

```
// Use the aligned allocator to allocate host data
template<class T, size_t Alignment>
using aligned_vector = std::vector<T, mare::aligned_allocator<T, Alignment> >;
template<class T>
using page_aligned_vector = aligned_vector<T, 4096>;

// allocate host memory for 1k floats.
page_aligned_vector<float> host_ptr(1024);

// Create a buffer from existing host data
auto buf_a = mare::create_buffer(host_ptr, 1024);
```

Warning

Once a buffer has been created using `ptr`, manipulating the underlying data directly using the raw `ptr` is undefined.

Buffers are managed using the following three principles:

- **Allocate on first use**: Memory required as backing store for mare::buffer is allocated based on where it gets used. Some of the scenarios are detailed below:
  - Creating a `mare::buffer` without any initialization (host) data.
    - Used on host before kernel launch, MARE runtime allocates host memory as backing store, this `mare::buffer` behaves similarly to one allocated with user provided host pointer.
    - Used only in kernel, memory is allocated in device-accessible host memory, any further use of `mare::buffer` on host will use the same memory.
  - Creating a `mare::buffer` created with existing host data.
    - Used only on host, no device memory will be created.
    - Used in kernel launch; a copy is done depending on the type of kernel arguments (see the table below).
- **Copy when required**: Data movement happens only when needed; it is dictated by the kernel parameter types used in `mare::create_kernel` and the actual arguments passed to `mare::create_ndrange_task`.
  - copy-in copies to device
  - copy-out copies from device

Kernel Parameter –>	T / inout<T>	const T / in<T>	out<T>
Buffer Type			
T	copy-in and copy-out	copy-in, no copy-out	no copy-in, only copy-out
const T	Invalid, compile time error.	no copy-out	Invalid, compile time error.

- **No implicit thread safety**: The following applies for concurrent access of `mare::buffer`.
  - Cannot be accessed concurrently across multiple devices
  - `mare::buffer` elements can be accessed concurrently after calling `mare::buffer::sync` (GPU implicitly synchronizes buffers that are kernel arguments before kernel execution)
  - `mare::buffer::sync` & other `mare::buffer` APIs are not thread-safe
  - Accessing `mare::buffer` elements triggers an implicit copy when required; consequently `mare::buffer` elements cannot be accessed concurrently from tasks or threads without prior synchronization using `mare::buffer::sync`

For using Heterogeneous Compute API on different platforms,

please refer to the requirements specified in Installing MARE

4.6 Interoperability

The MARE programming model isolates programmers from threads; however, MARE applications are multithreaded. In this section we discuss some of the interoperability issues that arise from using threads in the MARE runtime. A MARE application starts in a main thread and may create other threads. In addition, `mare::runtime::init()` creates a thread pool that executes MARE tasks. A MARE task might be created in any thread, and other operations, such as launching and executing on any other threads. While any thread in the application can call `mare::create_task()` and `mare::launch()`, only a thread from MARE's thread pool can execute a MARE task. The only guarantees regarding which threads a task is executing are defined below.

4.6.1 Safe Points

Safe points are MARE API methods where the following property holds: the thread on which the task executes before the API call might not be the same as the thread on which the task executes after the API call.

Following is a comprehensive list of the safe points in the current MARE relase:

- `mare::wait_for(mare::task_ptr)`
- `mare::wait_for(mare::unsafe_task_ptr)`
- `mare::wait_for(mare::group_ptr)`

4.6.2 Using MARE with the Fork() System Call

The Unix `fork()` system call is designed to duplicate the current process as a new process. This call is commonly used within shells to start new commands, within web servers to handle new connections in a separate process, and within web browsers to implement security between different browser tabs.

An important limitation of `fork()` is that it copies the memory of the process, but starts the child with only one thread, cloned from the thread that made the fork() system call. This is a known problem for multithreaded programs, and MARE is no exception. Calling `fork()` from a task running in the thread pool starts the new process with only one thread from the pool, and no other threads would exist. Tasks running on the other threads would be copied in an inconsistent state, and the output would be indeterminate. MARE implements various features to prevent this misuse of `fork()`.

No calls to `fork()` are allowed after `mare::runtime::init()` is called and before `mare::runtime::shutdown()` is invoked. MARE generates a runtime error message in this situation. Once `mare::runtime::shutdown()` is called, the thread pool is destroyed and `fork()` can be called after this point — assuming that the application does not have threads other than MARE's. If the application requires using `fork()`, it should be called either before `mare::runtime::init()` or after `mare::runtime::shutdown()`, and attention should be paid to the use of other threads in the application.

4.6.3 Using MARE with TLS-aware Libraries

There are other libraries, such as libraries that use Thread Local Storage (TLS) that also interact with the MARE runtime in non-intuitive ways. Remember that when MARE tasks execute, they stay on the same thread until they complete execution or they arrive at a safe point. In most cases this is not a problem; however, when a task makes calls to libraries that use TLS, there are a number of issues that we discuss below.

Following are examples of common TLS-aware libraries:

Xlib

The Xlib libraries are typically not thread-safe, although each implementation is different. It is not possible to perform display operations from two threads at the same time because this could corrupt internal data structures. While multiple threads can be used, the programmer must ensure that only one thread can be using Xlib at any time.

UI Toolkits

User interface toolkits —such as QT— typically have a main thread which is dedicated to processing input events, manipulating a display, and then sleeping until more input occurs. It is important that control is returned to the UI toolkit as soon as possible to ensure that the user experience is smooth and uninterrupted. If a call from a different thread is made to a function that manipulates the UI or triggers an event, the toolkit may corrupt a data structure, or detect this and generate an error message.

OpenGL

Each OpenGL implementation varies in how it can be used with multiple threads. In typical usage, you create an OpenGL context in the thread where you intend to use it. The OpenGL library then sets internal state information into TLS. This internal state is used so that when calls are made to OpenGL, you do not need to pass the context around each time. However, the TLS is set for only one thread. So

if you try to make an OpenGL call from a different thread, the implementation may fail. Some implementations allow multiple contexts, with each context being created on the thread where it will be used. With multiple contexts, some implementations also allow parallel access to the OpenGL library, although this support varies depending on the vendor. Hardware implementing OpenGL typically uses some kind of command buffer, which can force a sequential ordering of commands. Therefore, trying to implement calls to OpenGL in parallel may not provide any benefit, and may actually slow things down due to contention on the mutex used to protect the command buffer. An OpenGL application is typically used with some kind of user interface and event handler, which will be running on the main thread. So it is recommended that you perform your OpenGL calls in the same thread as the user interface.

While these are limitations that need to be taken into consideration, it is still possible to exploit parallelism using MARE in these types of applications. For example, let us consider the case of a game with physics simulation, where the user can click on the display to launch spheres into a room. In a nonparallel implementation, the user touches the display, which generates a UI event. The UI thread wakes up and processes the UI event, which needs to generate the new sphere in the physics simulation. The physics simulation runs for the time required to compute the result. The location of all objects in the physics simulation is then traversed, and OpenGL calls are made to draw the scene. The OpenGL buffers are then swapped onto the display, and the thread goes back to sleep to wait for either a UI event, or a timeout to refresh the display with no change.

When analyzing the previous example, the bulk of the calculations are performed in the physics engine. This is very computationally expensive and where the most optimizational work can be applied. So MARE can be used here to perform the computation in parallel, assuming the underlying implementation supports this. The user breaks down the parts of the simulation into a suitable number of tasks, specifies dependencies, and then launches them with MARE. When the tasks are launched, the thread does a `wait_for()` until the tasks have completed. In the meantime, the thread pool begins executing the tasks, which spreads the computational load across all available processors. When the tasks are done, the `wait_for()` will return, and execution on the main thread can continue with the calls to OpenGL for rendering. With this arrangement, you can see that the operations that are thread-sensitive are performed in one thread, guaranteeing safe use of libraries such as OpenGL and the user interface toolkit. Many computationally intensive OpenGL applications are written with an event loop very similar to that described above, so these changes should be relatively simple to implement to take advantage of MARE.

4.6.4 Scaling Memory Allocation Performance

The scalability of the default memory allocator for your system can have a profound effect on the scalability of a MARE application. The developers have experimented with several publically available allocators and, at the time of this writing, have identified `jemalloc` to have the right properties to allow a MARE application for scale well. See your platform documentation on how to interpose this library.

Instructions for Android

The lack of scalability is particularly apparent when using the memory allocator that comes with the `Android NDK`. As of Android 4.0, Ice Cream Sandwich, it is possible to run applications with wrappers to identify alternate facilities, such as memory allocators. To use an alternate memory allocator:

1. copy the memory allocator library to your device

2. find the process name of your application and set the wrap property:

```
PROCNAME=com.app
adb shell setprop wrap.$PROCNAME LD_PRELOAD=/path/jemalloc/jemalloc.so
```

4.6.5 Avoid the Use of C++ iostream and stringstream Libraries

Warning

The C++11 standard indicates that the iostream library should be thread safe. As of this writing, the developers have experienced stability issues on some platforms, such as Android and OSX. In order to maximize portability, MARE applications should avoid using `cout` and `cerr` to perform asynchronous writes, especially to the console. It is recommended to use the C-based `stdio printf` routines. On Android, the developers have experienced additional issues with `stringstream` objects.

5 Parallel Processing Tutorial

5.1 Abstract

In this tutorial we introduce general principles of parallel programming with an emphasis on task-based parallel programming models. We first introduce scaling as a metric of evaluating the potential speedup that an algorithm can obtain. Then we discuss different parallel programming paradigms and a number of optimizations for parallel code. We illustrate our discussion with examples using the MARE programming model.

5.2 Parallel Speedups

Amdahl [2] put forward an argument that the maximum speedup that can be obtained by a parallel algorithm is bounded by the serial fraction of the program. Intuitively, even if we could execute the parallel fraction infinitely fast (zero time), the serial fraction will determine the total execution time. This argument, commonly known as *Amdahl's Law*, can be summarized by the following equation, when considering N parallel processors:

$$ParallelSpeedup = \frac{s+p}{s+p/N} = \frac{1}{s+p/N},$$

where $s+p=1$, representing the serial and parallel fractions of the program, respectively. Using Amdahl's law, the speedup that can be obtained with 8 processors as a function of the serial fraction is illustrated in

Figure Amdahl.

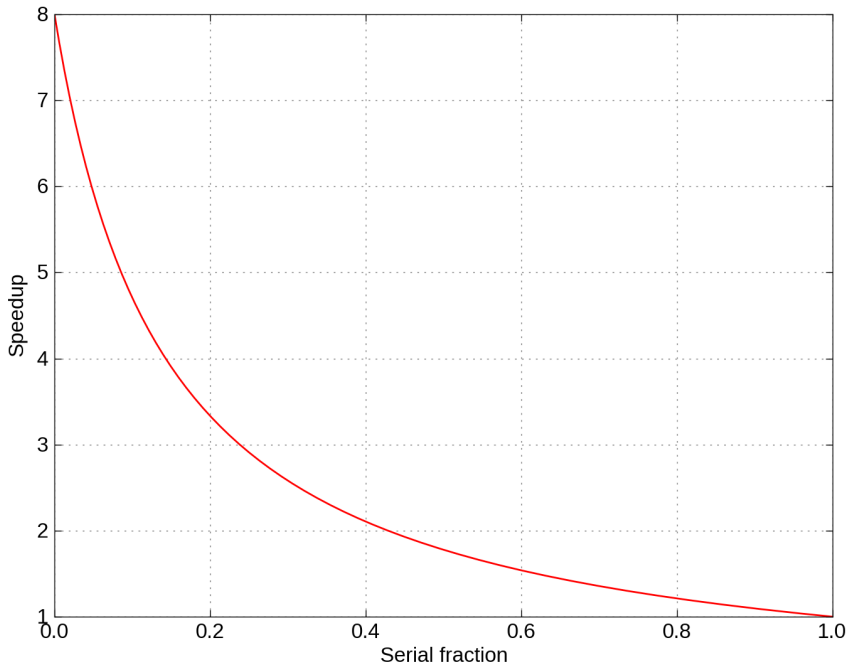


Figure 5-1 Theoretical speedup on 8 processors using Amdahl's Law.

Note that even if the serial fraction is only 10%, the maximum theoretical speedup achievable is 4.58. In practice, however, hardware architecture characteristics, such as caching, allow programmers to obtain much better performance from multicore systems. Amdahl's law expresses performance increase for constant problem size (*strong scaling*). Gustafson [11] demonstrates that parallel processing can be used to perform more work in the same amount of time by increasing the problem size, thus improving scalability. We call this technique *weak scaling*. Architectural artifacts [13] also play an important role; additional processors come with additional cache and memory resources, often enabling applications to obtain super-linear speedup. We shall discuss a number of optimizations that take advantage of architectural features in Section Optimizations.

5.3 Parallel Programming Paradigms

When discussing parallel programming, practitioners classify the different types of parallelism loosely following machine organizations [8] :

- **Data parallelism (SIMD):** SIMD machines include vector units, array processors, and GPUs. In this model, the program is executing the same code on different data elements. Data parallel algorithms are typically expressed as operations on a multi-dimensional array. Control flow is uniform; however, operations on certain elements may be masked out. Image processing algorithms are prototypical for data parallelism. In the current version of MARE, one can exploit data parallelism by using vector intrinsics [17] to target the NEON units, or by calling OpenGL functions to execute on the GPU. Future versions of MARE will support SIMD compute on the GPU as part of the programming model. In the code example below we show a simple example of scalar vector multiply (SAXPY)

using MARE tasks and vector operations.

```
void saxpy(float* y, float a, float* x, int n) { // Y = a * X
  int i;

  assert(n%4 == 0); // for simplicity we multiply only vector sizes multiple of 4

  auto g = create_group("saxpy");
  float32x4_t av = vmovq_n_f32(a);    // initialize all lanes to a
  for (i = 0; i < n; i+=4) {
    launch(g, [&av, x, y, i] {        // create a task for each vector op
      float32x4_t xv, yv;
      memcpy(&xv.res, x[i], 4*sizeof(float)); // initialize the vector regs
      memcpy(&yv.res, y[i], 4*sizeof(float));
      vmlaq_f32(yv, av, xv);          //     y[i] += a * x[i];
    }
  }
  wait_for(g);                        // wait for all tasks to complete
}
```

- **Task parallelism (MIMD):** MIMD machines are *multiprocessors*. In this model, different hardware execution contexts (e.g., threads, cores, processors) execute different code on different data elements. Tasks are either independent or cooperate on processing over a shared data structure. Thus, tasks may have control or data dependencies. The irregular structure of the computation complicates handling of dependencies and synchronization. Typical applications include: physical simulations, computer simulations, browsers [6], etc. Task parallelism is supported in MARE and examples are provided in the MARE user's manual [18].

- **Braided parallelism:** Modern machines combine CPUs and GPUs for heterogeneous general purpose computation. Recently there has been a significant increase in GPGPU (general purpose GPU) programming. The braided parallelism model combines task parallel computation with data parallel execution [9]. This unified model is used to dynamically exploit data parallelism on SIMD units and GPUs from within concurrent tasks executing on the MIMD units. Examples include gaming applications, which have many concurrent tasks (physics, AI, UI) that are composed from data-parallel computations, such as particle simulations, image processing and rendering.

- **Pipeline parallelism or Streaming:** Distributed processor architectures are composed of separate hardware contexts, such as Cell [15], and designed to exploit small working sets and/or algorithms with little locality. Computation organized as pipeline stages allows code to reside on each unit and the data is streamed across. The pattern of dependencies is fixed, simplyfying the parallel execution. Tuning and balancing is complicated by the predefined computation structure, and may require significant reorganization. Many algorithms are amenable to pipeline parallelism when partitioned. Examples include computer vision algorithms, search, etc.

MARE uses a task based programming model (we plan to support braided parallelism in future iterations). that encourages programmers to think in terms of abstractions that express concurrency as independent units of work. Dependencies are allowed and MARE makes it easy to express these as dynamic task graphs. The MARE programming model is embedded in the C++ language and supported by a runtime library. The entire programming model is implemented as library APIs, therefore can run on any compiler that supports the C++11 standard [5]. This design allows us to take advantage of some of the concurrency abstractions defined in the C++11 standard, such as memory model and synchronization.

To parallelize a computation using MARE, programmers need to address four aspects:

- **Computation:** Computation is expressed as a partially ordered task graph. Each task is an independent unit of work. In MARE tasks are created using the `mare::create_task` method. Dependencies between tasks ensure the proper ordering of computation. In MARE we decouple task

creation from task spawning (or launching) to allow the programmer to set up all the necessary dependencies. Dependencies are set up using the `mare::after` method or the `operator <<`). Once a task is launched (using `mare::launch`) the MARE runtime takes control of the task and it executes it whenever all its dependencies are satisfied and resources are available.

- **Memory:** MARE is supported on shared memory systems. The entire address space is visible to all tasks, following the C++11 memory model. The programmer is responsible for using appropriate synchronization primitives to protect potentially concurrent memory accesses to the same location: C++11 mutexes and locks (e.g. `std::mutex`, `std::unique_lock`), and atomic variables and operations [5], [10].
- **Synchronization:** Besides memory synchronization, MARE also supports task synchronization. Programmers setup dependencies between tasks, which ensure a partial ordering of the computation. In addition, tasks can wait for launched computation to finish using the `mare::wait_for` method, either individually or as a group. Task synchronization can also be implemented using C++11 condition variables (`std::condition_variable`).

5.4 Parallel Programming Patterns

Built upon the basic APIs to create and manage tasks, MARE supports a number of higher level parallel programming patterns. The APIs are described [18], and summarized here.

Parallel iteration is the most common pattern of expressing that the elements of a collection can be processed in parallel. MARE provides two version of this pattern `mare::pfor_each`, which waits at the end of the iteration for all tasks to complete, and `mare::pfor_each_async` which continues execution, while tasks may still be processing elements.

The method `mare::pscan` performs an inplace parallel prefix operation [14] for all elements of a collection. The method `mare::ptransform` performs a map operation on all elements of the collection, returning a new collection.

Future versions of MARE will provide more patterns, however, programmers are encouraged to define their own.

5.5 Optimizations

Beside algorithmic decomposition of work and data, a parallel program requires tuning to a specific platform to achieve optimal performance. As a general rule, the following process should be used:

- **Serial tuning:** The code executed by each task should be optimized using classical optimization techniques: loop optimizations, strength reduction, and cache locality optimizations.
- **Synchronization tuning:** Coordinating parallel execution is typically considered overhead – the program executes additional instructions that are not necessarily part of the effective work. Such overhead includes serialization in critical sections, waiting for dependencies to be satisfied and/or condition variables to be signaled, etc. A well-tuned parallel program spends most of its time executing work as opposed to managing work. However, it may be necessary to replicate computation in order to minimize synchronization.
- **Parallel efficiency:** A parallel execution is optimal when all the execution units are equally busy, doing minimal redundant work. Therefore it is important to balance the computation across all processors. This can be achieved by a combination of algorithmic decomposition — finer grain tasks

allow better load balancing, and taking into account architectural characteristics, such as resource sharing and overhead of spawning tasks — coarser grain tasks typically incur less overhead.

In the next sections we touch briefly on some of these topics.

5.5.1 Cache locality

There are two types of memory reference locality, *temporal locality* in which program references to a given memory address are clustered together in time, and *spatial locality*, where program references to neighboring addresses are clustered in time [12]. Caches transparently take advantage of both types of locality: replacement policies exploit temporal locality, while wide cache lines and prefetching techniques exploit spatial locality. Moreover, current architectures provide several levels of caches, with different sharing patterns. For example, level one caches (L1) are typically split between instructions and data, and are private to a core (shared by the hyperthreads in the case of an SMT architecture), and level two caches (L2) are shared by multiple cores.

Despite programmers not having direct control over caching, code and data can be structured to improves the locality of reference, and thus making effective use of cache mechanisms [20]. In a multicore system caches are a shared resource, so programmers should consider the following:

- **Consistency:** Most multicore shared memory systems provide hardware coherency [12]. However, architectures implement different consistency models [1], thereby affecting the way shared memory updates are visible to different threads. In particular, the ARM architecture defines a weak memory consistency model. The C++11 standard defines primitives to enforce the ordering of memory operations for all atomic accesses. The expert programmer can exploit non-sequential consistent orderings to obtain better performance on such systems.
- **False sharing:** False sharing [21] arises when independent data items used by two tasks executing concurrently on two different cores are co-located in the same cache line. Because the unit of coherence is the cache line, if the items are accessed by both tasks, the line will be forced by the coherence protocol to bounce between caches. False sharing can be avoided by separating data items accessed by different concurrent tasks into separate cache lines, using techniques such as padding [19] and/or allocation to cache line boundaries. To improve locality of reference and limit memory fragmentation, programmers should group data items accessed by a single task as close as possible, preferrably in contiguous blocks of memory addresses.
- **Cache interference:** In serial applications cache optimizations are tuned to the entire cache. However, in a parallel application, caches are shared by execution units. A carefully tuned parallel program should maximize the utilization of the cache, by ensuring reuse of true shared data. For example, by maintaining a single copy of read-only shared data, and referencing it simultaneously, one will exploit temporal locality and minimize the amount of cache used. To minimize contention and interference, the working set sizes of the tasks should fit in the cache. Tiling and cache blocking [7], [22] parameters must be tuned considering the capacity when the caches are shared.

Many other cache locality optimizations are described in the literature.

5.5.2 Minimizing wait time and synchronization

As mentioned, efficient parallel execution implies balanced execution of non-reduntant work on all computing units, while minimizing management overhead. We have shown how the fraction of serial execution (Amdahl's law) limits the efectiveness of the parallel application in Section Parallel Speedups. Therefore, programmers should carefully consider different factors that serialize execution, including:

- **Avoid waiting for single tasks:** Long chains of dependencies, and/or often waiting for the results of single tasks, limits the level of parallelism available in applications [16]. MARE groups can be used to wait for sets of tasks, thus potentially minimizing the overall amount of stalling.
- **Data synchronization:** Synchronizing shared memory accesses may introduce considerable serialization or cache conflict overhead. Such overhead can be reduced by the following optimizations:
  - Privatize data [3] – mutually exclusive partitioning of shared data. For example, partitioning an image into tiles, where each task works on a different tile. In cases where the partitioning is not obvious, programmers can copy shared data into private buffers, work on the private data, and then synchronize changes to the shared copy. Parallel reductions, and parallel gather and scatter operations are helpful in reshaping the private and shared data formats.
  - Avoid large critical sections – Since critical sections guarantee mutual exclusion, they serialize the execution of tasks that are accessing these areas. Minimizing the time spent in critical sections, in particular when they are highly contended will reduce the synchronization overhead.
  - Use atomic operations – the appropriate memory ordering further reduces the synchronization overhead and relies on hardware capabilities for efficient shared data accesses.

MARE encourages an asynchronous programming style, in which fine- grained tasks are placed in a dependence graph, and thus minimizes the need for waits. By contrast, fork-join models spawn a large set of work which needs to complete before the control flows from the join. Asynchronous concurrency is also preferable in the case of heterogeneous computing since resources need not be blocked waiting for an off-load device to complete the work.

5.5.3 Load balancing

Serialization is one of the potential pitfalls of parallel programming. Another is the under-utilization of all compute units. If the parallel computation is unbalanced, some processors will be idle, thereby cutting into the potential performance gains. To avoid such scenarios, programmers should pay attention to balancing the work. This can be achieved in several ways, the most popular being:

- **Tuning the task granularity:** Task granularity represents the amount of work in a task. Ideally, if the amount of work is known, one can balance the computation manually. However, this is not the case for irregular applications, in which case, overdecomposition and relying on the MARE runtime dynamic scheduling is a better option. Task granularity also plays an important role in managing the overhead. As task granularity decreases, the overhead of managing the parallel execution becomes a larger fraction of the total time. Therefore, coarser tasks are preferred to minimize the overhead. This is an important balance that the programmers need to weigh. MARE makes it easy to explore these trade-offs by providing a set of flexible APIs to create tasks.
- **Overdecomposition:** Overdecomposition is the mechanism by which programmers ensure there is enough parallel work in the system, so that the runtime always has work to schedule. Overdecomposition is defined as creating more tasks than the number of computation units available, such that if a task blocks or waits for dependencies to be satisfied, other independent tasks continue to make progress. The more independent tasks are provided, the better the load balancing that can be achieved. Of course, one needs to take into consideration the task granularity and manage the overhead.

5.6 Conclusions

Parallel programming is fun and intellectually challenging. There are many factors that come into play when building a parallel application, which may not be obvious. The techniques described in this tutorial will help you reach the main goal of parallel programming — speeding up the execution of the application. MARE is designed to ease this task and provide abstractions that make it convenient to express parallel computation. The hard work of creating a parallel algorithm remains; however, MARE and these techniques will help encoding these algorithms into an efficient solution.

6 Image Processing Tutorial

6.1 Abstract

The goal of this tutorial is to illustrate how to use the MARE programming model to process images using task parallelism and shared memory.

6.2 Image Processing Filter

As an example, we shall explore the non-local means (NL-means) image denoising algorithm [4]. In this algorithm, the estimated value of a pixel is computed as a weighted average of all pixels in the image. The weights depend on the similarity between pairs of pixels, a similarity which is defined as a decreasing function of the weighted Euclidian distance. Pixels with a similar gray level neighborhood have, on average, larger weights. For a practical computational algorithm, we restrict the search of similar windows in an SxS window. In [4] a 21x21 search window with a 7x7 similarity square neighborhood is considered robust enough to denoise, while taking care of the finer details.

The weights are computed using the following equation:

$$w(i,j) = \frac{1}{Z(i)} e^{-\frac{||v(N_i)-v(N_j)||^2_{2,a}}{h^2}},$$

where $Z(i)$ is the normalizing constant:

$$Z(i) = \sum_j e^{-\frac{||v(N_i)-v(N_j)||^2_{2,a}}{h^2}}$$

Pseudo-code for implementing this algorithm is shown below.

```
#define SEARCH_WINDOW_SIZE     21
#define SIMILARITY_WINDOW_SIZE  7

void compute_weights(Pixel restrict *input, int x, int y, int *weights)
{
  // compute similarity using Euclidian distance in the similarity window,
  // using equation above.
  // reads from the input, writes int the array weights
}

int denoise_image(Pixel restrict *input, int width, int height, Pixel *output)
{
  for(auto pixel : input[0:width][0:height]) { // iterate through all pixels in the input image
      // compute weights for pixels in the search window
      int w[SEARCH_WINDOW_SIZE][SEARCH_WINDOW_SIZE];
      compute_weights(input, pixel.x, pixel.y, w);

      // denoise: compute the weighted average for this pixel
      output[pixel] = input[pixel];
```

```
      for(int i = 0; i < SEARCH_WINDOW_SIZE; i++) {
        for(int j = 0; j < SEARCH_WINDOW_SIZE; j++) {
          Pixel neighbor(pixel.x - SEARCH_WINDOW_SIZE/2 + i,
                         pixel.y - SEARCH_WINDOW_SIZE/2 + j);
          output[pixel] += w[i][j] * input[neighbor];
        }
      }
  }
}
```

6.3 Parallel Image Processing using MARE

The denoising algorithm presented above is embarrasingly parallel. In this implementation, we do not use an in-place algorithm (the output is written in a separate image); therefore, each pixel can be processed in parallel with all other pixels.

6.3.1 Naive Parallelization

Given this algorithm, a naive implementation will simply parallelize the outermost loop in denoise_image, creating a task for each pixel, and launching it asynchronously:

```
int denoise_image(Pixel restrict *input, int width, int height, Pixel *output)
{
  auto g = create_group("denoise");

  for(auto pixel : input[0:width][0:height]) { // iterate through all pixels
    launch(g, [=] {
      // compute weights for pixels in the search window
      int w[SEARCH_WINDOW_SIZE][SEARCH_WINDOW_SIZE];
      compute_weights(input, pixel.x, pixel.y, w);

      // denoise: compute the weighted average for this pixel
      output[pixel] = input[pixel];
      for(int i = 0; i < SEARCH_WINDOW_SIZE; i++) {
        for(int j = 0; j < SEARCH_WINDOW_SIZE; j++) {
          Pixel neighbor(pixel.x - SEARCH_WINDOW_SIZE/2 + i,
                         pixel.y - SEARCH_WINDOW_SIZE/2 + j);
          output[pixel] += w[i][j] * input[neighbor];
        }
      }
    });
  }
  wait_for(g);    // wait for all the tasks to complete
}
```

While such a parallelization strategy is very simple and easy to implement in MARE, the performance of such an implementation may not be optimal, for several reasons, as discussed in the Parallel Processing Tutorial. In particular, this implementation is too fine-grained to overcome the parallel overhead and does not exploit cache locality.

6.3.2 Tiling for Parallelization

A simple method to coarsen the granularity of tasks is to tile the image and spawn tasks for each tile. We can do this either by tiling the loop directly:

```
int denoise_image(Pixel restrict *input, int width, int height, Pixel *output)
{
  auto g = create_group("denoise");

  for(int x = 0; x < width; x += TILE_SIZE) {
    for(int y = 0; y < height; y += TILE_SIZE) {
```

```
      launch(g, [=] {
      for(auto pixel : input[x:TILE_SIZE][y:TILE_SIZE]) { // iterate through pixels in the tile
        // compute weights for pixels in the search window
        int w[SEARCH_WINDOW_SIZE][SEARCH_WINDOW_SIZE];
        compute_weights(input, pixel.x, pixel.y, w);

        // denoise: compute the weighted average for this pixel
        output[pixel] = input[pixel];
        for(int i = 0; i < SEARCH_WINDOW_SIZE; i++) {
          for(int j = 0; j < SEARCH_WINDOW_SIZE; j++) {
            Pixel neighbor(pixel.x - SEARCH_WINDOW_SIZE/2 + i,
                           pixel.y - SEARCH_WINDOW_SIZE/2 + j);
            output[pixel] += w[i][j] * input[neighbor];
          }
        }
      }));
    }
  }
  wait_for(g);    // wait for all the tasks to complete
}
```

or by restructuring the code, such that we preserve a denoise kernel that is identical to the serial implementation and parallelize its invocation.

```
int denoise_kernel(Pixel restrict *input, int x, int width, int y, int height, Pixel *output)
{
   for(auto pixel : input[x:width][y:height]) { // iterate through pixels
        // compute weights for pixels in the search window
        int w[SEARCH_WINDOW_SIZE][SEARCH_WINDOW_SIZE];
        compute_weights(input, pixel.x, pixel.y, w);

        // denoise: compute the weighted average for this pixel
        output[pixel] = input[pixel];
        for(int i = 0; i < SEARCH_WINDOW_SIZE; i++) {
          for(int j = 0; j < SEARCH_WINDOW_SIZE; j++) {
            Pixel neighbor(pixel.x - SEARCH_WINDOW_SIZE/2 + i,
                           pixel.y - SEARCH_WINDOW_SIZE/2 + j);
            output[pixel] += w[i][j] * input[neighbor];
          }
        }
   }
}

void denoise_image()
{
  // initialization, etc
  auto g = create_group("denoise");

  for(int x = 0; x < width; x += TILE_SIZE) {
    for(int y = 0; y < height; y += TILE_SIZE) {

      launch(g, [=] {
        denoise_kernel(input, x, TILE_SIZE, y, TILE_SIZE, output);
      });
    }
  }
  wait_for(g);    // wait for all the tasks to complete

}
```

7 Init and Shutdown Reference API

In this chapter we introduce the MARE initialization and termination API, as well as routines to interoperate with other threading packages.

7.1 Init and shutdown

Functions

- void mare::runtime::init ()
- void mare::runtime::shutdown ()

7.1.1 Function Documentation

7.1.1.1 void mare::runtime::init ( ) [inline]

Starts up MARE's runtime.

Initializes MARE internal data structures, tasks schedulers, and thread pools. There should only be one call to init in the entire program, and it should be called before launching tasks or waiting for tasks or groups.

Definition at line 33 of file runtime.hh.

7.1.1.2 void mare::runtime::shutdown ( )

Shuts down MARE's runtime.

Shuts down the runtime. It returns only when all running tasks have finished. Once the runtime has been shut down, it cannot be restarted.

7.2 Interoperability

Typedefs

- typedef void(∗ mare::runtime::callback_t)()

Functions

- callback_t mare::runtime::set_thread_created_callback (callback_t fptr)
- callback_t mare::runtime::set_thread_destroyed_callback (callback_t fptr)
- callback_t mare::runtime::thread_created_callback ()
- callback_t mare::runtime::thread_destroyed_callback ()

7.2.1 Typedef Documentation

7.2.1.1 typedef void(∗ mare::runtime::callback_t)()

Callback function pointer.

Definition at line 54 of file runtime.hh.

7.2.2 Function Documentation

7.2.2.1 callback_t mare::runtime::set_thread_created_callback (callback_t *fptr*)

Sets a callback function that MARE will invoke whenever it creates an internal runtime thread.

Programmers should invoke this method before init() because init() may create threads. Programmers may reset the thread construction callback by using a nullptr as parameter to this method.

Parameters

fptr	New callback function pointer

Returns

Previous callback function pointer or nullptr if none.

7.2.2.2 callback_t mare::runtime::set_thread_destroyed_callback (callback_t *fptr*)

Sets a callback function that MARE will invoke whenever it destroys an internal runtime thread.

Programmers should invoke this method before init() because init() may destroy threads. Programmers may reset the thread destruction callback by using a nullptr as parameter to this method.

Parameters

fptr	New callback function pointer.

Returns

Previous callback function pointer or nullptr if none.

7.2.2.3 callback_t mare::runtime::thread_created_callback ( )

Returns a pointer to the function that MARE will invoke whenever it creates a new thread.

Returns

Current callback function pointer or nullptr if none.

7.2.2.4 callback_t mare::runtime::thread_destroyed_callback ( )

Returns a pointer to the function that MARE will invoke whenever it destroys a new thread.

Returns

Current callback function pointer or nullptr if none.

8 Patterns Reference API

The MARE parallel patterns API provides programmers with a high-level interface to express commonly used parallel programming idioms, such as parallel loops, parallel prefix operations, parallel map and reduce operations, etc. See GPU Patterns for patterns which can run on the GPU.

8.1 Patterns

Using the patterns defined in this chapter requires including the following header file:

```
#include <mare/patterns.hh>
```

Functions

- template<class InputIterator , typename UnaryFn >
  void mare::pfor_each_async (group_ptr g, InputIterator first, InputIterator last, UnaryFn fn)
- template<size_t DIMS, typename UnaryFn >
  void mare::pfor_each_async (group_ptr g, const mare::range< DIMS > &r, UnaryFn fn)
- template<class InputIterator , typename UnaryFn >
  void mare::pfor_each (group_ptr group, InputIterator first, InputIterator last, UnaryFn &&fn)
- template<class InputIterator , typename UnaryFn >
  void mare::pfor_each (InputIterator first, InputIterator last, UnaryFn &&fn)
- template<size_t DIMS, typename UnaryFn >
  void mare::pfor_each (group_ptr group, const mare::range< DIMS > &r, UnaryFn &&fn)
- template<size_t DIMS, typename UnaryFn >
  void mare::pfor_each (const mare::range< DIMS > &r, UnaryFn &&fn)
- template<typename InputIterator , typename OutputIterator , typename UnaryFn >
  void mare::ptransform (group_ptr group, InputIterator first, InputIterator last, OutputIterator d_first, UnaryFn fn)
- template<typename InputIterator , typename OutputIterator , typename UnaryFn >
  void mare::ptransform (InputIterator first, InputIterator last, OutputIterator d_first, UnaryFn &&fn)
- template<typename InputIterator , typename OutputIterator , typename BinaryFn >
  void mare::ptransform (group_ptr group, InputIterator first1, InputIterator last1, InputIterator first2, OutputIterator d_first, BinaryFn fn)
- template<typename InputIterator , typename OutputIterator , typename BinaryFn >
  void mare::ptransform (InputIterator first1, InputIterator last1, InputIterator first2, OutputIterator d_first, BinaryFn &&fn)
- template<typename InputIterator , typename UnaryFn >
  void mare::ptransform (group_ptr group, InputIterator first, InputIterator last, UnaryFn fn)
- template<typename InputIterator , typename UnaryFn >
  void mare::ptransform (InputIterator first, InputIterator last, UnaryFn &&fn)
- template<typename InputIterator , typename BinaryFn >
  void mare::pscan_inclusive (group_ptr group, InputIterator first, InputIterator last, BinaryFn fn)
- template<typename InputIterator , typename BinaryFn >
  void mare::pscan_inclusive (InputIterator first, InputIterator last, BinaryFn &&fn)

8.1.1 Function Documentation

8.1.1.1 template<class InputIterator , typename UnaryFn > void mare::pfor_each_async (group_ptr *g,* InputIterator *first,* InputIterator *last,* UnaryFn *fn*)

Parallel version of `std::for_each` (asynchronous).

Applies function object `fn` in parallel to every iterator in the range [first, last).

Note: In contrast to `std::for_each` and `ptransform`, the iterator is passed to the function, instead of the element.

It is permissible to modify the elements of the range from `fn`, assuming that `InputIterator` is a mutable iterator.

Note: This function does NOT wait for all function applications to finish. Callers must wait on `g` (see `wait_for`), if this is desired.

Complexity

Exactly `last-first` applications of `fn`.

See Also

ptransform(group_ptr, InputIterator, InputIterator, UnaryFn)
pfor_each(group_ptr, InputIterator, InputIterator, UnaryFn&&)

Parameters

g	All MARE tasks created are added to this group.
first	Start of the range to which to apply `fn`.
last	End of the range to which to apply `fn`.
fn	Unary function object to be applied.

Definition at line 139 of file patterns.hh.

8.1.1.2 template<size_t DIMS, typename UnaryFn > void mare::pfor_each_async (group_ptr *g,* const mare::range< DIMS > & *r,* UnaryFn *fn*)

Parallel version of `std::for_each` (asynchronous).

See Also

pfor_each(mare::range<DIMS>&, body_with_attrs_gpu< Body, KernelPtr, Kargs...>&&)
pfor_each_async(group_ptr, InputIterator, InputIterator, UnaryFn)

Parameters

g	All MARE tasks created are added to this group.
r	Range object (1D, 2D or 3D) representing the iteration space.

fn	the unary function object to be applied

Definition at line 184 of file patterns.hh.

8.1.1.3 template<class InputIterator , typename UnaryFn > void mare::pfor_each (group_ptr *group,* InputIterator *first,* InputIterator *last,* UnaryFn && *fn*)

Parallel version of `std::for_each`.

Applies function object `fn` in parallel to every iterator in the range [first, last).

Note: In contrast to `std::for_each` and ptransform, the iterator is passed to the function, instead of the element.

It is permissible to modify the elements of the range from `fn`, provided that `InputIterator` is a mutable iterator.

Note: This function returns only after `fn` has been applied to the whole iteration range.

In addition, the call to this function can be canceled by canceling the group passed as argument. However, in the presence of cancelation it is undefined to which extent the iteration space will have been processed.

Note: The usual rules for cancelation apply, i.e., within `fn` the cancelation must be acknowledged using `abort_on_cancel`.

Complexity

Exactly `std::distance(first,last)` applications of `fn`.

See Also

ptransform(group_ptr, InputIterator, InputIterator, UnaryFn)
pfor_each_async(group_ptr, InputIterator, InputIterator, UnaryFn)
abort_on_cancel()

Examples

```
group_ptr g = create_group("g");
[...]
// Parallel for-loop using indices
pfor_each(g, size_t(0), vin.size(),
          [=,&vin,&vout] (size_t i) {
              while (!finished_lengthy_computation()) {
                 abort_on_cancel();
                 process(i);
          });
[...]
// elsewhere:
cancel(g);
```

Parameters

group	All MARE tasks created are added to this group.
first	Start of the range to which to apply `fn`.

last	End of the range to which to apply `fn`.
fn	Unary function object to be applied.

Definition at line 463 of file patterns.hh.

8.1.1.4 template<class InputIterator , typename UnaryFn > void mare::pfor_each (InputIterator *first,* InputIterator *last,* UnaryFn && *fn*)

Parallel version of std::for_each.

See Also

pfor_each(group_ptr, InputIterator, InputIterator, UnaryFn&&)
ptransform(group_ptr, InputIterator, InputIterator, UnaryFn)
pfor_each_async(group_ptr, InputIterator, InputIterator, UnaryFn)

Examples

```
// Parallel for-loop using indices
vector<size_t> vout(vin.size());
// vout[i] := 2*vin[i]
pfor_each(size_t(0), vin.size(),
          [=,&vin,&vout] (size_t i) {
              vout[i] = 2*vin[i];
          });
```

Parameters

first	start of the range to which to apply 'fn'
last	end of the range to which to apply 'fn'
fn	the unary function object to be applied

Definition at line 496 of file patterns.hh.

8.1.1.5 template<size_t DIMS, typename UnaryFn > void mare::pfor_each (group_ptr *group,* const mare::range< DIMS > & *r,* UnaryFn && *fn*)

Parallel version of `std::for_each`

See Also

pfor_each(mare::range<DIMS>&, body_with_attrs_gpu< Body, KernelPtr, Kargs...>&&)
pfor_each(group_ptr, InputIterator, InputIterator, UnaryFn&&)

Parameters

g	All MARE tasks created are added to this group.
r	Range object (1D, 2D or 3D) representing the iteration space.
fn	the unary function object to be applied

Definition at line 517 of file patterns.hh.

8.1.1.6 template<size_t DIMS, typename UnaryFn > void mare::pfor_each (const mare::range< DIMS > & *r,* UnaryFn && *fn*)

Parallel version of `std::for_each`.

See Also

pfor_each(mare::range<DIMS>&, body_with_attrs_gpu< Body, KernelPtr, Kargs...>&&)
pfor_each(InputIterator, InputIterator, UnaryFn)

Parameters

r	Range object (1D, 2D or 3D) representing the iteration space.
fn	the unary function object to be applied

Definition at line 537 of file patterns.hh.

8.1.1.7 template<typename InputIterator , typename OutputIterator , typename UnaryFn > void mare::ptransform (group_ptr *group,* InputIterator *first,* InputIterator *last,* OutputIterator *d_first,* UnaryFn *fn*)

Parallel version of `std::transform`.

Applies function object `fn` in parallel to every dereferenced iterator in the range [first, last) and stores the return value in another range, starting at `d_first`.

Note: This function returns only after `fn` has been applied to the whole iteration range.

In addition, the call to this function can be canceled by canceling the group passed as argument. However, in the presence of cancelation it is undefined to which extent the iteration space will have been processed.

Note: The usual rules for cancelation apply, i.e., within `fn` the cancelation must be acknowledged using `abort_on_cancel`.

Complexity

Exactly `std::distance(first,last)` applications of `fn`.

See Also

pfor_each

Examples

```
// arr[i] == 2*vin[N-i]
size_t arr[vin.size()];
ptransform(group, begin(vin), end(vin), arr,
           [=] (size_t const& i) {
             return 2*i;
           });
```

Parameters

group	All MARE tasks created are added to this group.
first	Start of the range to which to apply `fn`.
last	End of the range to which to apply `fn`.
d_first	Start of the destination range.
fn	Unary function object to be applied.

Definition at line 587 of file patterns.hh.

8.1.1.8 template<typename InputIterator , typename OutputIterator , typename UnaryFn > void mare::ptransform (InputIterator *first,* InputIterator *last,* OutputIterator *d_first,* UnaryFn && *fn*)

Parallel version of `std::transform`.

See Also

ptransform(group_ptr, InputIterator, InputIterator, UnaryFn)
pfor_each(group_ptr, InputIterator, InputIterator, UnaryFn&&)

Examples

```
// arr[i] == 2*vin[N-i]
size_t arr[vin.size()];
ptransform(begin(vin), end(vin), arr,
           [=] (size_t const& i) {
             return 2*i;
           });
```

Parameters

first	Start of the range to which to apply `fn`.
last	End of the range to which to apply `fn`.
d_first	Start of the destination range.
fn	Unary function object to be applied.

Definition at line 643 of file patterns.hh.

8.1.1.9 template<typename InputIterator , typename OutputIterator , typename BinaryFn > void mare::ptransform (group_ptr *group,* InputIterator *first1,* InputIterator *last1,* InputIterator *first2,* OutputIterator *d_first,* BinaryFn *fn*)

Parallel version of `std::transform`.

Applies function object `fn` in parallel to every pair of dereferenced destination iterators in the range [first1, last1) and [first2,...), and stores the return value in another range, starting at `d_first`.

Note: This function returns only after `fn` has been applied to the whole iteration range.

In addition, the call to this function can be canceled by canceling the group passed as argument. However, in the presence of cancelation it is undefined to which extent the iteration space will have been processed.

Note: The usual rules for cancelation apply, i.e., within `fn` the cancelation must be acknowledged using `abort_on_cancel`.

Complexity

Exactly `std::distance(first1,last1)` applications of `fn`.

See Also

pfor_each(group_ptr, InputIterator, InputIterator, UnaryFn&&)

Examples

```
// vout[i] == vin[i] + vin[i+1]
vector<size_t> vout(vin.size()-1);
ptransform(group,
           begin(vin), begin(vin)+vout.size(),
           begin(vin)+1,
           begin(vout),
           [=] (size_t const& i, size_t const& j) {
             return i+j;
           });
```

Parameters

group	All MARE tasks created are added to this group.
first1	Start of the range to which to apply `fn`.
last1	End of the range to which to apply `fn`.
first2	Start of the second range to which to apply `fn`.
d_first	Start of the destination range.
fn	Binary function object to be applied.

Definition at line 699 of file patterns.hh.

8.1.1.10 template<typename InputIterator , typename OutputIterator , typename BinaryFn > void mare::ptransform (InputIterator *first1,* InputIterator *last1,* InputIterator *first2,* OutputIterator *d_first,* BinaryFn && *fn*)

Parallel version of `std::transform`.

Applies function object `fn` in parallel to every pair of dereferenced destination iterators in the range [first1, last1) and [first2,...), and stores the return value in another range, starting at `d_first`.

Note: This function returns only after `fn` has been applied to the whole iteration range.

Complexity

Exactly `std::distance(first1,last1)` applications of `fn`.

See Also

ptransform(group_ptr, InputIterator, InputIterator, UnaryFn)
pfor_each(group_ptr, InputIterator, InputIterator, UnaryFn&&)

Examples

```
// vout[i] == vin[i] + vin[i+1]
vector<size_t> vout(vin.size()-1);
ptransform(begin(vin), begin(vin)+vout.size(),
           begin(vin)+1,
           begin(vout),
           [=] (size_t const& i, size_t const& j) {
             return i+j;
           });
```

Parameters

first1	Start of the range to which to apply `fn`.
last1	End of the range to which to apply `fn`.
first2	Start of the second range to which to apply `fn`.
d_first	Start of the destination range.
fn	Binary function object to be applied.

Definition at line 772 of file patterns.hh.

8.1.1.11 template<typename InputIterator , typename UnaryFn > void mare::ptransform (group_ptr *group,* InputIterator *first,* InputIterator *last,* UnaryFn *fn*)

Parallel version of `std::transform`.

Applies function object `fn` in parallel to every dereferenced iterator in the range [first, last).

Note: In contrast to pfor_each, the dereferenced iterator is passed to the function.

It is permissible to modify the elements of the range from `fn`, assuming that `InputIterator` is a mutable iterator.

Note: This function returns only after `fn` has been applied to the whole iteration range.

In addition, the call to this function can be canceled by canceling the group passed as argument. However, in the presence of cancelation it is undefined to which extent the iteration space will have been processed.

Note: The usual rules for cancelation apply, i.e., within `fn` the cancelation must be acknowledged using `abort_on_cancel`.

Complexity

Exactly `std::distance(first,last)` applications of `fn`.

See Also

pfor_each(group_ptr, InputIterator, InputIterator, UnaryFn&&)

Examples

```
// Parallel for-loop using indices
vector<size_t> vout(vin.size());
// vout[i] := 2*vin[i]
pfor_each(group,
          size_t(0), vin.size(),
          [=,&vin,&vout] (size_t i) {
              vout[i] = 2*vin[i];
          });

// Parallel for-loop using iterators
vector<size_t> vout(vin.size());
// vout[i] := 2*vin[i]
pfor_each(group,
          begin(vin), end(vin),
          [&vin,&vout] (vector<size_t>::const_iterator it) {
              vout[it - begin(vin)] = 2 * *it;
          });
```

Parameters

group	All MARE tasks created are added to this group.
first	Start of the range to which to apply `fn`.
last	End of the range to which to apply `fn`.
fn	Unary function object to be applied.

Definition at line 839 of file patterns.hh.

8.1.1.12 template<typename InputIterator , typename UnaryFn > void mare::ptransform (InputIterator *first*, InputIterator *last*, UnaryFn && *fn*)

Parallel version of `std::transform`.

See Also

ptransform(group_ptr, InputIterator, InputIterator, UnaryFn)
pfor_each(group_ptr, InputIterator, InputIterator, UnaryFn&&)

Examples

```
// Parallel for-loop using indices
vector<size_t> vout(vin.size());
// vout[i] := 2*vin[i]
pfor_each(size_t(0), vin.size(),
          [=,&vin,&vout] (size_t i) {
              vout[i] = 2*vin[i];
          });

// Parallel for-loop using iterators
vector<size_t> vout(vin.size());
// vout[i] := 2*vin[i]
pfor_each(begin(vin), end(vin),
          [&vin,&vout] (vector<size_t>::const_iterator it) {
              vout[it - begin(vin)] = 2 * *it;
```

```
});
```

Parameters

first	Start of the range to which to apply `fn`.
last	End of the range to which to apply `fn`.
fn	Unary function object to be applied.

Definition at line 877 of file patterns.hh.

8.1.1.13 template<typename InputIterator , typename BinaryFn > void mare::pscan_inclusive (group_ptr *group,* InputIterator *first,* InputIterator *last,* BinaryFn *fn*)

Sklansky-style parallel inclusive scan.

Performs an in-place parallel prefix computation using the function object `fn` for the range [first, last).

It is not permissible for `fn` to modify the elements of the range. Also, `fn` should be associative, because the order of applications is not fixed.

Note: This function returns only after `fn` has been applied to the whole iteration range.

In addition, the call to this function can be canceled by canceling the group passed as argument. However, in the presence of cancelation it is undefined to which extent the iteration space will have been processed.

Note: The usual rules for cancelation apply, i.e., within `fn` the cancelation must be acknowledged using `abort_on_cancel`.

Examples

```
// After: v' = { v[0], v[0] x v[1], v[0] x v[1] x v[2], ... }
pscan_inclusive(group,
                begin(v), end(v),
                [] (size_t const& i, size_t const& j) {
                    return i + j;
                });
```

Parameters

group	All MARE tasks created are added to this group.
first	Start of the range to which to apply `fn`.
last	End of the range to which to apply `fn`.
fn	Binary function object to be applied.

Definition at line 949 of file patterns.hh.

8.1.1.14 template<typename InputIterator , typename BinaryFn > void mare::pscan_inclusive (InputIterator *first,* InputIterator *last,* BinaryFn && *fn*)

Sklansky-style parallel inclusive scan.

See Also

pscan_inclusive(group_ptr, InputIterator, InputIterator, BinaryFn)

Examples

```
// After: v' = { v[0], v[0] x v[1], v[0] x v[1] x v[2], ... }
pscan_inclusive(begin(v), end(v),
                [] (size_t const& i, size_t const& j) {
                    return i + j;
                });
```

Parameters

first	Start of the range to which to apply `fn`.
last	End of the range to which to apply `fn`.
fn	Binary function object to be applied.

Definition at line 979 of file patterns.hh.

8.2 Synchronous Dataflow

Using the patterns defined in this chapter requires including the following header file:

```
#include <mare/sdf.hh>
```

Classes

- class mare::channel
- class mare::data_channel< T >
- class mare::sdf_graph_query_info
- class mare::node_channels

Typedefs

- typedef std::tuple
  < internal::direction, channel * > mare::tuple_dir_channel

Enumerations

- enum mare::sdf_interrupt_type { **undef**, **iter_non_synced**, **iter_synced** }

Functions

- template<typename T , typename Trange >
  void mare::preload_channel (data_channel< T > &dc, Trange const &tr)
- sdf_graph_ptr mare::create_sdf_graph ()
- void mare::destroy_sdf_graph (sdf_graph_ptr &g)
- template<typename... Ts>
  io_channels< Ts...> mare::with_inputs (data_channel< Ts > &...dcs)
- template<typename... Ts>
  io_channels< Ts...> mare::with_outputs (data_channel< Ts > &...dcs)
- template<typename Body , typename... IOC_GROUPS>
  std::enable_if
  < internal::is_MDCAW
  < IOC_GROUPS...>::value,
  sdf_node_ptr >::type mare::create_sdf_node (sdf_graph_ptr g, Body &&body, IOC_GROUPS const &...io_channels_groups)
- void mare::assign_cost (sdf_node_ptr n, double execution_cost)
- sdf_graph_ptr mare::get_graph_ptr (sdf_node_ptr node)
- std::size_t mare::get_debug_id (sdf_node_ptr node)
- void mare::launch_and_wait (sdf_graph_ptr g)
- void mare::launch_and_wait (sdf_graph_ptr g, std::size_t num_iterations)
- void mare::launch (sdf_graph_ptr g)

- void mare::launch (sdf_graph_ptr g, std::size_t num_iterations)
- void mare::wait_for (sdf_graph_ptr g)
- void mare::cancel (sdf_graph_ptr g, sdf_interrupt_type intr_type=sdf_interrupt_type::iter_non_synced)
- void mare::cancel (sdf_graph_ptr g, std::size_t desired_cancel_iteration, sdf_interrupt_type intr_type=sdf_interrupt_type::iter_non_synced)
- bool mare::pause (sdf_graph_ptr g, sdf_interrupt_type intr_type=sdf_interrupt_type::iter_non_synced)
- bool mare::pause (sdf_graph_ptr g, std::size_t desired_pause_iteration, sdf_interrupt_type intr_type=sdf_interrupt_type::iter_non_synced)
- void mare::resume (sdf_graph_ptr g)
- sdf_graph_query_info mare::sdf_graph_query (sdf_graph_ptr g)
- std::string mare::to_string (sdf_graph_query_info const &info)
- template<typename OStream >
  OStream & mare::operator<< (OStream &os, sdf_graph_query_info const &info)
- template<typename T >
  tuple_dir_channel mare::as_in_channel_tuple (data_channel< T > &dc)
- template<typename T >
  tuple_dir_channel mare::as_out_channel_tuple (data_channel< T > &dc)
- sdf_node_ptr mare::create_sdf_node (sdf_graph_ptr g, void(*body)(node_channels &), std::vector< tuple_dir_channel > &v_dir_channels)
- template<typename Body >
  sdf_node_ptr mare::create_sdf_node (sdf_graph_ptr g, Body &&body, std::vector< tuple_dir_channel > &v_dir_channels)

8.2.1 Class Documentation

8.2.1.1 class mare::channel

Base class representing a channel connection between two SDF nodes.

Cannot be instantiated directly by user. User code must instantiate the derived class `data_channel<T>` to create a channel carrying elements of user-defined-type `T`.

However, user code can pass pointers of the base type `channel*` and use those to connect SDF nodes instead of the templated `data_channel<T>`. This is useful for the programmatic construction of an SDF graph, where the number and user-data-types of channels connected to a node may not be fixed at compile-time: specialized user code may create typed data_channels, while generic user code may only use `channel*` to connect the graph.

See Also

data_channel

Definition at line 40 of file channel.hh.

Public member functions

- std::size_t get_elem_size () const

Protected Member Functions

- **channel** (std::size_t Tsize)

Private member functions

- **MARE_DELETE_METHOD** (channel(channel const &))
- **MARE_DELETE_METHOD** (channel(channel &&))
- **MARE_DELETE_METHOD** (channel &operator=(channel const &))
- **MARE_DELETE_METHOD** (channel &operator=(channel &&))

Private Attributes

- internal::channel_internal_record * **_cir**

Friends

- class **internal::channel_accessor**

8.2.1.1.1 Member Function Documentation

8.2.1.1.1.1 std::size_t mare::channel::get_elem_size ( ) const

Returns `sizeof(T)` corresponding to the user-data-type `T` used to construct the channel `data_channel<T>`.

Provides some introspection or debugging capability to generic user-code when the application is divided into specialized user-code that creates `data_channel<T>`, but only `channel*` is passed to the generic user code.

Returns

`sizeof(T)` for `T` used in channel construction via `data_channel<T>`

8.2.1.2 class mare::data_channel

template<typename T>class mare::data_channel< T >

Construct a channel carrying elements of arbitrary user-data-type `T`.

A channel must be connected as input to exactly one SDF node and as output to exactly one SDF node (possibly the same, if "delays" are added on the channel), before the launch of a graph containing the nodes. Both nodes must belong to the same SDF graph.

See Also

channel the base class.
preload_channel() for adding delays on a channel.

Definition at line 83 of file channel.hh.

Public member functions

- **MARE_DELETE_METHOD** (data_channel(data_channel< T > const &))
- **MARE_DELETE_METHOD** (data_channel(data_channel< T > &&))
- **MARE_DELETE_METHOD** (data_channel &operator=(data_channel< T > const &))
- **MARE_DELETE_METHOD** (data_channel &operator=(data_channel< T > &&))

Additional Inherited Members

8.2.1.3 class mare::sdf_graph_query_info

Query information about the state of an SDF graph.

Captures a sample of an SDF graph's execution state when sdf_graph_query() is used to query a graph's execution state.

See Also

sdf_graph_query()
to_string()

Definition at line 490 of file sdf.hh.

Public member functions

- bool was_launched () const
- bool has_completed () const
- bool is_paused () const
- std::size_t current_min_iteration () const
- std::size_t current_max_iteration () const
- sdf_interrupt_type intr_type () const

Private Attributes

- bool **_was_launched**
- bool **_has_completed**
- bool **_is_paused**
- std::size_t **_current_min_iteration**
- std::size_t **_current_max_iteration**
- sdf_interrupt_type **_intr_type**

Friends

- sdf_graph_query_info sdf_graph_query (sdf_graph_ptr g)
- std::string to_string (sdf_graph_query_info const &info)

8.2.1.3.1 Member Function Documentation

8.2.1.3.1.1 bool mare::sdf_graph_query_info::was_launched ( ) const [inline]

Indicates if graph has been launched and started execution.

Returns

TRUE if the graph has already been launched and has started execution.
FALSE otherwise.

Definition at line 499 of file sdf.hh.

8.2.1.3.1.2 bool mare::sdf_graph_query_info::has_completed ( ) const [inline]

Indicates if the graph has completed execution or been cancelled.

Returns

TRUE if the graph has completed execution or been cancelled.
FALSE otherwise.

Definition at line 508 of file sdf.hh.

8.2.1.3.1.3 bool mare::sdf_graph_query_info::is_paused ( ) const [inline]

Indicates if the graph was launched and has been paused.

Returns

TRUE if the graph was launched and has been paused.
FALSE otherwise.

Definition at line 517 of file sdf.hh.

8.2.1.3.1.4 std::size_t mare::sdf_graph_query_info::current_min_iteration ( ) const [inline]

Provides the minimum graph iteration started by all graph nodes.

- Defined iff `has_completed() == true` or `is_paused() == true`.
- On `iter_non_synced` exit or pause, current_min_iteration() and current_max_iteration() together bound the range of iterations completed by all graph nodes.
- On `iter_synced` exit or pause, both are equal. In contrast, both being equal does not imply `iter_synced`, since some nodes may have been interrupted in the middle of executing a node iteration.

Returns

The minimum graph iteration that every node in the graph has started or exceeded.

See Also

current_max_iteration()

Definition at line 538 of file sdf.hh.

8.2.1.3.1.5 std::size_t mare::sdf_graph_query_info::current_max_iteration ( ) const [inline]

Provides the maximum graph iteration completed by any graph node.

- Defined iff `has_completed() == true` or `is_paused() == true`.
- On `iter_non_synced` exit or pause, current_min_iteration() and current_max_iteration() together bound the range of iterations completed by all graph nodes.
- On `iter_synced` exit or pause, both are equal. In contrast, both being equal does not imply `iter_synced`, since some nodes may have been interrupted in the middle of executing a node iteration.

Returns

The maximum graph iteration that a node anywhere in the graph has completed.

See Also

current_min_iteration()

Definition at line 564 of file sdf.hh.

8.2.1.3.1.6 sdf_interrupt_type mare::sdf_graph_query_info::intr_type ( ) const [inline]

Interruption type observed after the last cancel or pause request (if any) to take effect.

Returns

The interruption type in effect when the query was made.

`undef`, if the graph was not paused, cancelled or completed when the query was made.

See Also

sdf_interrupt_type

Definition at line 582 of file sdf.hh.

8.2.1.3.2 Related Function Documentation

8.2.1.3.2.1 sdf_graph_query_info sdf_graph_query (sdf_graph_ptr *g*) [friend]

Queries the state of an SDF graph.

Parameters

g	Handle to graph to query.

Returns

Query information that samples the current execution state of the graph.

See Also

sdf_graph_query_info for a description of the graph query information.

8.2.1.3.2.2 std::string to_string (sdf_graph_query_info const & *info*) [friend]

Converts the graph query information to a string suitable for printing.

Parameters

info	The graph query information.

Returns

Query information formatted into a string suitable for printing.

8.2.1.4 class mare::node_channels

Introspection data-structure used by the programmatic body of an SDF node.

In addition to using `body(T1& t1, T2& t2, ..., Tn& tn)`, where the type signature is fixed at compile-time and a priori fixes the data-types, direction and number of channels connecting to that body's SDF node, MARE SDF allows a programmatic introspection body with the following type signature that can discover the nature of channels connected to it:

```
body(node_channels& ncs)
```

Inside the body, the user-code can use the `ncs` object to query the number of channels connected, and for each channel query its direction (in or out) and the element-size carried by it. While node_channels provides the user the flexibility to attach however many channels of whatever user-data-types to a node, type-safety is reduced as there is no longer compile-time type matching between the type signature of `body` and the types and number of channels attached to the node.

read() and write() methods inside node_channels allow the user-code to access data popped from in-channels and provide data to push to out-channels. The user-code must pass program variables by reference, whose sizes must match the element-sizes of the corresponding channels. read() and write() provide some runtime safety by checking that the size of the passed program variable matches the element-size of the channel.

Runtime error if a write is attempted on an in-channel or if a read is attempted on an out-channel.

Definition at line 48 of file sdfpr.hh.

Public member functions

- std::size_t get_num_channels () const
- bool is_in_channel (std::size_t channel_index) const
- bool is_out_channel (std::size_t channel_index) const
- std::size_t get_elemsize (std::size_t channel_index) const
- template<typename T >
  void read (T &t, std::size_t channel_index)
- template<typename T >
  void write (T const &t, std::size_t channel_index)

Private Attributes

- std::vector< internal::direction > **_vdir**
- std::vector< std::size_t > **_velemsize**
- std::vector< char * > **_vbuffer**

Friends

- class **internal::node_channels_accessor**

8.2.1.4.1 Member Function Documentation

8.2.1.4.1.1 std::size_t mare::node_channels::get_num_channels ( ) const

Number of channels connected to the node. Channels are accessed by their indices: 0 to get_num_channels()-1

Returns

Number of channels connected the node.

8.2.1.4.1.2 bool mare::node_channels::is_in_channel (std::size_t *channel_index*) const [inline]

Determines if the specified channel is an in-channel.

Parameters

channel_index	Index to a connected channel.

Returns

TRUE if in-channel to the node. FALSE otherwise.

8.2.1.4.1.3 bool mare::node_channels::is_out_channel (std::size_t *channel_index*) const [inline]

Determines if the specified channel is an out-channel.

Parameters

channel_index	Index to a connected channel.

Returns

TRUE if out-channel to the node. FALSE otherwise.

8.2.1.4.1.4 std::size_t mare::node_channels::get_elemsize (std::size_t *channel_index*) const

Retrieves element-size carried by the specified channel.

Parameters

channel_index	Index to a connected channel.

Returns

Element-size carried by the specified channel.

8.2.1.4.1.5 template<typename T > void mare::node_channels::read (T & *t*, std::size_t *channel_index*)

Copies a popped value from the specified channel into a program variable of data-type `T`.

Multiple reads on a channel are allowed, each returning the same popped value (since the channel pop occurred prior to the invocation of the node's body).

Runtime error if `sizeof(T) != get_elemsize(channel_index)`.

Runtime error if `!is_in_channel(channel_index)`.

Parameters

t	Reference to a program variable to which the popped value will be copied.
channel_index	Index to a connected channel.

8.2.1.4.1.6 template<typename T > void mare::node_channels::write (T const & *t*, std::size_t *channel_index*)

Saves the value of a program variable of data-type `T`. The saved value will subsequently be pushed to the specified channel.

Multiple writes on a channel are allowed, with only the final write providing the value pushed (channel push will occur after completion of the node's body).

Runtime error if `sizeof(T) != get_elemsize(channel_index)`.

Runtime error if `!is_out_channel(channel_index)`.

Parameters

t	Reference to a program variable whose value will be saved for pushing to the channel.
channel_index	Index to a connected channel.

8.2.2 Typedef Documentation

8.2.2.1 typedef std::tuple<internal::direction, channel*> mare::tuple_dir_channel

A tuple holding a channel's binding to an in or out direction.

Tuple is not templated on the user-data-type of the channel data. Suitable for programmatic construction of graph.

See Also

as_in_channel_tuple() and as_out_channel_tuple() produce bindings of type tuple_dir_channel.

Definition at line 153 of file sdfpr.hh.

8.2.3 Enumeration Type Documentation

8.2.3.1 enum mare::sdf_interrupt_type [strong]

Types of graph interruption. Used both for qualifying interruption requests and for characterizing the outcome of an interruption.

- `undef`:
  - Interruption type is not set
- `iter_non_synced`:
  - At interruption, all nodes in the graph need not have executed the same number of graph iterations.
- `iter_synced`:
  - At interruption, all nodes in the graph must have completed exactly the same number of graph iterations.

Note that `iter_synced` is a special case of `iter_non_synced`. It is possible for a pause() or cancel() to request interruption with `iter_non_synced` semantics, and the graph to interrupt with `iter_synced` semantics.

See Also

pause() and cancel() make use of sdf_interrupt_type to specify the interruption semantics when requesting an interruption of the graph execution.
sdf_graph_query_info where a sdf_interrupt_type field provides status information on the graph execution state after an interruption or completion of the graph.

Definition at line 299 of file sdf.hh.

8.2.4 Function Documentation

8.2.4.1 template<typename T , typename Trange > void mare::preload_channel (data_channel< T > & *dc,* Trange const & *tr*)

Adds delay initializer elements to a channel.

Edges in a synchronous dataflow graph can have delays associated with them. Any cycle in the graph must have at least one delay on at least one of the edges constituting the cycle in order to be a valid synchronous dataflow graph. In general, additional delays in a cycle or delays on purely feed-forward paths alter the semantics of the graph.

With MARE SDF, edges are implemented as channels. A delay of 'N' is associated with a channel by adding N initializer elements to the channel using preload_channel().

Parameters

dc	A `data_channel<T>` of user-defined-type `T`.
tr	A container of N elements of data-type `T`, where N is the desired number of delays to add to the channel. N must be > 0. The container can be any data-structure supporting range-based iteration.

8.2.4.2 sdf_graph_ptr mare::create_sdf_graph ( )

Creates an empty graph.

Nodes must subsequently be created with create_sdf_node() calls prior to launching graph.

Returns

Handle to the created graph.

8.2.4.3 void mare::destroy_sdf_graph (sdf_graph_ptr & *g*)

Destroys a graph.

Invoke on graph g that has either not been launched, has completed execution, or has been cancelled. Undefined behavior if an executing graph is destroyed.

Parameters

g	Handle to the graph to be destroyed.

8.2.4.4 template<typename... Ts> io_channels<Ts...> mare::with_inputs (data_channel< Ts > &... *dcs*)

Indicates that a list of data-channels will be inputs.

Parameters

dcs	A variadic list of data-channels `data_channel<T1>` to `data_channels<Tn>` of user-defined-types `T1` to `Tn`.

Returns

An object binding the data-channels to the input direction. Pass to create_sdf_node() to indicate that the created node will treat these channels as input.

Due to some limitations in Visual Studio 2013 support for C++11, only upto 4 channels are supported in each with_inputs() call when compiling with VS 2013 (no limitation when compiling with gcc or clang).

8.2.4.5 template<typename... Ts> io_channels<Ts...> mare::with_outputs (data_channel< Ts > &... *dcs*)

Indicates that a list of data-channels will be outputs.

Parameters

dcs	A variadic list of data-channels `data_channel<T1>` to `data_channels<Tn>` of user-defined-types `T1` to `Tn`.

Returns

An object binding the data-channels to the output direction. Pass to create_sdf_node() to indicate that the created node will treat these channels as output.

Due to some limitations in Visual Studio 2013 support for C++11, only upto 4 channels are supported in each with_outputs() call when compiling with VS 2013 (no limitation when compiling with gcc or clang).

8.2.4.6 template<typename Body , typename... IOC_GROUPS> std::enable_-if<internal::is_MDCAW<IOC_GROUPS...>::value, sdf_node_ptr>::type mare::create_sdf_node (sdf_graph_ptr *g,* Body && *body,* IOC_GROUPS const &... *io_channels_groups*)

Creates a new node within the specified SDF graph.

The node needs a user-defined `body` and a sequence of input/output data-channels to connect to.

The `body` is a function or a callable object supporting invocation as follows:

```
body(T1& t1, T2& t2, T3& t3, ..., Tn& tn)
```

where `T1` to `Tn` are the user-data-types of `data_channel<T1>` to `data_channel<T2>` connected to the node in the same order.

Data-channels are connected as inputs or outputs to the node when passed as parameters wrapped in `with_inputs()` or `with_outputs()`, respectively, to the create_sdf_node() call.

During the execution of the graph, the SDF runtime will invoke `body` with an argument list of references to elements of types `T1` to `Tn`, corresponding to a single element from each of the connected channels. All input channels would have been popped by the SDF runtime so `body` can read a value for each of those channels. `body` must subsequently write a value to the elements corresponding to the output channels, which the SDF runtime will subsequently push on those channels.

Parameters

g	Handle to the graph in which this node is to be created.
body	A function or callable object that accepts references to an element of each of the connected channels. `body` can also be a programmatic introspection function.
io_channels_groups	A sequence of channels, in groups of `io_channels` created with_inputs() and/or with_outputs(). The ordering of the channels must correspond to the types of the parameters accepted by `body`.

Returns

A handle to the created node.

See Also

node_channels for programmatic introspection body

Due to some limitations in Visual Studio 2013 support for C++11, only upto 5 `io_channels_groups` are allowed when compiling with VS 2013 (no limitation when compiling with gcc or clang).

8.2.4.7 void mare::assign_cost (sdf_node_ptr *n,* double *execution_cost*)

Assigns an execution cost to the node (cost defaults to 1.0, if unspecified).

The current execution model of SDF needs node costs as estimates of the relative execution times of nodes, in order to perform load balancing. The next SDF release will perform load balancing without the use of assign_cost().

Parameters

n	Handle to an SDF node.
execution_cost	The execution cost to assign to the node.

8.2.4.8 sdf_graph_ptr mare::get_graph_ptr (sdf_node_ptr *node*)

Retrieve the graph that contains the specified node.

Parameters

node	Handle to an SDF node.

Returns

Handle to the graph that contains the node.

8.2.4.9 std::size_t mare::get_debug_id (sdf_node_ptr *node*)

Retrieves the unique integer ID automatically assigned to the node within its graph.

At node creation (via create_sdf_node()), an integer ID is automatically generated and assigned to the node. The value of the ID starts from 0 for the first node created in the graph and increments for each subsequent node created in the same graph. The ID is provided only as a convenience for the user-code: to aid debugging, identify creation order and identify uniquely. The ID is fixed once the node is created.

Parameters

node	Handle to an SDF node.

Returns

The integer ID assigned to the node.

8.2.4.10 void mare::launch_and_wait (sdf_graph_ptr *g*)

Launches execution of an SDF graph for an unbounded number of graph iterations, and blocks until the graph is cancelled.

Runtime error if graph g was previously launched. Runtime error if graph g is structurally incomplete, i.e., there are channels unconnected on one end. Runtime error if graph g is not schedulable due to 0-delay cycles: a cycle must have at least one channel with at least one preloaded value.

Parameters

g	Handle to the SDF graph to launch.

See Also

preload_channel()

8.2.4.11 void mare::launch_and_wait (sdf_graph_ptr *g,* std::size_t *num_iterations*)

Launches execution of an SDF graph for a specified number of graph iterations, and blocks until the graph completes or is cancelled.

All nodes run for exactly num_iterations, unless cancelled.

Parameters

g	Handle to the SDF graph to launch.
num_iterations	Number of graph iterations to run g.

See Also

launch_and_wait(sdf_graph_ptr g) for the possible runtime errors on g.

8.2.4.12 void mare::launch (sdf_graph_ptr *g*)

Asynchronous launch of an SDF graph for an unbounded number of graph iterations. Does not block for the graph to launch or complete.

Parameters

g	Handle to the SDF graph to launch.

See Also

launch_and_wait(sdf_graph_ptr g) for the possible runtime errors on g.
wait_for() to block on an asynchronously launched graph.

8.2.4.13 void mare::launch (sdf_graph_ptr *g,* std::size_t *num_iterations*)

Asynchronous launch of an SDF graph for a specified number of graph iterations. Does not block for the graph to launch or complete.

All nodes run for exactly num_iterations, unless cancelled.

Parameters

g	Handle to the SDF graph to launch.
num_iterations	Number of graph iterations to run g.

See Also

launch_and_wait(sdf_graph_ptr g) for the possible runtime errors on g.
wait_for() to block on an asynchronously launched graph.

8.2.4.14 void mare::wait_for (sdf_graph_ptr *g*)

Blocks on an asynchronously launched graph to complete execution or be cancelled.

Multiple wait_for() calls are allowed to block on same graph. A wait_for() invoked after the graph has already completed or been cancelled will return immediately.

Parameters

g	Handle to the SDF graph to block on.

8.2.4.15 void mare::cancel (sdf_graph_ptr *g,* sdf_interrupt_type *intr_type = sdf_interrupt_type::iter_non_synced*)

Cancels the execution of a graph ASAP.

A graph can be cancelled

- before it launches (on launch, graph will terminate without execution)
- while it is executing
- or, after it has already finished execution (no effect)

cancel() returns immediately, while any wait_for() will block until the cancel takes effect.

Parameters

g	Handle to SDF graph to cancel.
intr_type	Interruption type requested.

- `intr_type == iter_non_synced`:
  - Non-iteration-synchronized cancel, no guarantee that all nodes will stop on the same graph iteration. But executing nodes are not interrupted.

- `intr_type == iter_synced`:
  - Iteration-synchronized cancel, all graph nodes complete the exact same number of graph iterations.

See Also

sdf_interrupt_type

Okay to cancel either from program external to graph g or from within a node of g.

Interruption Request Queue

Each graph has its own interruption-request queue. cancel() enqueues an interruption request for the SDF runtime to process. Multiple incoming requests get enqueued and are processed in order by the graph's runtime. Similarly, pause() also enqueues an interruption requests into the same interruotion queue. When a cancel request earlier in the queue takes effect, it renders the remaining requests to have no effect (all pauses will unblock, resumes will have no effect).

See Also

wait_for(), pause(), resume()

8.2.4.16 void mare::cancel (sdf_graph_ptr *g*, std::size_t *desired_cancel_iteration*, sdf_interrupt_type *intr_type* = *`sdf_interrupt_type::iter_non_synced`*)

Cancels the execution of a graph after all nodes have reached or exceeded a specified number of graph iterations.

Will ∗attempt∗ to cancel exactly when all nodes have completed exactly `desired_cancel_iteration` iteration. If the interruption queue already has a prior request that is not processed as yet, then it is guaranteed∗ that the cancel will happen precisely when all nodes complete `desired_cancel_iteration` (i.e., `iter_synced` semantics), regardless of the `intr_type` in the cancel request.

Parameters

g	Handle to the SDF graph to cancel.
desired_cancel_-iteration	Graph iteration that all nodes must complete before cancel.
intr_type	Interruption type requested.

- `intr_type == iter_non_synced`:
  - some nodes may execute additional iterations, though every node will execute till `desired_cancel_iteration` iteration.
- `intr_type == iter_synced`:
  - all nodes will complete the exact same number of iterations, possibly past the `desired_cancel_iteration` iteration.

See Also

sdf_graph_query() determines on which graph iteration the cancel took effect and with what interrution semantics.

8.2.4.17 bool mare::pause (sdf_graph_ptr *g*, sdf_interrupt_type *intr_type* = *sdf_interrupt_type::iter_non_synced*)

Pauses the execution of a graph ASAP.

A graph can be paused

- before it launches (on launch, graph will pause before executing)
- while it is executing
- or, after it has already finished execution (no effect)

pause() blocks until the graph has either paused execution in response to this particular pause request (not any other pause request), or has terminated for other reasons (such as a preceding cancel request in the graph's interruption queue).

resume() invoked after a pause() causes the graph to resume execution precisely from where it paused.

Parameters

g	Handle to SDF graph to pause.
intr_type	Interruption type requested.

- `intr_type == iter_non_synced`:
  - Non-iteration-synchronized pause, no guarantee that all nodes will pause on the same graph iteration. But executing nodes are not interrupted.
- `intr_type == iter_synced`:
  - Iteration-synchronized pause, all graph nodes complete the exact same number of graph iterations.

Returns

TRUE if this pause took effect
FALSE if graph got terminated or had already completed.

See Also

sdf_interrupt_type
cancel(sdf_graph_ptr g, sdf_interrupt_type intr_type) for details on the interruption request queue.

Okay to pause either from program external to graph g or from within a node of g.

See Also

resume()

8.2.4.18 bool mare::pause (sdf_graph_ptr *g*, std::size_t *desired_pause_iteration*, sdf_interrupt_type *intr_type* = *sdf_interrupt_type::iter_non_synced*)

Pauses the execution of a graph after all nodes have reached or exceeded a specified number of graph iterations.

Parameters

g	Handle to the SDF graph to pause.
desired_pause_-iteration	Graph iteration that all nodes must complete before cancel.
intr_type	Interruption type requested.

Returns

TRUE if this pause took effect
FALSE if graph got terminated or had already completed.

See Also

pause(sdf_graph_ptr g, sdf_interrupt_type intr_type) for details on pause blocking semantics, interruption semantics and resume semantics.

8.2.4.19 void mare::resume (sdf_graph_ptr *g*)

Resumes execution after the graph has been paused.

Non-blocking. Runtime error if the graph was not paused when resume() is called. No effect if the graph is already completed or cancelled. Can be called from any thread or task, not necessarily from where pause was executed.

Parameters

g	Handle to the SDF graph to pause.

See Also

pause(sdf_graph_ptr g, sdf_interrupt_type intr_type);
pause(sdf_graph_ptr g, std::size_t desired_pause_iteration, sdf_interrupt_type intr_type)

8.2.4.20 sdf_graph_query_info mare::sdf_graph_query (sdf_graph_ptr *g*)

Queries the state of an SDF graph.

Parameters

g	Handle to graph to query.

Returns

Query information that samples the current execution state of the graph.

See Also

sdf_graph_query_info for a description of the graph query information.

8.2.4.21 std::string mare::to_string (sdf_graph_query_info const & *info*)

Converts the graph query information to a string suitable for printing.

Parameters

info	The graph query information.

Returns

Query information formatted into a string suitable for printing.

8.2.4.22 template<typename OStream > OStream& mare::operator<< (OStream & *os,* sdf_graph_query_info const & *info*)

Insertion operator to output formatted graph query information.

Parameters

os	The output stream
info	The graph query information.

8.2.4.23 template<typename T > tuple_dir_channel mare::as_in_channel_tuple (data_channel< T > & *dc*)

Binds a channel as in-channel for connecting to a node.

Used for connecting the channel to a programmatically connected node.

Parameters

dc	A data-channel templated on a user-data-type.

Returns

A template-free binding of the channel as in-channel, suitable for passing around to generic parts of the application that are not specific to the user-data-type used for channel construction.

Definition at line 168 of file sdfpr.hh.

8.2.4.24 template<typename T > tuple_dir_channel mare::as_out_channel_tuple (data_channel< T > & *dc*)

Binds a channel as out-channel for connecting to a node.

Used for connecting the channel to a programmatically connected node.

Parameters

dc	A data-channel templated on a user-data-type.

Returns

A template-free binding of the channel as out-channel, suitable for passing around to generic parts of the application that are not specific to the user-data-type used for channel construction.

Definition at line 187 of file sdfpr.hh.

8.2.4.25 sdf_node_ptr mare::create_sdf_node (sdf_graph_ptr *g,* void(∗)(node_channels &) *body,* std::vector< tuple_dir_channel > & *v_dir_channels*) [inline]

Programmatic connection of a node to channels, using a body with programmatic introspection.

The signature of `body` does not carry template information on the user-data-types of the connected channels, the number of connected channels or their directions.

Parameters

g	Handle to the SDF graph in which the node is to created.
body	An introspection function on node_channels.
v_dir_channels	Captures the channels connected to the node along with their directions. Expressed as a vector of channel-direction bindings.

Returns

A handle to the created node.

See Also

node_channels on how `body` can introspect on the channels connected to it.
as_in_channel_tuple() for creation of an in-channel binding.
as_out_channel_tuple() for creation of an out-channel binding.

8.2.4.26 template<typename Body > sdf_node_ptr mare::create_sdf_node (sdf_graph_ptr *g,* Body && *body,* std::vector< tuple_dir_channel > & *v_dir_channels*)

Programmatic connection of a node to channels, using a body with type-signature listing the user-data-types of connected channels.

`body` is a callable (e.g., a function, object with operator(), a lambda with capture) whose callable type-signature lists the user-data-types of the connected channels. `body` is callable with a type signature like the following:

```
body(T1& t1, T2& t2, ..., Tn& tn)
```

where `T1` to `Tn` are the user-data-types of the channels connected to the node, though not visible to the compiler in the channel descriptions in `v_dir_channels`.

The channels are expressed without user-data-type information. This form of create_sdf_node() is useful when the channels may have been passed through generic parts of the application code that was not specialized to the user-data-types used in channel creation, but node creation occurs within specialized code aware of the channel user-data-types.

Parameters

g	Handle to the SDF graph in which the node is to created.
body	A callable with type-signature explicitly listing the user-data-types of the connected channels.
v_dir_channels	Captures the channels connected to the node along with their directions. Expressed as a vector of channel-direction bindings.

Returns

A handle to the created node.

See Also

as_in_channel_tuple() for creation of an in-channel binding.
as_out_channel_tuple() for creation of an out-channel binding.

9 Tasks Reference API

Tasks represent independent units of work that can be executed asynchronously. MARE programmers are responsible for partitioning their application into tasks and organizing them into a task graph using dependencies. This chapter documents the interfaces to create tasks, setup dependencies, and launch (execute) tasks. It also discusses task synchronization (waiting) and cancelation. Grouping is the mechanism to wait and cancel on a set of tasks. And finally, attributes is a more advanced feature which allows programmers to pass additional information about task behavior to the MARE runtime system.

9.1 Task Objects

Classes

- class mare::task_attrs
- class mare::body_with_attrs< Body >
- class mare::body_with_attrs_gpu< Body, KernelPtr, Kargs...>
- class mare::body_with_attrs< Body, Cancel_Handler >
- class mare::unsafe_task_ptr
- class mare::task_ptr

Functions

- template<typename Body >
  MARE_CONSTEXPR body_with_attrs
  < Body > mare::with_attrs (task_attrs const &attrs, Body &&body)
- template<typename Body , typename Cancel_Handler >
  MARE_CONSTEXPR body_with_attrs
  < Body, Cancel_Handler > mare::with_attrs (task_attrs const &attrs, Body &&body,
  Cancel_Handler &&handler)
- template<typename KernelPtr , typename... Kargs>
  MARE_CONSTEXPR
  body_with_attrs_gpu
  < CpuKernelType, KernelPtr,
  Kargs...> mare::with_attrs (task_attrs const &attrs, KernelPtr kernel, Kargs...args)
- bool operator== (mare::unsafe_task_ptr const &a, mare::unsafe_task_ptr const &b)
- bool operator== (mare::unsafe_task_ptr const &a, std::nullptr_t)
- bool operator== (std::nullptr_t, mare::unsafe_task_ptr const &a)
- bool operator!= (mare::unsafe_task_ptr const &a, mare::unsafe_task_ptr const &b)
- bool operator!= (mare::unsafe_task_ptr const &ptr, std::nullptr_t)
- bool operator!= (std::nullptr_t, mare::unsafe_task_ptr const &ptr)
- bool operator== (mare::task_ptr const &a, mare::task_ptr const &b)
- bool operator== (mare::task_ptr const &ptr, std::nullptr_t)
- bool operator== (std::nullptr_t, mare::task_ptr const &ptr)
- bool operator!= (mare::task_ptr const &a, mare::task_ptr const &b)
- bool operator!= (mare::task_ptr const &ptr, std::nullptr_t)
- bool operator!= (std::nullptr_t, mare::task_ptr const &ptr)
- bool operator== (mare::group_ptr const &a, mare::group_ptr const &b)
- bool operator== (mare::group_ptr const &ptr, std::nullptr_t)

- bool operator== (std::nullptr_t, mare::group_ptr const &ptr)
- bool operator!= (mare::group_ptr const &a, mare::group_ptr const &b)
- bool operator!= (mare::group_ptr const &ptr, std::nullptr_t)
- bool operator!= (std::nullptr_t, mare::group_ptr const &ptr)

9.1.1 Class Documentation

9.1.1.1 class mare::task_attrs

Stores the attributes for a task. Use create_task_attrs(...) to create a task_attrs object.

Definition at line 99 of file attrobjs.hh.

Public member functions

- MARE_CONSTEXPR task_attrs (task_attrs const &other)
- task_attrs & operator= (task_attrs const &other)

Private member functions

- MARE_CONSTEXPR **task_attrs** (std::int32_t mask)

Private Attributes

- std::int32_t **_mask**

Friends

- constexpr **bool::operator==** (task_attrs const &a, task_attrs const &b)
- constexpr **bool::operator!=** (task_attrs const &a, task_attrs const &b)
- constexpr task_attrs create_task_attrs ()
- template<typename Attribute , typename... Attributes>
  constexpr task_attrs **create_task_attrs** (Attribute const &, Attributes const &...)
- template<typename Attribute >
  constexpr bool has_attr (task_attrs const &attrs, Attribute const &attr)
- template<typename Attribute >
  constexpr task_attrs remove_attr (task_attrs const &attrs, Attribute const &attr)
- template<typename Attribute >
  const task_attrs add_attr (task_attrs const &attrs, Attribute const &attr)

9.1.1.1.1 Constructors and Destructors

9.1.1.1.1.1 MARE_CONSTEXPR mare::task_attrs::task_attrs (task_attrs const & *other*) [inline]

Copy Constructor.

Parameters

other	Original task_attrs.

Definition at line 105 of file attrobjs.hh.

9.1.1.1.2 Member Function Documentation

9.1.1.1.2.1 task_attrs& mare::task_attrs::operator= (task_attrs const & *other*) [inline]

Copy assignment.

Parameters

other	Original task_attrs.

Definition at line 112 of file attrobjs.hh.

9.1.1.1.3 Related Function Documentation

9.1.1.1.3.1 constexpr task_attrs create_task_attrs ( ) [friend]

Creates an empty task_attrs object.

Returns

Empty task_attrs object.

Example

```
1 mare::task_attrs attrs = mare::create_task_attrs();
```

See Also

mare::create_task_attrs(Attribute const&, Attributes const&...)

Definition at line 200 of file attr.hh.

9.1.1.1.3.2 template<typename Attribute > constexpr bool has_attr (task_attrs const & *attrs,* Attribute const & *attr*) [friend]

Checks whether a task_attrs object includes a certain attribute.

Parameters

attrs	task_attrs object to query.
attr	Attribute.

Returns

TRUE – task_attrs Includes attribute.
FALSE – task_attrs Does not include attribute.

Example

```
1 ]//Creates task_attrs for a blocking task
2 auto attr1 = mare::create_task_attrs(mare::attr::blocking);
3
4 //It will never fire
5 assert(mare::has_attr(attr1, mare::attr::blocking));
6
7 //Creates empty task_attrs
8 auto attr2 = mare::create_task_attrs();
9
10 //It will always fire
11 assert(mare::has_attr(attr2, mare::attr::blocking));
```

Definition at line 132 of file attr.hh.

9.1.1.1.3.3 template<typename Attribute > constexpr task_attrs remove_attr (task_attrs const & *attrs,* Attribute const & *attr*) [friend]

Removes attribute from a task_attr object.

Returns a new task_attr object that includes all the attributes in the original task_attr, except the one specified in the argument list. If the attribute was not in the original task_attr object, the returned object is identical to the original object.

Parameters

attrs	Original task_attrs object.
attr	Attribute.

Returns

task_attrs – Includes all the attributes in attrs except attr. If task_attrs does not include attr, the returned task_attrs is identical to attrs.

Definition at line 156 of file attr.hh.

9.1.1.1.3.4 template<typename Attribute > const task_attrs add_attr (task_attrs const & *attrs,* Attribute const & *attr*) [friend]

Adds attribute to a mare::task_attrs object.

Returns a new task_attrs object that includes all the attributes in the original task_attr, plus the one specified in the argument list. If the attribute was already in the original task_attrs object, the returned object is identical to the original object.

Parameters

attrs	Original task_attrs object.
attr	Attribute.

Returns

task_attrs – Includes all the attibutes in attrs plus attr. If task_attrs already includes attr, the returned task_attrs is identical to attrs.

Example

```
1 // Creates empty attr
2 mare::task_attrs empty_attrs = mare::create_task_attrs();
3
4 // Adds blocking attribute
5 auto new_attrs1 = mare::add_attr(empty_attrs, mare::attr::blocking);
6
7 // Adds blocking attribute again
8 auto new_attrs2 = mare::add_attr(new_attrs1, mare::attr::blocking);
9
10 // It will never fireo
11 assert (new_attrs1 == new_attrs2);
```

Definition at line 182 of file attr.hh.

9.1.1.2 class mare::body_with_attrs< Body >

template<typename Body>class mare::body_with_attrs< Body >

Temporarily stores attributes and other information needed to create a task. Create them using the template method mare::with_attrs.

Definition at line 161 of file attrobjs.hh.

Public Types

- typedef internal::function_traits < Body > body_traits
- typedef body_traits::return_type return_type

Public member functions

- MARE_CONSTEXPR task_attrs const & get_attrs () const
- MARE_CONSTEXPR Body & get_body () const
- template<typename... Args>
  MARE_CONSTEXPR return_type operator() (Args...args) const

Private member functions

- constexpr **body_with_attrs** (task_attrs const &attrs, Body &&body)

Private Attributes

- task_attrs const & **_attrs**
- Body & **_body**

9.1.1.2.1 Member Typedef Documentation

9.1.1.2.1.1 template<typename Body > typedef internal::function_traits<Body> mare::body_with_attrs< Body >::body_traits

Traits of the body.

Definition at line 165 of file attrobjs.hh.

9.1.1.2.1.2 template<typename Body > typedef body_traits::return_type mare::body_with_attrs< Body >::return_type

Returns

type of the body.

Definition at line 167 of file attrobjs.hh.

9.1.1.2.2 Member Function Documentation

9.1.1.2.2.1 template<typename Body > MARE_CONSTEXPR task_attrs const& mare::body_with_attrs< Body >::get_attrs ( ) const [inline]

Returns body attributes.

Returns

task_attrs – body attributes.

Definition at line 173 of file attrobjs.hh.

9.1.1.2.2.2 template<typename Body > MARE_CONSTEXPR Body& mare::body_with_attrs< Body >::get_body ( ) const [inline]

Returns body.

Returns

Body – body.

Definition at line 179 of file attrobjs.hh.

9.1.1.2.2.3 template<typename Body > template<typename... Args> MARE_CONSTEXPR return_type mare::body_with_attrs< Body >::operator() (Args... *args*) const [inline]

Calls body.

Returns

Parameters

args	Arguments to be passed to the body.

return_type – Returns the value returned by the body.

Definition at line 188 of file attrobjs.hh.

9.1.1.3 class mare::body_with_attrs_gpu< Body, KernelPtr, Kargs...>

template<typename Body, typename KernelPtr, typename... Kargs>class mare::body_with_attrs_gpu< Body, KernelPtr, Kargs...>

Temporarily stores attributes and other information needed to create a gpu task. Create them using the template method mare::with_attrs.

Definition at line 238 of file attrobjs.hh.

Public Types

- typedef internal::function_traits < Body > body_traits
- typedef body_traits::return_type return_type
- typedef KernelPtr::type::parameters kernel_parameters
- typedef std::tuple< Kargs...> kernel_arguments

Public member functions

- MARE_CONSTEXPR task_attrs const & get_attrs () const
- MARE_CONSTEXPR Body & get_body () const
- template<typename... Args>
  MARE_CONSTEXPR return_type operator() (Args...args) const
- KernelPtr & get_gpu_kernel ()
- kernel_arguments & get_kernel_args ()

Private member functions

- **MARE_GCC_IGNORE_END** ("-Weffc++")
- MARE_CONSTEXPR **body_with_attrs_gpu** (task_attrs const &attrs, Body &&body, KernelPtr kernel, Kargs &&...args)

Private Attributes

- task_attrs const & **_attrs**
- Body & **_body**
- KernelPtr **_kernel**
- kernel_arguments **_kargs**

9.1.1.3.1 Member Typedef Documentation

9.1.1.3.1.1 template<typename Body , typename KernelPtr , typename... Kargs> typedef internal::function_traits<Body> mare::body_with_attrs_gpu< Body, KernelPtr, Kargs...>::body_traits

Traits of the body.

Definition at line 246 of file attrobjs.hh.

9.1.1.3.1.2 template<typename Body , typename KernelPtr , typename... Kargs> typedef body_traits-::return_type mare::body_with_attrs_gpu< Body, KernelPtr, Kargs...>::return_type

Return type of the body.

Definition at line 248 of file attrobjs.hh.

9.1.1.3.1.3 template<typename Body , typename KernelPtr , typename... Kargs> typedef KernelPtr::type::parameters mare::body_with_attrs_gpu< Body, KernelPtr, Kargs...>-::kernel_parameters

Kernel parameters

Definition at line 250 of file attrobjs.hh.

9.1.1.3.1.4 template<typename Body , typename KernelPtr , typename... Kargs> typedef std::tuple<Kargs...> mare::body_with_attrs_gpu< Body, KernelPtr, Kargs...>::kernel_arguments

Kernel arguments

Definition at line 252 of file attrobjs.hh.

9.1.1.3.2 Member Function Documentation

9.1.1.3.2.1 template<typename Body , typename KernelPtr , typename... Kargs> MARE_CONSTEXPR task_attrs const& mare::body_with_attrs_gpu< Body, KernelPtr, Kargs...>::get_attrs ( ) const [inline]

Returns body attributes

Returns

task_attrs — body attributes

Definition at line 259 of file attrobjs.hh.

9.1.1.3.2.2 template<typename Body , typename KernelPtr , typename... Kargs> MARE_CONSTEXPR Body& mare::body_with_attrs_gpu< Body, KernelPtr, Kargs...>::get_body ( ) const [inline]

Returns body

Returns

Body — body

Definition at line 265 of file attrobjs.hh.

9.1.1.3.2.3 template<typename Body , typename KernelPtr , typename... Kargs> template<typename... Args> MARE_CONSTEXPR return_type mare::body_with_attrs_gpu< Body, KernelPtr, Kargs...>::operator() (Args... *args*) const [inline]

Calls body

Parameters

args	Arguments to be passed to the body.

Returns

return_type — Returns the value returned by the body.

Definition at line 273 of file attrobjs.hh.

9.1.1.3.2.4 template<typename Body , typename KernelPtr , typename... Kargs> KernelPtr& mare::body_with_attrs_gpu< Body, KernelPtr, Kargs...>::get_gpu_kernel ( ) [inline]

Returns the kernel ptr passed to a gpu task. The kernel ptr points to a template object, the template parameters has the same signature as the OpenCL kernel parameters used in the current gpu task. This is used to check at compile time if the kernel arguments passed to gpu task have same type as the kernel parameters.

Returns

Returns the kernel arguments passed to a gpu task.

See Also

create_kernel()

Definition at line 285 of file attrobjs.hh.

9.1.1.3.2.5 template<typename Body , typename KernelPtr , typename... Kargs> kernel_arguments& mare::body_with_attrs_gpu< Body, KernelPtr, Kargs...>::get_kernel_args ( ) [inline]

Returns the kernel arguments passed to a gpu task.

Returns

Returns the kernel arguments passed to a gpu task.

Definition at line 290 of file attrobjs.hh.

9.1.1.4 class mare::body_with_attrs< Body, Cancel_Handler >

template<typename Body, typename Cancel_Handler>class mare::body_with_attrs< Body, Cancel_Handler >

Temporarily stores attributes and other information needed to create a task. Create them using the template method with_attrs.

Definition at line 331 of file attrobjs.hh.

Public Types

- typedef internal::function_traits < Body > body_traits
- typedef body_traits::return_type return_type
- typedef

internal::function_traits
< Cancel_Handler > cancel_handler_traits

- typedef
cancel_handler_traits::return_type cancel_handler_return_type

Public member functions

- MARE_CONSTEXPR task_attrs const & get_attrs () const
- MARE_CONSTEXPR Body & get_body () const
- MARE_CONSTEXPR Cancel_Handler & get_cancel_handler () const
- template<typename... Args>
MARE_CONSTEXPR return_type operator() (Args...args) const

Private member functions

- constexpr **body_with_attrs** (task_attrs const &attrs, Body &&body, Cancel_Handler &&handler)

Private Attributes

- task_attrs const & **_attrs**
- Body & **_body**
- Cancel_Handler & **_handler**

9.1.1.4.1 Member Typedef Documentation

9.1.1.4.1.1 template<typename Body , typename Cancel_Handler > typedef internal::function_traits<Body> mare::body_with_attrs< Body, Cancel_Handler >::body_traits

Traits of the body.

Definition at line 336 of file attrobjs.hh.

9.1.1.4.1.2 template<typename Body , typename Cancel_Handler > typedef body_traits::return_type mare::body_with_attrs< Body, Cancel_Handler >::return_type

Return type of the body.

Definition at line 338 of file attrobjs.hh.

9.1.1.4.1.3 template<typename Body , typename Cancel_Handler > typedef internal::function_traits<Cancel_Handler> mare::body_with_attrs< Body, Cancel_Handler >::cancel_handler_traits

Traits of the cancel handler.

Definition at line 341 of file attrobjs.hh.

9.1.1.4.1.4 template<typename Body , typename Cancel_Handler > typedef cancel_handler_traits-::return_type mare::body_with_attrs< Body, Cancel_Handler >::cancel_handler_return_type

Return type of the cancel handler.

Definition at line 344 of file attrobjs.hh.

9.1.1.4.2 Member Function Documentation

9.1.1.4.2.1 template<typename Body , typename Cancel_Handler > MARE_CONSTEXPR task_attrs const& mare::body_with_attrs< Body, Cancel_Handler >::get_attrs ( ) const [inline]

Returns body attributes.

Returns

task_attrs – Body attributes.

Definition at line 350 of file attrobjs.hh.

9.1.1.4.2.2 template<typename Body , typename Cancel_Handler > MARE_CONSTEXPR Body& mare::body_with_attrs< Body, Cancel_Handler >::get_body ( ) const [inline]

Returns body.

Returns

Body – body.

Definition at line 358 of file attrobjs.hh.

9.1.1.4.2.3 template<typename Body , typename Cancel_Handler > MARE_CONSTEXPR Cancel_-Handler& mare::body_with_attrs< Body, Cancel_Handler >::get_cancel_handler ( ) const [inline]

Returns cancel handler.

Returns

cancel_handler_type – Cancel_handler.

Definition at line 366 of file attrobjs.hh.

9.1.1.4.2.4 template<typename Body , typename Cancel_Handler > template<typename... Args> MARE_CONSTEXPR return_type mare::body_with_attrs< Body, Cancel_Handler >::operator() (Args... *args*) const [inline]

Calls body

Returns

Parameters

args	Arguments to be passed to the body. return_type – Returns the value returned by the body.

Definition at line 376 of file attrobjs.hh.

9.1.1.5 class mare::unsafe_task_ptr

An unsafe pointer to a task.

This pointer does not do any reference counting and does not affect the lifetime of the task. It is safe to use it only as long as a task_ptr to the same task also exists. Unsafe task pointers are primarily useful because they can be copied and passed around very inexpensively.

See Also

mare::task_ptr

Definition at line 24 of file mareptrs.hh.

Public member functions

- constexpr unsafe_task_ptr ()
- constexpr unsafe_task_ptr (std::nullptr_t)
- unsafe_task_ptr (unsafe_task_ptr const &other)
- unsafe_task_ptr (unsafe_task_ptr &&other)
- ~unsafe_task_ptr ()
- unsafe_task_ptr & operator= (unsafe_task_ptr const &other)
- unsafe_task_ptr & operator= (unsafe_task_ptr &&other)
- void reset ()
- void swap (unsafe_task_ptr &other)
- bool unique () const
- operator bool () const

9.1.1.5.1 Constructors and Destructors

9.1.1.5.1.1 constexpr mare::unsafe_task_ptr::unsafe_task_ptr ( )

Default constructor. Constructs null pointer.

9.1.1.5.1.2 constexpr mare::unsafe_task_ptr::unsafe_task_ptr (std::nullptr_t)

Constructs null pointer.

9.1.1.5.1.3 mare::unsafe_task_ptr::unsafe_task_ptr (unsafe_task_ptr const & *other*)

Copy constructor. Constructs pointer to the same location as other

Parameters

other	Another pointer used to copy-construct the pointer.

9.1.1.5.1.4 mare::unsafe_task_ptr::unsafe_task_ptr (unsafe_task_ptr && *other*)

Move constructor. Constructs pointer to the same location as other using move semantics.

Parameters

other	Another pointer used to copy-construct the pointer.

9.1.1.5.1.5 mare::unsafe_task_ptr::∼unsafe_task_ptr ( )

Destructor.

9.1.1.5.2 Member Function Documentation

9.1.1.5.2.1 unsafe_task_ptr& mare::unsafe_task_ptr::operator= (unsafe_task_ptr const & *other*)

Assigns address to pointer.

Parameters

other	Pointer used as source for the assignment.

Returns

∗this

9.1.1.5.2.2 unsafe_task_ptr& mare::unsafe_task_ptr::operator= (unsafe_task_ptr && *other*)

Assigns address to pointer using move semantics.

Parameters

other	Pointer used as source for the assignment.

Returns

∗this

9.1.1.5.2.3 void mare::unsafe_task_ptr::reset ( )

Resets pointer.

9.1.1.5.2.4 void mare::unsafe_task_ptr::swap (unsafe_task_ptr & *other*)

Swaps the location the pointer points to.

Parameters

other	Pointer used to exchange targets with.

9.1.1.5.2.5 bool mare::unsafe_task_ptr::unique ( ) const

Checks whether there is only one pointer pointing to the object.

Returns

FALSE – It always returns false because it is an unsafe pointer.

9.1.1.5.2.6 mare::unsafe_task_ptr::operator bool ( ) const

Checks whether the pointer is not a nullptr.

Returns

TRUE – The pointer is not a nullptr.
FALSE – The pointer is a nullptr.

9.1.1.6 class mare::task_ptr

An std::shared_ptr-like managed pointer to a task object.

mare::create_task returns an object of type mare::task_ptr, which is a custom smart pointer to the task object.

Tasks are reference counted, so they are automatically destroyed when no more mare::task_ptr pointers point to them. When a task launches, MARE runtime increments the task's reference counter. This prevents the task from being destroyed even if all mare::task_ptr pointers pointing to the task are reset. The MARE runtime decrements the task's reference counter after the task completes execution.

The task reference counter requires atomic operations. Copying a mare::task_ptr pointer requires an atomic increment, and an atomic decrement when the newly created mare::task_ptr pointer goes out of scope. For best results, minimize the number of times your application copies group_ptr pointers.

Some algorithms require constantly passing mare::task_ptr pointers. To prevent a decrease in performance, MARE provides another task pointer type that does not perform reference counting: mare::unsafe_task_ptr.

The following example demonstrates how to point `mare::unsafe_task_ptr` to a task:

```
1 // Create task that prints Hello World!
2 mare::task_ptr t1 = mare::create_task([]{
3     printf("Hello World!\n");
4   });
5
6 // Get unsafe_task_ptr pointing to task
7 mare::unsafe_task_ptr unsafe_t1 = t1.get();
```

Task lifetime is determined by the the number of mare::task_ptr pointers pointing to it. Programmers must avoid using a mare::unsafe_task_ptr unless there is at least one mare::task_ptr pointing to it because it will likely cause memory corruption or segmentation fault. You can use a mare::unsafe_task_ptr pointer in any API method in which you can use a mare::task_ptr, with the exception of mare::launch_and_reset(mare::task_ptr&).

Warning

Never use a reference to a mare::task_ptr pointer, as it can cause undefined behavior. This is because some MARE API methods are highly optimized to support the case where there is only one mare::task_ptr pointing to a task. Unfortunately, MARE cannot always prevent the programmer from taking a reference to a mare::task_ptr, so it is up to the programmer to ensure that this does not happen. DO NOT reference a MARE task pointer like the following example:

```
1 mare::task_ptr t1 = mare::create_task([] {
2     printf("Hello World from t1!\n");
3   });
4 mare::task_ptr t2 = mare::create_task( [&t1] {
5     printf("Hello World from t2!\n");
6   });
```

Instead, copy the pointer.

```
1 mare::task_ptr t1 = mare::create_task([] {
2     printf("Hello World from t1!\n");
3   });
4 mare::task_ptr t2 = mare::create_task( [t1] {
5     printf("Hello World from t2!\n");
6   });
```

Or use a mare::unsafe_task_ptr (of course, ensure that `t1` does not go out of scope):

```
1 mare::task_ptr t1 = mare::create_task([] {
2     printf("Hello World from t1!\n");
```

```
3   });
4
5 mare::unsafe_task_ptr unsafe_t1= t1.get();
6
7 mare::task_ptr t2 = mare::create_task([unsafe_t1] {
8     printf("Hello World from t2!\n");
9   });
10
11 mare::task_ptr t3 = mare::create_task([&unsafe_t1] {
12     printf("Hello World from t3!\n");
13   });
```

See Also

mare::group_ptr
mare::unsafe_task_ptr

Definition at line 163 of file mareptrs.hh.

Public member functions

- constexpr task_ptr ()
- constexpr task_ptr (std::nullptr_t)
- task_ptr (task_ptr const &other)
- task_ptr (task_ptr &&other)
- ~task_ptr ()
- task_ptr & operator= (task_ptr const &other)
- task_ptr & operator= (std::nullptr_t)
- task_ptr & operator= (task_ptr &&other)
- void swap (task_ptr &other)
- unsafe_task_ptr get ()
- void reset ()
- operator bool () const
- size_t use_count () const
- bool unique () const

9.1.1.6.1 Constructors and Destructors

9.1.1.6.1.1 constexpr mare::task_ptr::task_ptr ( )

Default constructor. Constructs null pointer.

9.1.1.6.1.2 constexpr mare::task_ptr::task_ptr (std::nullptr_t)

Constructs null pointer.

9.1.1.6.1.3 mare::task_ptr::task_ptr (task_ptr const & *other*)

Copy constructor. Constructs pointer to the same location as other and increases reference count to task.

Parameters

other	Another pointer used to copy-construct the pointer.

9.1.1.6.1.4 mare::task_ptr::task_ptr (task_ptr && *other*)

Move constructor. Constructs pointer to the same location as other using move semantics. Does not increase the reference count to the task.

Parameters

other	Another pointer used to copy-construct the pointer.

9.1.1.6.1.5 mare::task_ptr::∼task_ptr ( )

Destructor. It will destroy the target task if it is the last task_ptr pointing to to the same task and has not yet been launched.

9.1.1.6.2 Member Function Documentation

9.1.1.6.2.1 task_ptr& mare::task_ptr::operator= (task_ptr const & *other*)

Assigns address to pointer and increases reference counting to it.

Parameters

other	Pointer used as source for the assignment.

Returns

∗this

9.1.1.6.2.2 task_ptr& mare::task_ptr::operator= (std::nullptr_t)

Resets the pointer.

Returns

∗this

9.1.1.6.2.3 task_ptr& mare::task_ptr::operator= (task_ptr && *other*)

Assigns address to pointer using move semantics.

Parameters

other	Pointer used as source for the assignment.

Returns

∗this

9.1.1.6.2.4 void mare::task_ptr::swap (task_ptr & *other*)

Swaps the location the pointer points to.

Parameters

other	Pointer used to exchange targets with.

9.1.1.6.2.5 unsafe_task_ptr mare::task_ptr::get ( )

Returns unsafe_task_ptr pointing to the same task.

Returns

unsafe_task_ptr – Unsafe pointer pointing to the same task.

9.1.1.6.2.6 void mare::task_ptr::reset ( )

Resets the pointer.

9.1.1.6.2.7 mare::task_ptr::operator bool ( ) const [explicit]

Checks whether the pointer is not a nullptr.

Returns

TRUE – The pointer is not a nullptr.
FALSE – The pointer is a nullptr.

9.1.1.6.2.8 size_t mare::task_ptr::use_count ( ) const

Counts number of task_ptr pointing to same task.

This is an expensive operation as it requires an atomic operation. Use it sparingly.

Returns

size_t – Number of task_ptr pointing to the same task.

9.1.1.6.2.9 bool mare::task_ptr::unique ( ) const

Checks whether there is only one pointer pointing to the task.

Returns

TRUE – There is only one pointer pointing to the task.

FALSE – There is more than one pointer pointing to the task, or the pointer is null.

9.1.2 Function Documentation

9.1.2.1 template<typename Body > MARE_CONSTEXPR body_with_attrs<Body> mare::with_attrs (task_attrs const & *attrs,* Body && *body*)

Creates a body_with_attrs object that encapsulates the task body and the task attributes.

Parameters

attrs	Attributes.
body	Task body.

Returns

body_with_attrs with task attributes, task body, but no cancel handler.

9.1.2.2 template<typename Body , typename Cancel_Handler > MARE_CONSTEXPR body_with_attrs<Body, Cancel_Handler> mare::with_attrs (task_attrs const & *attrs,* Body && *body,* Cancel_Handler && *handler*)

Creates a body_with_attrs object that encapsulates the task body, the cancel handler, and the task attributes.

Parameters

attrs	Attributes.
body	Task body.
handler	Cancel handler.

Returns

body_with_attrs with task attributes, task body, and cancel handler body.

9.1.2.3 template<typename KernelPtr , typename... Kargs> MARE_CONSTEXPR body_with_attrs_gpu<CpuKernelType, KernelPtr, Kargs...> mare::with_attrs (task_attrs const & *attrs,* KernelPtr *kernel,* Kargs... *args*)

Creates a body_with_attrs object that encapsulates the task attributes, Kernel ptr, the OpenCL C kernel string, and the kernel arguments.

Parameters

attrs	Attributes.
kernel	Kernel pointer which points to a template kernel object.
args	Kernel arguments to be passed to the OpenCL kernel.

Returns

body_with_attrs with task attributes, kernel ptr, OpenCL C kernel string and kernel arguments.

See Also

create_kernel

9.1.2.4 bool operator== (mare::unsafe_task_ptr const & *a,* mare::unsafe_task_ptr const & *b*) [inline]

Compares two unsafe_task_ptr.

Parameters

a	Pointer.
b	Pointer.

Returns

TRUE – The two pointers point to the same task.
FALSE – The two pointers do not point to the same task.

9.1.2.5 bool operator== (mare::unsafe_task_ptr const & *a,* std::nullptr_t) [inline]

Compares an unsafe_task_ptr to nullptr.

Parameters

a	Pointer.

Returns

TRUE – a points to null.
FALSE – a does not point to null.

9.1.2.6 bool operator== (std::nullptr_t , mare::unsafe_task_ptr const & *a*) [inline]

Compares an unsafe_task_ptr to nullptr.

Parameters

a	Pointer.

Returns

TRUE – a points to null.
FALSE – a does not point to null.

9.1.2.7 bool operator!= (mare::unsafe_task_ptr const & *a,* mare::unsafe_task_ptr const & *b*) [inline]

Compares two unsafe_task_ptr.

Parameters

a	Pointer.
b	Pointer.

Returns

TRUE – The two pointers point to the same task.
FALSE – The two pointers do not point to the same task.

9.1.2.8 bool operator!= (mare::unsafe_task_ptr const & *ptr,* std::nullptr_t) [inline]

Compares an unsafe_task_ptr to nullptr.

Parameters

ptr	Pointer.

Returns

TRUE – ptr does not point to null.
FALSE – ptr points to null.

9.1.2.9 bool operator!= (std::nullptr_t , mare::unsafe_task_ptr const & *ptr*) [inline]

Compares an unsafe_task_ptr to nullptr.

Parameters

ptr	Pointer.

Returns

TRUE – ptr does not point to null.
FALSE – ptr points to null.

9.1.2.10 bool operator== (mare::task_ptr const & *a,* mare::task_ptr const & *b*) [inline]

Compares two task_ptr.

Parameters

a	Pointer.
b	Pointer.

Returns

TRUE – The two pointers point to the same task.
FALSE – The two pointers do not point to the same task.

9.1.2.11 bool operator== (mare::task_ptr const & *ptr,* std::nullptr_t) [inline]

Compares a task_ptr to nullptr.

Parameters

ptr	Pointer.

Returns

TRUE – ptr points to null.
FALSE – ptr does not point to null.

9.1.2.12 bool operator== (std::nullptr_t , mare::task_ptr const & *ptr*) [inline]

Compares a task_ptr to nullptr.

Parameters

ptr	Pointer.

Returns

TRUE – ptr points to null.
FALSE – ptr does not point to null.

9.1.2.13 bool operator!= (mare::task_ptr const & *a,* mare::task_ptr const & *b*) [inline]

Compares two task_ptr.

Parameters

a	Pointer.
b	Pointer.

Returns

TRUE – The two pointers do not point to the same task.
FALSE – The two pointers point to the same task.

9.1.2.14 bool operator!= (mare::task_ptr const & *ptr,* std::nullptr_t) [inline]

Compares a task_ptr to nullptr.

Parameters

ptr	Pointer.

Returns

TRUE – ptr does not point to null.
FALSE - ptr points to null.

9.1.2.15 bool operator!= (std::nullptr_t , mare::task_ptr const & *ptr*) [inline]

Compares a task_ptr to nullptr.

Parameters

ptr	Pointer.

Returns

TRUE – ptr does not point to null.
FALSE – ptr points to null.

9.1.2.16 bool operator== (mare::group_ptr const & *a,* mare::group_ptr const & *b*) [inline]

Compares two group_ptr.

Parameters

a	Pointer.
b	Pointer.

Returns

TRUE – The two pointers point to the same group.
FALSE – The two pointers do not point to the same group.

9.1.2.17 bool operator== (mare::group_ptr const & *ptr,* std::nullptr_t) [inline]

Compares a group_ptr to nullptr.

Parameters

ptr	Pointer.

Returns

TRUE – ptr points to null.
FALSE – ptr does not point to null

9.1.2.18 bool operator== (std::nullptr_t , mare::group_ptr const & *ptr*) [inline]

Compares a group_ptr to nullptr

Parameters

ptr	Pointer.

Returns

TRUE – ptr points to null.

FALSE – ptr does not point to null.

9.1.2.19 bool operator!= (mare::group_ptr const & *a,* mare::group_ptr const & *b*) [inline]

Compares two group_ptr.

Parameters

a	Pointer.
b	Pointer.

Returns

TRUE – The two pointers do not point to the same group.
FALSE – The two pointers point to the same group.

9.1.2.20 bool operator!= (mare::group_ptr const & *ptr,* std::nullptr_t) [inline]

Compares a group_ptr to nullptr.

Parameters

ptr	Pointer.

Returns

TRUE – ptr does not point to null.
FALSE – ptr points to null.

9.1.2.21 bool operator!= (std::nullptr_t , mare::group_ptr const & *ptr*) [inline]

Compares a group_ptr to nullptr.

Parameters

ptr	Pointer.

Returns

TRUE – ptr does not point to null.
FALSE – ptr points to null.

9.2 Creation

Functions

- template<size_t DIMS, typename Body >
  task_ptr mare::create_ndrange_task (const range< DIMS > &r, Body &&body)
- template<typename Body >
  task_ptr mare::create_task (Body &&body)
- template<typename Body >
  task_ptr mare::create_task (body_with_attrs< Body > &&attrd_body)
- template<typename Body >
  task_ptr mare::create_task (body_with_attrs< Body, Cancel_Handler > &&attrd_body)

9.2.1 Function Documentation

9.2.1.1 template<size_t DIMS, typename Body > task_ptr mare::create_ndrange_task (const range< DIMS > & *r*, Body && *body*) [inline]

Creates a new task and returns a task_ptr that points to that task.

This template method creates a task and returns a pointer to it. The task executes the `Body` passed as parameter for each point in the iteration space of mare::range specified by the first argument. The preferred type of `<typename Body>` is a C++11 lambda, although it is possible to use other types such as function objects and function pointers. Regardless of the `Body` type, it has to take an appropriate mare::index object as an argument.

See Also

mare::task(Body&&)

Parameters

r	Range object (1D, 2D or 3D) representing the iteration space.
body	Code that the task will run when the tasks executes.

Returns

task_ptr – Task pointer that points to the new task.

Example 1 Create a 1D range task using a C++11 lambda.

```
//Create a 1D range, spans from [0, 5).
mare::range<1> range_1d(5);

// Create task that prints Hello World 5 times
mare::task_ptr t1 = mare::create_task(range_1d, [](
      mare::index<1>& idx){
    printf("Hello World! - %zu\n", idx[0]);
   });
```

Example 2 Create a 2D range task using a C++11 lambda.

```
//Creates a 2D range, comprising of points
//from cross product [0, 2) x [0, 2).
mare::range<2> range_2d(2, 2);

// Create task that prints Hello World 4 times
mare::task_ptr t1 = mare::create_task(range_2d, [](
      mare::index<2>& idx){
    printf("Hello World! - (%zu, %zu)\n", idx[0], idx[1]);
  });
```

Example 3 Create a 3D range task using a C++11 lambda.

```
//Create a 3D range, comprising of points
//from cross product [0, 2) x [0, 2) x [0, 2)
mare::range<3> range_3d(2, 2, 2);

// Create task that prints Hello World 8 times
mare::task_ptr t1 = mare::create_task(range_2d, [](
      mare::index<3>& idx){
    printf("Hello World! - (%zu, %zu, %zu)\n", idx[0], idx[1], idx[2]);
  });
```

See Also

mare::launch(group_ptr const&, Body&&)

Definition at line 94 of file gputask.hh.

9.2.1.2 template<typename Body > task_ptr mare::create_task (Body && *body*) [inline]

Creates a new task and returns a task_ptr that points to that task.

This template method creates a task and returns a pointer to it. The task executes the `Body` passed as parameter. The preferred type of `<typename Body>` is a C++11 lambda, although it is possible to use other types such as function objects and function pointers. Regardless of the `Body` type, it cannot take any arguments.

Parameters

body	Code that the task will run when the tasks executes.

Returns

task_ptr – Task pointer that points to the new task.

Example 1 Create a task using a C++11 lambda (preferred).

```
1 // Create task that prints Hello World!
2 mare::task_ptr t1 = mare::create_task([]{
3     printf("Hello World!\n");
4   });
```

Example 2 Create a task using a user-defined class.

```
1
2 class user_class{
3 public:
4   user_class(int value)
5     :x(value) {
6   }
7
8   void operator()() {
9     printf("x = %d\n", x);
10   }
11
12   void set_x(int value) {
13     x = value;
14   }
15
16 private:
17   int x;
18 };
22 auto t1 = mare::create_task(user_class(42));
23
25 user_class obj(42);
26 auto t2 = mare::create_task(obj);
27
```

Example 3 Create a task using a function pointer.

```
1 void foo() {
2   printf("Hello World!\n");
3 };
4
5 // Create task that executes foo()
6 auto t = mare::create_task(foo);
```

Warning

Due to limitations in the Visual Studio C++ compiler, this does not work on Visual Studio. You can get around it by using a lambda function:

```
1 void foo() {
2   printf("Hello World!\n");
3 };
4
5 // Create task that executes foo()
6 auto t = mare::create_task([] {
7     foo();
8   });
```

See Also

mare::launch(group_ptr const&, Body&&)

Definition at line 496 of file task.hh.

9.2.1.3 template<typename Body > task_ptr mare::create_task (body_with_attrs< Body > && *attrd_body*) [inline]

Creates a new task with attributes and returns a task_ptr that points to that task.

This method is identical to mare::create_task(Body&&), except it uses a body with attributes, instead of a regular body. The preferred type of body is a C++11 lambda, although it is possible to use other types such as function objects and function pointers. Regardless of its type, the body cannot take any arguments.

Parameters

attrd_body	Body with attributes

Returns

task_ptr – Task pointer that points to the new task.

See Also

mare::create_task(Body&&);
mare::with_attrs(task_attrs const& attrs, Body &&body);

9.2.1.4 template<typename Body > task_ptr mare::create_task (body_with_attrs< Body, Cancel_Handler > && *attrd_body*) [inline]

Creates a new task with attributes and a cancel handler and returns a task_ptr that points to that task.

This method is identical to mare::create_task(Body&&), except it uses a body with attributes and a cancel handler, instead of a regular body. The preferred type of the body is a C++11 lambda, although it is possible to use other types such as function objects and function pointers. Regardless of its type, the body cannot take any arguments.

Parameters

attrd_body	Body with attributes and cancel handler

Returns

task_ptr – Task pointer that points to the new task.

Example

```
1 static std::mutex mutex;
2 static std::condition_variable cv;
3
4 auto attrs = mare::create_task_attrs(mare::attr::blocking);
5
6 auto body = [] {
7   printf("START  blocking task\n");
8   std::unique_lock<std::mutex> lock(mutex);
9   for (;;) {
10     mare::abort_on_cancel();
11     cv.wait(lock);
12   }
13   printf("STOP blocking task\n");
14 };
15
16 auto cancel_handler = [] {
17   printf("CANCEL blocking task\n");
18   std::lock_guard<std::mutex> lock(mutex);
19   cv.notify_all();
20 };
21
22 auto t = mare::create_task(mare::with_attrs(attrs, body, cancel_handler));
23
24 mare::launch(t);
25
```

```
26 // Wait for task to block
27 sleep(2);
28
29 // Cancel task. It will call t's cancel_handler
30 mare::cancel(t);
31
32 //Wait for t to complete
33 mare::wait_for(t);
34
```

See Also

mare::create_task(Body&&);
mare::with_attrs(task_attrs const& attrs, Body &&body, Cancel_Handler &&handler);

9.3 Range and Index

Classes

- class mare::index< DIMS >
- class mare::index< 1 >
- class mare::index< 2 >
- class mare::index< 3 >
- class mare::range< DIMS >
- class mare::range< 1 >
- class mare::range< 2 >
- class mare::range< 3 >

9.3.1 Class Documentation

9.3.1.1 class mare::index

template<size_t DIMS>class mare::index< DIMS >

Methods common to 1D, 2D and 3D index objects are listed here, the value for DIMS can be 1, 2 or 3.

Definition at line 19 of file index.hh.

Public member functions

- index (const internal::index< DIMS > &rhs)
- index< DIMS > & operator= (const index< DIMS > &rhs)
- index< DIMS > & operator+= (const index< DIMS > &rhs)
- index< DIMS > & operator-= (const index< DIMS > &rhs)
- index< DIMS > operator- (const index< DIMS > &rhs)
- index< DIMS > operator+ (const index< DIMS > &rhs)
- bool operator== (const index_base< DIMS > &rhs) const
- bool operator!= (const index_base< DIMS > &rhs) const
- bool operator< (const index_base< DIMS > &rhs) const
- bool operator<= (const index_base< DIMS > &rhs) const
- bool operator> (const index_base< DIMS > &rhs) const
- bool operator>= (const index_base< DIMS > &rhs) const
- size_t & operator[ ] (size_t i)
- const size_t & operator[ ] (size_t i) const
- const std::array< size_t, DIMS > & data () const

9.3.1.1.1 Constructors and Destructors

9.3.1.1.1.1 template<size_t DIMS> mare::index< DIMS >::index (const internal::index< DIMS > & *rhs*) [inline]

Constructs an index object from another index object.

Parameters

rhs	index object to be used for constructing a new object

Definition at line 27 of file index.hh.

9.3.1.1.2 Member Function Documentation

9.3.1.1.2.1 template<size_t DIMS> index<DIMS>& mare::index< DIMS >::operator= (const index< DIMS > & *rhs*) [inline]

Replaces the contents of current index object with an other index object.

Parameters

rhs	index object to be used for replacing the contents of current object

Definition at line 35 of file index.hh.

9.3.1.1.2.2 template<size_t DIMS> index<DIMS>& mare::index< DIMS >::operator+= (const index< DIMS > & *rhs*) [inline]

Sums the corresponding values of current index object and another index object and returns a reference to current index object.

Parameters

rhs	index object to be used for summing with the values of current object

Definition at line 47 of file index.hh.

9.3.1.1.2.3 template<size_t DIMS> index<DIMS>& mare::index< DIMS >::operator-= (const index< DIMS > & *rhs*) [inline]

Subtracts the corresponding values of current index object and another index object and returns a reference to current index object.

Parameters

rhs	index object to be used for subtraction with the values of current object

Definition at line 59 of file index.hh.

9.3.1.1.2.4 template<size_t DIMS> index<DIMS> mare::index< DIMS >::operator- (const index< DIMS > & *rhs*) [inline]

Subtracts the corresponding values of current index object and another index object and returns a new index object.

Parameters

rhs	index object to be used for subtraction with the values of current object

Definition at line 71 of file index.hh.

9.3.1.1.2.5 template<size_t DIMS> index<DIMS> mare::index< DIMS >::operator+ (const index< DIMS > & *rhs*) [inline]

Sums the corresponding values of current index object and another index object and returns new index object.

Parameters

rhs	index object to be used for summing with the values of current object

Definition at line 82 of file index.hh.

9.3.1.1.2.6 template<size_t DIMS> bool mare::index< DIMS >::operator== (const index_base< DIMS > & *rhs*) const

Compares this with another index object.

Parameters

rhs	Reference to index to be compared with this

Returns

TRUE – The two indices have same values
FALSE – The two indices have different values

9.3.1.1.2.7 template<size_t DIMS> bool mare::index< DIMS >::operator!= (const index_base< DIMS > & *rhs*) const

Checks for inequality of this with another index object.

Parameters

rhs	Reference to index to be compared with this

Returns

TRUE – The two indices have different values
FALSE – The two indices have same values

9.3.1.1.2.8 template<size_t DIMS> bool mare::index< DIMS >::operator< (const index_base< DIMS > & *rhs*) const

Checks if this object is less than another index object. Does a lexicographical comparison of two index objects, similar to std::lexicographical_compare().

Parameters

rhs	Reference to index to be compared with this

Returns

TRUE – If this is lexicographically smaller than rhs
FALSE – If this is lexicographically larger or equal to rhs

9.3.1.1.2.9 template<size_t DIMS> bool mare::index< DIMS >::operator<= (const index_base< DIMS > & *rhs*) const

Checks if this object is less than or equal to another index object. Does a lexicographical comparison of two index objects, similar to std::lexicographical_compare().

Parameters

rhs	Reference to index to be compared with this

Returns

TRUE – If this is lexicographically smaller or equal than rhs
FALSE – If this is lexicographically larger than rhs

9.3.1.1.2.10 template<size_t DIMS> bool mare::index< DIMS >::operator> (const index_base< DIMS > & *rhs*) const

Checks if this object is greater than another index object. Does a lexicographical comparison of two index objects, similar to std::lexicographical_compare().

Parameters

rhs	Reference to index to be compared with this

Returns

TRUE – If this is lexicographically larger than rhs
FALSE – If this is lexicographically smaller or equal than rhs

9.3.1.1.2.11 template<size_t DIMS> bool mare::index< DIMS >::operator>= (const index_base< DIMS > & *rhs*) const

Checks if this object is greater or equal to another index object. Does a lexicographical comparison of two index objects, similar to std::lexicographical_compare().

Parameters

rhs	Reference to index to be compared with this

Returns

TRUE – If this is lexicographically larger or equal to rhs
FALSE – If this is lexicographically smaller than rhs

9.3.1.1.2.12 template<size_t DIMS> size_t& mare::index< DIMS >::operator[] (size_t *i*)

Returns a reference to i-th coordinate of index object. No bounds checking is performed.

Parameters

i	Specifies which coordinate to return

Returns

Reference to i-th coordinate of the index object

9.3.1.1.2.13 template<size_t DIMS> const size_t& mare::index< DIMS >::operator[] (size_t *i*) const

Returns a const reference to i-th coordinate of index object. No bounds checking is performed.

Parameters

i	Specifies which coordinate to return

Returns

Const reference to i-th coordinate of the index object

9.3.1.1.2.14 template<size_t DIMS> const std::array<size_t, DIMS>& mare::index< DIMS >::data ( ) const

Returns a reference to an std::array of all coordinates of index object.

Returns

Const reference to an std::array of all coordinates of an index object.

9.3.1.2 class mare::index< 1 >

template<>class mare::index< 1 >

Defines a 1D index object

Definition at line 193 of file index.hh.

Public member functions

- index ()
- index (const std::array< size_t, 1 > &rhs)
- index (size_t i)

9.3.1.2.1 Constructors and Destructors

9.3.1.2.1.1 mare::index< 1 >::index ( )

Constructs a 1D index object, equivalent to index<1>(0).

9.3.1.2.1.2 mare::index< 1 >::index (const std::array< size_t, 1 > & *rhs*)

Creates an index object from std::array.

Parameters

rhs	std::array of size_t with one element

9.3.1.2.1.3 mare::index< 1 >::index (size_t *i*) [explicit]

Creates a 1D index object.

Parameters

i	Value for first coordinate of index object

9.3.1.3 class mare::index< 2 >

template<>class mare::index< 2 >

Defines a 2D index object

Definition at line 218 of file index.hh.

Public member functions

- index ()
- index (const std::array< size_t, 2 > &rhs)
- index (size_t i, size_t j)

9.3.1.3.1 Constructors and Destructors

9.3.1.3.1.1 mare::index< 2 >::index ( )

Constructs a 2D index object, equivalent to index<2>(0, 0).

9.3.1.3.1.2 mare::index< 2 >::index (const std::array< size_t, 2 > & *rhs*)

Creates an index object from std::array.

Parameters

rhs	std::array of size_t with two elements.

9.3.1.3.1.3 mare::index< 2 >::index (size_t *i*, size_t *j*)

Creates a 2D index object.

Parameters

i	Value for first coordinate of index object.
j	Value for second coordinate of index object.

9.3.1.4 class mare::index< 3 >

template<>class mare::index< 3 >

Defines a 3D index object

Definition at line 244 of file index.hh.

Public member functions

- index ()
- index (const std::array< size_t, 3 > &rhs)
- index (size_t i, size_t j, size_t k)

9.3.1.4.1 Constructors and Destructors

9.3.1.4.1.1 mare::index< 3 >::index ( )

Constructs a 3D index object, equivalent to index<3>(0, 0, 0).

9.3.1.4.1.2 mare::index< 3 >::index (const std::array< size_t, 3 > & *rhs*)

Creates an index object from std::array.

Parameters

rhs	std::array of size_t with three elements.

9.3.1.4.1.3 mare::index< 3 >::index (size_t *i*, size_t *j*, size_t *k*)

Creates a 3D index object.

Parameters

i	Value for first coordinate of index object.
j	Value for second coordinate of index object.
k	Value for third coordinate of index object.

9.3.1.5 class mare::range

template<size_t DIMS>class mare::range< DIMS >

Methods common to 1D, 2D and 3D ranges are here.

Definition at line 19 of file range.hh.

9.3.1.6 class mare::range< 1 >

template<>class mare::range< 1 >

Describes a 1D range.

Definition at line 67 of file range.hh.

Public member functions

- range ()
- range (size_t b0, size_t e0)
- range (size_t e0)

9.3.1.6.1 Constructors and Destructors

9.3.1.6.1.1 mare::range< 1 >::range ( ) [inline]

Creates an empty 1D range.

Definition at line 73 of file range.hh.

9.3.1.6.1.2 mare::range< 1 >::range (size_t *b0*, size_t *e0*) [inline]

Creates a 1D range, spans from [b0, e0).

Parameters

b0	Beginning of 1D range.
e0	End of 1D range.

Definition at line 81 of file range.hh.

9.3.1.6.1.3 mare::range< 1 >::range (size_t *e0*) [inline]

Creates a 1D range, spans from [0, e0).

Parameters

e0	End of 1D range.

Definition at line 88 of file range.hh.

9.3.1.7 class mare::range< 2 >

template<>class mare::range< 2 >

Describes a 2D range.

Definition at line 95 of file range.hh.

Public member functions

- range ()
- range (size_t b0, size_t e0, size_t b1, size_t e1)
- range (size_t e0, size_t e1)

9.3.1.7.1 Constructors and Destructors

9.3.1.7.1.1 mare::range< 2 >::range ( ) [inline]

Creates an empty 2D range.

Definition at line 101 of file range.hh.

9.3.1.7.1.2 mare::range< 2 >::range (size_t *b0*, size_t *e0*, size_t *b1*, size_t *e1*) [inline]

Creates a 2D range, comprising of points from cross product [b0, e0) x [b1, e1).

Parameters

b0	First coordinate of beginning of 2D range
e0	First coordinate of end of 2D range
b1	Second coordinate of beginning of 2D range
e1	Second coordinate of end of 2D range

Definition at line 112 of file range.hh.

9.3.1.7.1.3 mare::range< 2 >::range (size_t *e0*, size_t *e1*) [inline]

Creates a 2D range, comprising of points from cross product [0, e0) x [0, e1).

Parameters

e0	First coordinate of end of 2D range
e1	Second coordinate of end of 2D range

Definition at line 122 of file range.hh.

9.3.1.8 class mare::range< 3 >

template<>class mare::range< 3 >

Describes a 3D range.

Definition at line 129 of file range.hh.

Public member functions

- range ()
- range (size_t b0, size_t e0, size_t b1, size_t e1, size_t b2, size_t e2)
- range (size_t e0, size_t e1, size_t e2)

9.3.1.8.1 Constructors and Destructors

9.3.1.8.1.1 mare::range< 3 >::range ( ) [inline]

Creates an empty 3D range.

Definition at line 135 of file range.hh.

9.3.1.8.1.2 mare::range< 3 >::range (size_t *b0*, size_t *e0*, size_t *b1*, size_t *e1*, size_t *b2*, size_t *e2*) [inline]

Creates a 3D range, comprising of points from cross product [b0, e0) x [b1, e1) x [b2, e2)

Parameters

b0	First coordinate of beginning of 3D range
e0	First coordinate of end of 3D range
b1	Second coordinate of beginning of 3D range
e1	Second coordinate of end of 3D range
b2	Third coordinate of beginning of 3D range
e2	Third coordinate of end of 3D range

Definition at line 148 of file range.hh.

9.3.1.8.1.3 mare::range< 3 >::range (size_t *e0*, size_t *e1*, size_t *e2*) [inline]

Creates a 3D range, comprising of points from cross product [0, e0) x [0, e1) x [0, e2)

Parameters

e0	First coordinate of end of 3D range
e1	Second coordinate of end of 3D range
e2	Third coordinate of end of 3D range

Definition at line 159 of file range.hh.

9.4 Execution

Launching into the runtime.

Functions

- void mare::launch (task_ptr const &task)
- void mare::launch (unsafe_task_ptr const &task)
- void mare::launch (group_ptr const &group, task_ptr const &task)
- void mare::launch (group_ptr const &group, unsafe_task_ptr const &task)
- void mare::launch_and_reset (group_ptr const &group, task_ptr &task)
- void mare::launch_and_reset (task_ptr &task)
- template<typename Body >
  void mare::launch (group_ptr const &a_group, Body &&body)
- template<typename Body >
  void mare::launch (group_ptr const &group, body_with_attrs< Body > &&attrd_body)
- template<typename Body , typename Cancel_Handler >
  void mare::launch (group_ptr const &group, body_with_attrs< Body, Cancel_Handler >
  &&attrd_body)

9.4.1 Function Documentation

9.4.1.1 void mare::launch (task_ptr const & *task*) [inline]

Launches task created with mare::create_task(...).

Use this method to launch a task created with mare::create_task(...). In general, you create tasks using mare::create_task(...) if they are part of a DAG. This method informs the MARE runtime that the task is ready to execute as soon as there is an available hardware context ∗and∗ after all its predecessors have executed. Notice that the task is launched into a group only if the task already belongs to a group.

If you do not need to use the task pointer after using this method, consider using mare::launch_and_reset() instead because it is faster.

Parameters

task	Pointer to task.

Exceptions

api_exception	If task pointer is NULL.

Example

```
1 mare::task_ptr t1 = mare::create_task([] {
2     printf("Hello World!\n");
3   });
4
```

```
5 //...
6 // Set up dependencies
7 // ..
8 // t1 is ready, launch it
9
10 mare::launch(t1);
```

See Also

mare::launch_and_reset(group_ptr const& group, task_ptr& task)
mare::create_task(Body&& b)
mare::launch(unsafe_task_ptr const& task)
mare::launch(group_ptr const&, task_ptr const&)

Definition at line 537 of file task.hh.

9.4.1.2 void mare::launch (unsafe_task_ptr const & *task*) [inline]

Launches task created with mare::create_task(...)

See mare::launch(task_ptr const& task).

Parameters

task	Pointer to task to launch.

Exceptions

api_exception	If task pointer is NULL.

Definition at line 551 of file task.hh.

9.4.1.3 void mare::launch (group_ptr const & *group,* task_ptr const & *task*) [inline]

Launches task created with mare::create_task(...) into a group.

Use this method to launch a task created with mare::create_task(...). In general, you create tasks using mare::create_task(...) if they are part of a DAG. This method informs the MARE runtime that the task is ready to execute as soon as there is an available hardware context ∗and∗ after all its predecessors have executed. If you do not need to use the task pointer after using this method, consider using mare::launch_and_reset() instead because it is faster.

Parameters

group	Pointer to group.
task	Pointer to task.

Exceptions

api_exception	If task pointer or group pointer is NULL.

Example

```
1 mare::group_ptr g = mare::create_group();
2 mare::task_ptr t1 = mare::create_task([] { printf("Hello World!\n"); });
3
4 //...
5 // Set up dependencies
6 // ..
7 // t1 is ready, launch it
8
9 mare::launch(g, t1);
```

See Also

launch_and_reset(group_ptr const& group, task_ptr& task)
create_task(Body&& b)
launch(unsafe_task_ptr const& task)

9.4.1.4 void mare::launch (group_ptr const & *group,* unsafe_task_ptr const & *task*) [inline]

Launches task created with mare::create_task(...) into a group

See mare::launch(task_ptr const& task).

Parameters

group	Pointer to group.
task	Pointer to task.

Exceptions

api_exception	If task pointer or group pointer is NULL.

9.4.1.5 void mare::launch_and_reset (group_ptr const & *group,* task_ptr & *task*) [inline]

Launches task created with mare::create_task(...) into a group and resets task pointer

Use this method to launch into a group a task created with mare::create_task(...) and reset the task pointer in a single step. mare::launch_and_reset() is faster than mare::launch(group_ptr const&, task_ptr const&) if `task` is the last pointer to the task. This is because MARE runtime can assume that it is the sole owner of the task and does not need to protect access to it.

In general, you create tasks using mare::create_task(...) if they are part of a DAG. This method informs the MARE runtime that the task is ready to execute as soon as there is an available hardware context ∗and∗ after all its predecessors have executed.

Parameters

group	Pointer to group.
task	Pointer to task.

Exceptions

api_exception	If task pointer or group pointer is NULL.

Example

```
1 mare::group_ptr g = mare::create_group();
2 mare::task_ptr t = mare::create_task([]{ printf("Hello World!\n");});
3
4 //assert will not fire
5 assert(t != nullptr);
6
7 mare::launch_and_reset(g, t);
8
9 //assert will not fire
10 assert(t == nullptr);
```

See Also

mare::launch(unsafe_task_ptr const& task)
mare::launch(group_ptr const&, task_ptr const&)

Definition at line 629 of file task.hh.

9.4.1.6 void mare::launch_and_reset (task_ptr & *task*) [inline]

Launches task created with mare::create_task(...) and resets task pointer

Use this method to launch a task created with mare::create_task(...) and reset the task pointer in a single step. mare::launch_and_reset() is faster than mare::launch(group_ptr const&, task_ptr const&) if `task` is the last pointer to the task. This is because MARE runtime can assume that it is the sole owner of the task and does not need to protect access to it. Notice that the task is launched into a group only if the task already belongs to a group.

In general, you create tasks using mare::create_task(...) if they are part of a DAG. This method informs the MARE runtime that the task is ready to execute as soon as there is an available hardware context ∗and∗ after all its predecessors have executed.

Parameters

task	Pointer to task.

Exceptions

api_exception	If task pointer or group pointer is NULL.

Example

```
1 auto t = mare::create_task([] {
2     printf("Hello World!\n");
3   });
4
5 //assert will not fire
6 assert(t != nullptr);
7
8 mare::launch_and_reset(t);
9
10 //assert will not fire
```

```
11 assert(t == nullptr);
```

See Also

mare::launch(unsafe_task_ptr const& task)
mare::launch(task_ptr const&)

Definition at line 669 of file task.hh.

9.4.1.7 template<typename Body > void mare::launch (group_ptr const & *a_group*, Body && *body*) [inline]

Creates a new task and launches it into a group.

This is the fastest way to create and launch a task* into a group. We recommend you use it as much as possible. However, notice that this method does not return a task_ptr. Therefore, use this method if the new task is not part of a DAG. The MARE runtime will execute the task as soon as there is an available hardware context.

The new task executes the `Body` passed as parameter. The preferred type of `<typename Body>` is a C++11 lambda, although it is possible to use other types such as function objects and function pointers. Regardless of the `Body` type, it cannot take any arguments.

When launching into many groups, remember that group intersection is a somewhat expensive operation. If you need to launch into multiple groups several times, intersect the groups once and launch the tasks into the intersection.

Parameters

a_group	Pointer to group.
body	Task body.

Exceptions

api_exception	If group points to null.

Example 1 Creating and launching tasks into one group.

```
1 // Create group named "Example"
2 mare::group_ptr g = mare::create_group("Example");
3
4 // Create and launch tasks into g
5 for (int i=0; i < 1000; i++)
6   mare::launch(g, []{ printf("Hello World!\n"); });
7
8 // Wait for all the tasks in group g to complete
9 mare::wait_for(g);
```

Example 2 Creating and launching tasks into multiple groups.

```
1 // Create groups
2 mare::group_ptr g1 = mare::create_group("Example 1");
3 mare::group_ptr g2 = mare::create_group("Example 2");
4 mare::group_ptr g12 = g1 & g2;
5
6 // Launch 1000 tasks into g1 and g2
```

```
7 for (int i = 0; i < 1000; i++) {
8   mare::launch(g12, []{ printf("Hello World!\n");});
9  }
```

See Also

mare::create_task(Body&& b)
mare::launch(group_ptr const&, task_ptr const&);

Definition at line 711 of file task.hh.

9.4.1.8 template<typename Body > void mare::launch (group_ptr const & *group,* body_with_attrs< Body > && *attrd_body*) [inline]

Creates a new task with attributes and launches it into a group.

This is the fastest way to create and launch into a group an attributed task∗. This method is identical to mare::launch(group_ptr const&, Body&&), except that it uses a body with attributes, instead of a regular body.

Parameters

group	Pointer to group.
attrd_body	Body with attributes

Exceptions

api_exception	If group points to null

See Also

mare::launch(group_ptr const&, Body&&);

9.4.1.9 template<typename Body , typename Cancel_Handler > void mare::launch (group_ptr const & *group,* body_with_attrs< Body, Cancel_Handler > && *attrd_body*) [inline]

Creates a new task with attributes and a cancel handler and launches it into a group.

This is the fastest way to create and launch into a group an attributed task with a cancel handler∗. This method is identical to mare::launch(group_ptr const&, Body&&), except that it uses a body with attributes, instead of a regular body.

Use this method to create blocking tasks.

Parameters

group	Pointer to group.
attrd_body	Body with attributes and a cancel handler.

Exceptions

api_exception	If group points to null

Example

```
1 static std::mutex mutex;
2 static std::condition_variable cv;
3
4 auto body = [] {
5   printf("START  blocking task\n");
6   std::unique_lock<std::mutex> lock(mutex);
7   for (;;) {
8     mare::abort_on_cancel();
9     cv.wait(lock);
10   }
11   printf("STOP blocking task\n");
12 };
13
14 auto cancel_handler = [] {
15   printf("CANCEL blocking task\n");
16   std::lock_guard<std::mutex> lock(mutex);
17   cv.notify_all();
18 };
19
20 auto g = mare::create_group();
21 mare::launch(g, mare::with_attrs(
      mare::attr::blocking, body, cancel_handler));
22
23 // Wait for task to block
24 sleep(2);
25
26 // Cancel group. It will call the task's cancel_handler
27 mare::cancel(g);
28
29 //Wait for g to complete
30 mare::wait_for(g);
31
```

See Also

mare::launch(group_ptr const&, Body&&);

9.5 Dependencies

Functions

- void mare::after (task_ptr const &pred, task_ptr const &succ)
- void mare::after (task_ptr const &pred, unsafe_task_ptr const &succ)
- void mare::after (unsafe_task_ptr const &pred, task_ptr const &succ)
- void mare::after (unsafe_task_ptr const &pred, unsafe_task_ptr const &succ)
- void mare::before (task_ptr &succ, task_ptr &pred)
- void mare::before (task_ptr &succ, unsafe_task_ptr &pred)
- void mare::before (unsafe_task_ptr &succ, task_ptr &pred)
- void mare::before (unsafe_task_ptr &succ, unsafe_task_ptr &pred)
- mare::task_ptr & operator>> (mare::task_ptr &pred, mare::task_ptr &succ)
- mare::unsafe_task_ptr & operator>> (mare::task_ptr &pred, mare::unsafe_task_ptr &succ)
- mare::task_ptr & operator>> (mare::unsafe_task_ptr &pred, mare::task_ptr &succ)
- mare::unsafe_task_ptr & operator>> (mare::unsafe_task_ptr &pred, mare::unsafe_task_ptr &succ)
- mare::task_ptr & operator<< (mare::task_ptr &succ, mare::task_ptr &pred)
- mare::unsafe_task_ptr & operator<< (mare::task_ptr &succ, mare::unsafe_task_ptr &pred)
- mare::task_ptr & operator<< (mare::unsafe_task_ptr &succ, mare::task_ptr &pred)
- mare::unsafe_task_ptr & operator<< (mare::unsafe_task_ptr &succ, mare::unsafe_task_ptr &pred)

9.5.1 Function Documentation

9.5.1.1 void mare::after (task_ptr const & *pred,* task_ptr const & *succ*) [inline]

Adds the `succ` task to `pred`'s list of successors.

Creates a dependency between two tasks (`pred` and `succ`) so that `succ` starts executing only after `pred` has completed its execution, regardless of how many hardware execution contexts are available in the device. Use this method to create task dependency graphs.

Note: The programmer is responsible for ensuring that there are no cycles in the task graph.

If `succ` has already been launched, mare::after() will throw an api_exception. This is because it makes little sense to add a predecessor to a task that might already be running. On the other hand, if `pred` successfully completed execution, no dependency is created, and mare::after returns immediately. If `pred` was canceled (or if it is canceled in the future), `succ` will be canceled as well due to cancelation propagation.

Parameters

pred	Predecessor task.
succ	Successor task.

Exceptions

api_exception	If `pred` or `succ` are NULL.
api_exception	If `succ` has already been launched.

Example

```
1 #ifdef HAVE_CONFIG_H
2 #include <config.h>
3 #endif
4
5 #include <mare/mare.h>
6 #include <stdio.h>
7
8 using namespace mare;
9
10 int main() {
11
12   // Initialize the MARE runtime.
13   mare::runtime::init();
14
15   // Create group.
16   auto g = mare::create_group("Hello World Group");
17
18   // Create tasks t1 and t2
19   auto t1 = mare::create_task([]{
20       printf ("Hello World! from task t1\n");
21     });
22
23   auto t2 = mare::create_task([]{
24       printf ("Hello World! from task t2\n");
25     });
26
27   // Create dependency between t1 and t2
28   mare::after(t1, t2);
29
30   // Launch both t1 and t2 into g
31   mare::launch(g, t1);
32   mare::launch(g, t2);
33
34   // Wait until t1 and t2 finish.
35   mare::wait_for(g);
36
37   // Shutdown the MARE runtime.
38   mare::runtime::shutdown();
39
40   return 0;
41 }
```

Output:

```
Hello World! from task t1
Hello World! from task t2
```

No other output is possible because of the dependency between t1 and t2.

See Also

mare::before(task_ptr&, task_ptr&).

Definition at line 358 of file task.hh.

9.5.1.2 void mare::after (task_ptr const & *pred,* unsafe_task_ptr const & *succ*) [inline]

Adds the `succ` task to `pred`'s list of successors.

See Also

mare::after(task_ptr&, task_ptr&).

Parameters

pred	Predecessor task.
succ	Successor task.

Definition at line 371 of file task.hh.

9.5.1.3 void mare::after (unsafe_task_ptr const & *pred,* task_ptr const & *succ*) [inline]

Adds the `succ` task to `pred`'s list of successors.

See Also

mare::after(task_ptr&, task_ptr&).

Parameters

pred	Predecessor task.
succ	Successor task.

Definition at line 385 of file task.hh.

9.5.1.4 void mare::after (unsafe_task_ptr const & *pred,* unsafe_task_ptr const & *succ*) [inline]

Adds the `succ` task to `pred`'s list of successors.

See Also

mare::after(task_ptr&, task_ptr&).

Parameters

pred	Predecessor task.
succ	Successor task.

Definition at line 397 of file task.hh.

9.5.1.5 void mare::before (task_ptr & *succ,* task_ptr & *pred*) [inline]

Adds the `succ` task to `pred`'s list of successors.

See Also

mare::after(task_ptr&, task_ptr&).

Parameters

succ	Successor task.
pred	Predecessor task.

Example

```
// Equivalent to after(task1, task2);
before(task2, task1);
```

Definition at line 416 of file task.hh.

9.5.1.6 void mare::before (task_ptr & *succ,* unsafe_task_ptr & *pred*) [inline]

Adds the `succ` task to `pred`'s list of successors.

See Also

mare::before(task_ptr&, task_ptr&.

Parameters

succ	Successor task.
pred	Predecessor task.

Definition at line 429 of file task.hh.

9.5.1.7 void mare::before (unsafe_task_ptr & *succ,* task_ptr & *pred*) [inline]

Adds the `succ` task to `pred`'s list of successors.

See Also

mare::before(task_ptr&, task_ptr&.

Parameters

succ	Successor task.
pred	Predecessor task.

Definition at line 442 of file task.hh.

9.5.1.8 void mare::before (unsafe_task_ptr & *succ,* unsafe_task_ptr & *pred*) [inline]

Adds the `succ` task to `pred`'s list of successors.

See Also

mare::before(task_ptr&, task_ptr&).

Parameters

succ	Successor task.
pred	Predecessor task.

Definition at line 455 of file task.hh.

9.5.1.9 mare::task_ptr& operator>> (mare::task_ptr & *pred,* mare::task_ptr & *succ*) [inline]

Adds the `succ` task to `pred`'s list of successors.

See Also

mare::after(task_ptr&, task_ptr&).

Parameters

pred	Predecessor task.
succ	Successor task.

Example

```
auto t1 = create_task([]{foo();})
auto t2 = create_task([]{bar();})
auto t3 = create_task([]{foo();})
// make t1 execute before t2 before t3
t1 >> t2 >> t3;
```

Definition at line 739 of file task.hh.

9.5.1.10 mare::unsafe_task_ptr& operator>> (mare::task_ptr & *pred,* mare::unsafe_task_ptr & *succ*) [inline]

Adds the `succ` task to `pred`'s list of successors.

See Also

mare::after(task_ptr&, task_ptr&).

Parameters

pred	Predecessor task.
succ	Successor task.

Definition at line 753 of file task.hh.

9.5.1.11 mare::task_ptr& operator>> (mare::unsafe_task_ptr & *pred,* mare::task_ptr & *succ*) [inline]

Adds the `succ` task to `pred`'s list of successors.

See Also

mare::after(task_ptr&, task_ptr&).

Parameters

pred	Predecessor task.
succ	Successor task.

Definition at line 768 of file task.hh.

9.5.1.12 mare::unsafe_task_ptr& operator>> (mare::unsafe_task_ptr & *pred,* mare::unsafe_task_ptr & *succ*) [inline]

Adds the `succ` task to `pred`'s list of successors.

See Also

mare::after(task_ptr&, task_ptr&).

Parameters

pred	Predecessor task.
succ	Successor task.

Definition at line 783 of file task.hh.

9.5.1.13 mare::task_ptr& operator<< (mare::task_ptr & *succ,* mare::task_ptr & *pred*) [inline]

Adds the `succ` task to `pred`'s list of successors.

See Also

mare::before(task_ptr&, task_ptr&).

Parameters

succ	Successor task.
pred	Predecessor task.

Example

```
auto t1 = create_task([]{foo();})
auto t2 = create_task([]{bar();})
auto t3 = create_task([]{foo();})
// make t1 execute before t2 before t3
t3 << t2 << t1;
```

Definition at line 807 of file task.hh.

9.5.1.14 mare::unsafe_task_ptr& operator<< (mare::task_ptr & *succ,* mare::unsafe_task_ptr & *pred*) [inline]

Adds the `succ` task to `pred`'s list of successors.

See Also

mare::before(task_ptr&, task_ptr&).

Parameters

succ	Successor task.
pred	Predecessor task.

Definition at line 821 of file task.hh.

9.5.1.15 mare::task_ptr& operator<< (mare::unsafe_task_ptr & *succ,* mare::task_ptr & *pred*) [inline]

Adds the `succ` task to `pred`'s list of successors.

See Also

mare::before(task_ptr&, task_ptr&).

Parameters

succ	Successor task.
pred	Predecessor task.

Definition at line 836 of file task.hh.

9.5.1.16 mare::unsafe_task_ptr& operator<< (mare::unsafe_task_ptr & *succ,* mare::unsafe_task_ptr & *pred*) [inline]

Adds the `succ` task to `pred`'s list of successors.

See Also

mare::before(task_ptr&, task_ptr&).

Parameters

succ	Successor task.
pred	Predecessor task.

Definition at line 851 of file task.hh.

9.6 Grouping

Functions

- void mare::join_group (group_ptr const &group, task_ptr const &task)
- void mare::join_group (group_ptr const &group, unsafe_task_ptr const &task)

9.6.1 Function Documentation

9.6.1.1 void mare::join_group (group_ptr const & *group,* task_ptr const & *task*) [inline]

Adds a task to group without launching it.

The recommended way to add a task to a group is by using mare::launch(). mare::join_group(group_ptr const& group, task_ptr const& task) allows you to add a task into a group without having to launch it. Use mare::join_group(group_ptr const& group, task_ptr const& task) only when you absolutely need the task to belong to a group, but you are not yet ready to launch the task. For example, perhaps you want to prevent the group from being empty, so you can wait on it somewhere else. This method is slow, and should be employed infrequently. It is possible – though not recommended for performance reasons – to use mare::join_group(group_ptr const&, task_ptr const&) repeatedly to add a task to several groups. Repeatedly adding a task to the same group is not an error.

Regardless of the method you use to add tasks to a group, the following rules always apply:

- Tasks stay in the group until they complete execution. Once a task is added to a group, there is no way to remove it from the group.
- Once a task belonging to multiple groups completes execution, MARE removes it from all the groups to which it belongs.
- Neither completed nor canceled tasks can join groups.
- Tasks cannot be added to a canceled group.

Parameters

group	Pointer to target group.
task	Pointer to task to add to group.

Exceptions

api_exception	If either `group` or `task` points to null

Example

```
1 // Create group g
2 mare::group_ptr g = mare::create_group("Example");
3
4 // Create task t1
5 mare::task_ptr t1 = mare::create_task([]{ printf("Hello World from t1!\n");}
      );
6
```

```
7 // Add t1 to g. Don't do this often
8 mare::join_group(g, t1);
9
10 // Launch t1. Because it belongs to group g,
11 // there is no reason to launch it into the
12 // same group again.
13 mare::launch(t1);
14
15 // t1 no longer needed, reset pointer
16 t1.reset();
17
18 // Wait for tasks in group g to complete
19 mare::wait_for(g);
```

See Also

mare::launch(group_ptr const&, task_ptr const&)

Definition at line 227 of file group.hh.

9.6.1.2 void mare::join_group (group_ptr const & *group,* unsafe_task_ptr const & *task*) [inline]

Adds a task to a group without launching it.

See join_group(group_ptr const& group, task_ptr const& task).

Parameters

group	Pointer to target group.
task	Pointer to task to add to group.

Exceptions

api_exception	If either `group` or `task` points to null.

Definition at line 249 of file group.hh.

9.7 Synchronization

Classes

- class mare::barrier
- class mare::condition_variable
- class mare::mutex
- class mare::futex

Functions

- void mare::wait_for (task_ptr const &task)
- void mare::wait_for (unsafe_task_ptr const &task)

9.7.1 Class Documentation

9.7.1.1 class mare::barrier

Synchronizes multiple tasks at the call to wait().

Synchronizes multiple tasks (or threads) at the point where wait() is called without blocking the MARE scheduler.

Definition at line 25 of file barrier.hh.

Public member functions

- barrier (size_t total)
- **barrier** (barrier &)=delete
- **barrier** (barrier &&)=delete
- barrier & **operator=** (barrier const &)=delete
- barrier & **operator=** (barrier &&)=delete
- void wait ()

9.7.1.1.1 Constructors and Destructors

9.7.1.1.1.1 mare::barrier::barrier (size_t *total*)

Constructs the barrier, given the number of tasks that will wait on it.

Parameters

total	Number of tasks that will wait on the barrier.

9.7.1.1.2 Member Function Documentation

9.7.1.1.2.1 void mare::barrier::wait ( )

Wait on the barrier.

Task may yield to the MARE scheduler.

9.7.1.2 class mare::condition_variable

Provides inter-task communication via wait() and notify().

Provides inter-task communication via wait() and notify() calls without blocking the MARE scheduler.

Definition at line 37 of file condition_variable.hh.

Public member functions

- constexpr condition_variable ()
- **condition_variable** (condition_variable &)=delete
- **condition_variable** (condition_variable &&)=delete
- condition_variable & **operator=** (condition_variable const &)=delete
- void notify_one ()
- void notify_all ()
- void wait (std::unique_lock< mare::mutex > &lock)
- template<class Predicate >
  void wait (std::unique_lock< mare::mutex > &lock, Predicate pred)
- template<class Rep , class Period >
  std::cv_status wait_for (std::unique_lock< mare::mutex > &lock, const std::chrono::duration< Rep, Period > &rel_time)
- template<class Rep , class Period , class Predicate >
  bool wait_for (std::unique_lock< mare::mutex > &lock, const std::chrono::duration< Rep, Period > &rel_time, Predicate pred)
- template<class Clock , class Duration >
  std::cv_status wait_until (std::unique_lock< mare::mutex > &lock, const std::chrono::time_point< Clock, Duration > &timeout_time)
- template<class Clock , class Duration , class Predicate >
  bool wait_until (std::unique_lock< mare::mutex > &lock, const std::chrono::time_point< Clock, Duration > &timeout_time, Predicate pred)

9.7.1.2.1 Constructors and Destructors

9.7.1.2.1.1 constexpr mare::condition_variable::condition_variable ( )

Default constructor.

9.7.1.2.2 Member Function Documentation

9.7.1.2.2.1 void mare::condition_variable::notify_one ( )

Wakes up one waiting task.

9.7.1.2.2.2 void mare::condition_variable::notify_all ( )

Wakes up all waiting tasks.

9.7.1.2.2.3 void mare::condition_variable::wait (std::unique_lock< mare::mutex > & *lock*)

Task waits until woken up by another task.

Task may yield to the MARE scheduler.

Parameters

lock	Mutex associated with the condition_variable. Must be locked prior to calling wait. Mutex will be held by the task when wait returns.

9.7.1.2.2.4 template<class Predicate > void mare::condition_variable::wait (std::unique_lock< mare::mutex > & *lock,* Predicate *pred*)

While the predicate is false, the task waits.

Task may yield to the MARE scheduler.

Parameters

lock	Mutex associated with the condition_variable. Must be locked prior to calling wait. lock will be held by the task when wait returns.
pred	The condition_variable will continue to wait until this Predicate is true.

9.7.1.2.2.5 template<class Rep , class Period > std::cv_status mare::condition_variable::wait_for (std::unique_lock< mare::mutex > & *lock,* const std::chrono::duration< Rep, Period > & *rel_time*)

Task waits until either 1) woken by another task or 2) wait time has been exceeded.

Task may yield to the MARE scheduler.

Parameters

lock	Mutex associated with the condition_variable. Mutex must be locked prior to calling wait_for.
rel_time	Amount of time to wait to be woken up.

Returns

std::cv_status::no_timeout - Task was woken up by another task before rel_time was exceeded. lock will be held by the task when wait_for returns.

std:cv_status::timeout - rel_time was exceeded before the task was woken up.

9.7.1.2.2.6 template<class Rep , class Period , class Predicate > bool mare::condition_variable::wait_for (std::unique_lock< mare::mutex > & *lock,* const std::chrono::duration< Rep, Period > & *rel_time,* Predicate *pred*)

While the predicate is false, task waits until 1) woken by another task or 2) wait time has been exceeded.

Task may yield to the MARE scheduler.

Parameters

lock	Mutex associated with the condition_variable. Mutex must be locked prior to calling wait_for.
rel_time	Amount of time to wait to be woken up.
pred	The condition_variable will continue to wait until this Predicate is true.

Returns

TRUE – Task was woken up and predicate was true before rel_time was exceeded. lock will be held by the task when wait_for returns.

FALSE – Predicate was not true before rel_time was exceeded.

9.7.1.2.2.7 template<class Clock , class Duration > std::cv_status mare::condition_variable::wait_until (std::unique_lock< mare::mutex > & *lock,* const std::chrono::time_point< Clock, Duration > & *timeout_time*)

Task waits until either 1) woken by another task or 2) wait time has been exceeded.

Task may yield to the MARE scheduler.

Parameters

lock	Mutex associated with the condition_variable. Mutex must be locked prior to calling wait_until.
timeout_time	Time to wait until.

Returns

std::cv_status::no_timeout – Task was woken up by another task before timeout_time was exceeded. lock will be held by the task when wait_until returns.

std:cv_status::timeout – timeout_time was exceeded before the task was woken up.

9.7.1.2.2.8 template<class Clock , class Duration , class Predicate > bool mare::condition_variable-::wait_until (std::unique_lock< mare::mutex > & *lock,* const std::chrono::time_point< Clock, Duration > & *timeout_time,* Predicate *pred*)

While the predicate is false, task waits until 1) woken by another task or 2) wait time has been exceeded.

Task may yield to the MARE scheduler.

Parameters

lock	Mutex associated with the condition_variable. Mutex must be locked prior to calling wait_until.
timeout_time	Amount of time to wait to be woken up.
pred	The condition_variable will continue to wait until this Predicate is true.

Returns

TRUE – Task was woken up and predicate was true before timeout_time was exceeded. lock will be held by the task when wait_until returns.

FALSE – Predicate was not true before timeout_time was exceeded.

9.7.1.3 class mare::mutex

Provides basic mutual exclusion.

Provides exclusive access to a single task or thread without blocking the MARE scheduler.

Definition at line 31 of file mutex.hh.

Public member functions

- constexpr mutex ()
- **mutex** (mutex &)=delete
- **mutex** (mutex &&)=delete
- mutex & **operator=** (mutex const &)=delete
- ~mutex ()
- void lock ()
- bool try_lock ()
- void unlock ()

9.7.1.3.1 Constructors and Destructors

9.7.1.3.1.1 constexpr mare::mutex::mutex ( )

Default constructor. Initializes to not locked.

9.7.1.3.1.2 mare::mutex::~mutex ( )

Destructor.

9.7.1.3.2 Member Function Documentation

9.7.1.3.2.1 void mare::mutex::lock ( )

Acquires the mutex, waits if the mutex is not available.

May yield to the MARE scheduler.

9.7.1.3.2.2 bool mare::mutex::try_lock ( )

Attempts to acquire the mutex.

Returns

TRUE – Successfully acquired the mutex.
FALSE – Did not acquire the mutex.

9.7.1.3.2.3 void mare::mutex::unlock ( )

Releases the mutex.

9.7.1.4 class mare::futex

Provides wait/wakeup capabilities for implementing synchronization primitives.

Allows a task to wait for another task to explicitly wake it up. Does not block to the operating system.

Definition at line 71 of file mutex.hh.

Public member functions

- futex ()
- ~futex ()
- **futex** (futex &)=delete
- **futex** (futex &&)=delete
- futex & **operator=** (futex const &)=delete

- void wait ()
- size_t wakeup (size_t num_tasks)

9.7.1.4.1 Constructors and Destructors

9.7.1.4.1.1 mare::futex::futex ( )

Default constructor.

9.7.1.4.1.2 mare::futex::~futex ( )

Destructor.

9.7.1.4.2 Member Function Documentation

9.7.1.4.2.1 void mare::futex::wait ( )

Current task will block to the MARE scheduler until another task calls wakeup.

9.7.1.4.2.2 size_t mare::futex::wakeup (size_t *num_tasks*)

Attempts to wake up requested number of waiting tasks.

Parameters

num_tasks	Number of tasks to attempt to wake up. Will only wake up tasks if enough exist to be woken up. num_tasks == 0 is reserved for broadcast (i.e., will wake up as many tasks as possible).

Returns

Number of tasks that were successfully woken up.

9.7.2 Function Documentation

9.7.2.1 void mare::wait_for (task_ptr const & *task*) [inline]

Waits for task to complete execution.

This method does not return until the task completes its execution. It returns immediately if the task has already finished. If `wait_for(mare::task_ptr)` is called from within a task, MARE context-switches the task and finds another one to run. If called from outside a task (i.e., the main thread), MARE blocks the thread until `wait_for(mare::task_ptr)` returns.

This method is a safe point. Safe points are MARE API methods where the following property holds:

> The thread on which the task executes before the API call might not be the same as the thread on which the task executes after the API call.

Parameters

task	Pointer to target task.

Exceptions

api_exception	If task points to null.

Example

```
1 mare::task_ptr t1 = mare::create_task([] {
2     printf("Hello World!\n");
3   });
4 mare::launch(t1);
5 mare::wait_for(t1); //won't return until t1 has executed
```

See Also

mare::wait_for(group_ptr const& group)

Definition at line 44 of file task.hh.

9.7.2.2 void mare::wait_for (unsafe_task_ptr const & *task*) [inline]

Waits for task to complete execution.

See mare::wait_for(task_ptr const&).

The thread on which the task executes before the API call might not be the same as the thread on which the task executes after the API call.

Parameters

task	Pointer to target task.

Exceptions

api_exception	If task points to null.

Definition at line 63 of file task.hh.

9.8 Cancelation

Functions

- bool mare::canceled (task_ptr const &task)
- bool mare::canceled (unsafe_task_ptr const &task)
- void mare::cancel (task_ptr const &task)
- void mare::cancel (unsafe_task_ptr const &task)
- void mare::abort_on_cancel ()
- void mare::abort_task ()

9.8.1 Function Documentation

9.8.1.1 bool mare::canceled (task_ptr const & *task*) [inline]

Checks whether a task is canceled.

Use the function to know whether a task is canceled. If the task was canceled – via cancelation propagation, mare::cancel(group_ptr const&) or mare::cancel(task_ptr const&) – before it started executing, mare::canceled(task_ptr const&) returns true.

If the task was canceled – via mare::cancel(group_ptr const&)or mare::cancel(group_ptr const&) – while it was executing, then mare::canceled(task_ptr const&) returns true only if the task is not executing any more and it exited via mare::abort_on_cancel() or mare::abort_task().

Finally, if the task completed successfully, mare::canceled(task_ptr const&) always returns false.

Parameters

task	Task pointer.

Returns

TRUE – The task has transitioned to a canceled state.
FALSE – The task has not yet transitioned to a canceled state.

Exceptions

api_exception	If task pointer is NULL.

Example:

```
1 auto t = mare::create_task([]{
2     for (int i = 0; i < 1000; ++i) {
3       mare::abort_on_cancel();
4       printf("Hello World!\n");
5       sleep(1); //sleep for one second
6     }
7   });
8
9 auto g = mare::create_group("example");
10
```

```
11 // It will never fire
12 assert(mare::canceled(t) == false);
13
14 // launch task
15 mare::launch(g, t);
16
17 // It Will never fire
18 assert(mare::canceled(t) == false);
19
20 // Sleep for 10 seconds
21 sleep(10);
22
23 // It Will never fire
24 mare::assert(canceled(t) == false);
25
26 // Cancel both the task and the group
27 mare::cancel(t);
28 mare::cancel(g);
29
30 // It might fire if the task has not executed abort_on_cancel() yet
31 assert(mare::canceled(t) == true);
32
33 // Wait for the task to transition to canceled state
34 mare::wait_for(t);
35
36 // It Will never fire
37 assert(mare::canceled(t) == true);
38
```

Definition at line 102 of file task.hh.

9.8.1.2 bool mare::canceled (unsafe_task_ptr const & *task*) [inline]

Checks whether a task is canceled.

Parameters

task	Task pointer.

Returns

TRUE – The task has transitioned to a canceled state.
FALSE – The task has not yet transitioned to a canceled state.

Exceptions

api_exception	If task pointer is NULL.

See Also

mare::canceled(task_ptr const& task)

Definition at line 122 of file task.hh.

9.8.1.3 void mare::cancel (task_ptr const & *task*) [inline]

Cancels task.

Use `mare::cancel()` to cancel a task and its successors. The effects of `mare::cancel()` depend on the task status:

- If a task is canceled before it launches, it never executes – even if it is launched afterwards. In addition, it propagates the cancelation to the task's successors. This is called "cancelation propagation".
- If a task is canceled after it is launched, but before it starts executing, it will never execute and it will propagate cancelation to its successors.
- If the task is running when someone else calls `mare::cancel()`, it is up to the task to ignore the cancelation request and continue its execution, or to honor the request via `mare::abort_on_cancel()`, which aborts the task's execution and propagates the cancelation to the task's successors.
- Finally, if a task is canceled after it completes its execution (successfully or not), it does not change its status and it does not propagate cancelation.

Parameters

task	Task to cancel.

Exceptions

api_exception	If task pointer is NULL.

Example 1: Canceling a task before launching it:

```
1 auto t1 = mare::create_task([]{
2     assert(false);
3   });
4
5 auto t2 = mare::create_task([]{
6     assert(false);
7   });
8
9 // Create dependencies
10 t1 >> t2;
11
12 // Cancel t1, which propagates cancelation to t2
13 mare::cancel(t1);
14
15 // Launch t2. Does nothing, t2 got canceleded via cancelation propagation
16 mare::launch(t2);
17
18 // Returns immediately, t2 is canceled.
19 mare::wait_for(t2);
```

In the example above, we create two tasks `t1` and `t2` and create a dependency between them. Notice that, if any of the tasks executes, it will raise an assertion. In line 13, we cancel `t1`, which causes `t2` to be canceled as well. In line 16, we launch `t2`, but it does not matter as it will not execute because it was canceled when `t1` propagated its cancelation.

Example 2: Canceling a task after launching it, but before it

executes.

```
1 auto t1 = mare::create_task([]{
2     printf("Hello World from t1!\n");
3   });
4
5 auto t2 = mare::create_task([]{
6     assert(false);
7   });
8
9 auto t3 = mare::create_task([]{
10     assert(false);
11   });
12
13 // Create dependencies
14 t1 >> t2 >> t3;
15
16 // Launch t2. It can't execute yet because
17 // t1 hasn't been launched.
18 mare::launch(t2);
19
20 // Cancel t2, which propagates cancelation to t3
21 mare::cancel(t2);
22
23 // Launch t1. It will execute because nobody
24 // canceled it.
25 mare::launch(t1);
26
27 // Returns after t1 completes execution
28 mare::wait_for(t1);
```

In the example above, we create and chain three tasks `t1`, `t2`, and `t3`. In line 18, we launch `t2`, but it cannot execute because its predecessor has not executed yet. In line 21, we cancel `t2`, which means that it will never execute. Because `t3` is `t2`'s successor, it is also canceled – if `t3` had a successor, it would also be canceled.

Example 3: Canceling a task while it executes.

```
1 mare::task_ptr t = mare::create_task([]{
2     while(1) {
3       mare::abort_on_cancel();
4       printf("Waiting to be canceled.\n");
5       usleep(10);
6     }
7     assert(false); // This will never fire
8   }
9   );
10
11 //Launch t
12 mare::launch(t);
13
14 // Wait for 2 seconds.
15 sleep(2);
16
17 // Cancel task. Returns immediately.
18 mare::cancel(t);
19
20 // Wait for the task.
21 mare::wait_for(t);
22
```

In the example above, task `t`'s will never finish unless it is canceled. `t` is launched in line 12. After launching the task, we block for 2 seconds in line 15 to ensure that `t` is scheduled and prints its messages.

In line 18, we ask MARE to cancel the task, which should be running by now. `mare::cancel()` returns immediately after it marks the task as "pending for cancelation". This means that `t` might still be executing after `mare::cancel(t)` returns. That is why we call `mare::wait_for(t)` in line 21, to ensure that we wait for `t` to complete its execution. Remember: a task does not know whether someone has requested its cancelation unless it calls `mare::abort_on_cancel()` during its execution.

Example 4: Canceling a completed task.

```
1 auto t1 = mare::create_task([]{
2     printf("Hello World from t1!\n");
3   });
4
5 auto t2 = mare::create_task([]{
6     while (1){
7       mare::abort_on_cancel();
8       printf("Hello World from t1!\n");
9       usleep(100);
10     };
11   });
12
13 // Create dependencies
14 t1 >> t2;
15
16 // Launch tasks
17 mare::launch(t1);
18 mare::launch(t2);
19
20 // Wait for t1 to complete
21 mare::wait_for(t1);
22
23 // Cancel t1. Because it has already completed, it does not do
24 // cancelation propagation.
25 mare::cancel(t1);
26
27 // Will never return
28 mare::wait_for(t2);
```

In the example above, we launch `t1` and `t2` after we set up a dependency between them. On line 25, we cancel `t1` after it has completed. By then, `t1` has finished execution (we wait for it in line 21) so `cancel(t1)` has no effect. Thus, nobody cancels `t2` and `wait_for(t2)` in line 28 never returns.

See Also

mare::after(task_ptr const &, task_ptr const &).
mare::abort_on_cancel().
mare::wait_for(task_ptr const&);

Definition at line 214 of file task.hh.

9.8.1.4 void mare::cancel (unsafe_task_ptr const & *task*) [inline]

Cancel tasks

See mare::cancel(task_ptr const& task).

Parameters

task	Task to cancel.

Exceptions

api_exception	If task pointer is NULL.

Definition at line 230 of file task.hh.

9.8.1.5 void mare::abort_on_cancel ( ) [inline]

Aborts execution of calling task if any of its groups is canceled or if someone has canceled it by calling mare::cancel().

MARE uses cooperative multitasking. Therefore, it cannot abort an executing task without help from the task. In MARE, each executing task is responsible for periodically checking whether it should abort. Thus, tasks call mare::abort_on_cancel() to test whether they, or any of the groups to which they belong, have been canceled. If true, mare::abort_on_cancel() does not return. Instead, it thows mare::abort_task_exception, which the MARE runtime catches. The runtime then transitions the task to a canceled state and propagates cancelation to the task's successors, if any.

Because mare::abort_on_cancel() never returns, we recommend that you use use RAII to allocate and deallocate the resources used inside a task. If using RAII in your code is not an option, surround mare::abort_on_cancel() with try – catch, and call throw from within the catch block after the cleanup code.

Exceptions

abort_task_exception	If called from a task that has been canceled via mare::cancel() or that belongs to a canceled group.
api_exception	If called from outside a task.

Example 1

```
1 #ifdef HAVE_CONFIG_H
2 #include <config.h>
3 #endif
4
5 #include <mare/mare.h>
6
7 using namespace mare;
8
9 int main() {
10
11   // Initialize the MARE runtime.
12   mare::runtime::init();
13
14   // Create task
15   auto t = mare::create_task([]{
16       size_t num_iters = 0;
17       while(1)
18       {
19         // Check whether the task needs to stop execution.
20         // Without abort_on_cancel() the task would never
21         // return
22         mare::abort_on_cancel();
23
24         MARE_ALOG("Task has executed %zu iterations!", num_iters);
25         usleep(100);
26         num_iters++;
27       }
28     });
29
30   // Create group g
31   auto g = mare::create_group("example group");
32
```

```
33   // Launch t into g. We don't use t after launch(), so we can
34   // use launch_and_reset
35   mare::launch_and_reset(g, t);
36
37   // Wait for the task to execute a few iterations
38   usleep(10000);
39
40   // Cancel group g, and wait for t to complete
41   mare::cancel(g);
42   mare::wait_for(g);
43
44   // Shutdown the MARE runtime.
45   mare::runtime::shutdown();
46
47   return 0;
48 }
49
```

Output

```
Task has executed 1 iterations!
Task has executed 2 iterations!
...
Task has executed 47 iterations!
```

Example 2

```
1 mare::task_ptr t = mare::create_task([]{
2     while(1) {
3       try{
4         mare::abort_on_cancel();
5       }catch(mare::abort_task_exception const& e) {
6         //..do cleanup
7         throw;
8       }
9       printf("Waiting to be canceled.\n");
10       usleep(10);
11     }
12     assert(false); // This will never fire
13   }
14   );
15
16 //Launch t
17 mare::launch(t);
18
19 // Wait for 2 seconds.
20 sleep(2);
21
22 // Cancel task. Returns immediately.
23 mare::cancel(t);
24
25 // Wait for the task to complete.
26 mare::wait_for(t);
```

Definition at line 277 of file task.hh.

9.8.1.6 void mare::abort_task ( ) [inline]

Aborts execution of calling task.

Use this method from within a running task to immediately abort that task and all its successors. Like mare::abort_on_cancel(), mare::abort_task() never returns. Instead, it throws mare::abort_task_exception, which the MARE runtime catches. The runtime then transitions the task to a canceled state and propagates propagation to the task's successors, if any.

Exceptions

abort_task_exception	If called from a task has been canceled via mare::cancel() or a task that belongs to a canceled group.
api_exception	If called from outside a task.

Example

```
1 auto t1 = mare::create_task([]{
2     int i = 0;
3     while(true) {
4       printf("Hello World %d\n", i);
5       sleep(1);
6       i++;
7       if(i == 10)
8         mare::abort_task();
9     }
10     // This will never fire
11     assert(false);
12   });
13
14 auto t2 = mare::create_task([] {
15     // This will never fire
16     assert(false);
17   });
18
19 t1 >> t2;
20
21 //Launch tasks
22 mare::launch(t1);
23 mare::launch(t2);
24
25 // Wait for t1 to complete.
26 mare::wait_for(t1);
27
28 // Returns immediately, t2 is canceled.
29 mare::wait_for(t2);
```

Output

```
Hello World !
Hello World!
...
Hello World!
```

Definition at line 309 of file task.hh.

9.9 Attributes

Functions

- template<typename Attribute >
  constexpr bool mare::has_attr (task_attrs const &attrs, Attribute const &attr)
- template<typename Attribute >
  constexpr task_attrs mare::remove_attr (task_attrs const &attrs, Attribute const &attr)
- template<typename Attribute >
  const task_attrs mare::add_attr (task_attrs const &attrs, Attribute const &attr)
- MARE_CONSTEXPR task_attrs mare::create_task_attrs ()
- template<typename Attribute , typename... Attributes>
  constexpr task_attrs mare::create_task_attrs (Attribute const &attr1, Attributes const &...attrn)
- MARE_CONSTEXPR bool operator== (mare::task_attrs const &a, mare::task_attrs const &b)
- MARE_CONSTEXPR bool operator!= (mare::task_attrs const &a, mare::task_attrs const &b)

Variables

- static const
  internal::task_attr_blocking mare::attr::blocking
- static const
  internal::task_attr_yield **mare::attr::yield**
- static const
  internal::task_attr_opencl **mare::attr::gpu**

9.9.1 Function Documentation

9.9.1.1 template<typename Attribute > constexpr bool mare::has_attr (task_attrs const & *attrs,* Attribute const & *attr*)

Checks whether a task_attrs object includes a certain attribute.

Parameters

attrs	task_attrs object to query.
attr	Attribute.

Returns

TRUE – task_attrs Includes attribute.
FALSE – task_attrs Does not include attribute.

Example

```
1 ]//Creates task_attrs for a blocking task
2 auto attr1 = mare::create_task_attrs(mare::attr::blocking);
3
```

```
4 //It will never fire
5 assert(mare::has_attr(attr1, mare::attr::blocking));
6
7 //Creates empty task_attrs
8 auto attr2 = mare::create_task_attrs();
9
10 //It will always fire
11 assert(mare::has_attr(attr2, mare::attr::blocking));
```

Definition at line 132 of file attr.hh.

9.9.1.2 template<typename Attribute > constexpr task_attrs mare::remove_attr (task_attrs const & *attrs,* Attribute const & *attr*)

Removes attribute from a task_attr object.

Returns a new task_attr object that includes all the attributes in the original task_attr, except the one specified in the argument list. If the attribute was not in the original task_attr object, the returned object is identical to the original object.

Parameters

attrs	Original task_attrs object.
attr	Attribute.

Returns

task_attrs – Includes all the attributes in attrs except attr. If task_attrs does not include attr, the returned task_attrs is identical to attrs.

Definition at line 156 of file attr.hh.

9.9.1.3 template<typename Attribute > const task_attrs mare::add_attr (task_attrs const & *attrs,* Attribute const & *attr*)

Adds attribute to a mare::task_attrs object.

Returns a new task_attrs object that includes all the attributes in the original task_attr, plus the one specified in the argument list. If the attribute was already in the original task_attrs object, the returned object is identical to the original object.

Parameters

attrs	Original task_attrs object.
attr	Attribute.

Returns

task_attrs – Includes all the attibutes in attrs plus attr. If task_attrs already includes attr, the returned task_attrs is identical to attrs.

Example

```
1 // Creates empty attr
2 mare::task_attrs empty_attrs = mare::create_task_attrs();
3
4 // Adds blocking attribute
5 auto new_attrs1 = mare::add_attr(empty_attrs, mare::attr::blocking);
6
7 // Adds blocking attribute again
8 auto new_attrs2 = mare::add_attr(new_attrs1, mare::attr::blocking);
9
10 // It will never fireo
11 assert (new_attrs1 == new_attrs2);
```

Definition at line 182 of file attr.hh.

9.9.1.4 MARE_CONSTEXPR task_attrs mare::create_task_attrs ( ) [inline]

Creates an empty task_attrs object.

Returns

Empty task_attrs object.

Example

```
1 mare::task_attrs attrs = mare::create_task_attrs();
```

See Also

mare::create_task_attrs(Attribute const&, Attributes const&...)

Definition at line 200 of file attr.hh.

9.9.1.5 template<typename Attribute , typename... Attributes> constexpr task_attrs mare::create_task_attrs (Attribute const & *attr1,* Attributes const &... *attrn*) [inline]

Creates task_attrs object.

Creates a mare::task_attrs object that summarizes all the attributes passed as parameters.

Parameters

attr1	First attribute.
attrn	Last attribute.

Returns

task_attrs object – Includes all the attributes passed as parameters.

Example

```
1 mare::task_attrs my_attr = mare::create_task_attrs(
```

```
      mare::attr::blocking);
2
3 //It will never fire
4 assert(mare::has_attr(my_attr, mare::attr::blocking));
5
```

See Also

mare::create_task_attrs()

9.9.1.6 MARE_CONSTEXPR bool operator== (mare::task_attrs const & *a,* mare::task_attrs const & *b*) [inline]

Checks whether two task_attrs objects are equivalent.

Parameters

a	First task_attrs object.
b	Second task_attrs object.

Returns

TRUE – The two task_attrs objects are equivalent. FALSE – The two task_attrs objects are not equivalent.

Definition at line 591 of file attrobjs.hh.

9.9.1.7 MARE_CONSTEXPR bool operator!= (mare::task_attrs const & *a,* mare::task_attrs const & *b*) [inline]

Checks whether two task_attrs objects are equivalent.

Parameters

a	First task_attrs object.
b	Second task_attrs object.

Returns

TRUE – The two task_attrs objects are not equivalent. FALSE – The two task_attrs objects are equivalent.

Definition at line 606 of file attrobjs.hh.

9.9.2 Variable Documentation

9.9.2.1 const internal::task_attr_blocking mare::attr::blocking [static]

The attr namespace contains the attributes that can be applied to tasks. A blocking task is defined as a task that is dependent on external (non-MARE) synchronization to make guaranteed forward progress. Typically, this includes the completion of I/O requests and other OS syscalls with indefinite runtime, but also busy-waiting. It does not include waiting on MARE tasks or groups using `mare::wait_for`.

There are two problems with blocking tasks. First, once tasks execute, they take over a thread in MARE's thread pool, thus preventing other tasks from executing in the same thread. Because a blocking task spends most of its time blocking on an event, we are essentially wasting one of the threads in the thread pool. When the programmer decorates a task with the `mare::attr::blocking` attribute, MARE ensures that the thread pool does not lose thread on the task.

The second problem involves cancelation. If a blocking task is canceled while it is waiting on an external event, we need to wake the task up, so that it can execute `mare::abort_on_cancel`. This is why blocking tasks require an extra lambda function, called the "cancel handler". The cancel handler of a task is executed only once, and only if the task is running.

Example 1

```
1 static std::mutex mutex;
2 static std::condition_variable cv;
3
4 auto attrs = mare::create_task_attrs(mare::attr::blocking);
5
6 auto body = [] {
7   printf("START  blocking task\n");
8   std::unique_lock<std::mutex> lock(mutex);
9   for (;;) {
10     mare::abort_on_cancel();
11     cv.wait(lock);
12   }
13   printf("STOP blocking task\n");
14 };
15
16 auto cancel_handler = [] {
17   printf("CANCEL blocking task\n");
18   std::lock_guard<std::mutex> lock(mutex);
19   cv.notify_all();
20 };
21
22 auto t = mare::create_task(mare::with_attrs(attrs, body, cancel_handler));
23
24 mare::launch(t);
25
26 // Wait for task to block
27 sleep(2);
28
29 // Cancel task. It will call t's cancel_handler
30 mare::cancel(t);
31
32 //Wait for t to complete
33 mare::wait_for(t);
34
```

In the example above, we create a blocking task using two lambdas, `body` and `cancel_handler`. After launching `t` and sleeping for a couple of seconds, we cancel it (line 24). Most likely, by the time we cancel `t`, it will be waiting on the condition variable. The cancel handler function calls wakes up the task body so that it can abort (line 10)

Example 2

```
1 #ifdef HAVE_CONFIG_H
2 #include <config.h>
3 #endif
4
5 #include <mare/mare.h>
6
7 using namespace mare;
8 using namespace std;
9
10 static atomic<int> task_running(0);
11 static atomic<int> keep_blocking(1);
12
13 int main() {
14
15   // Initialize the MARE runtime.
16   runtime::init();
17
18   // Create attrs object
19   auto attrs = create_task_attrs(attr::blocking);
20
21   // Create blocking task with two lambdas.  The first lambda is
22   // executed as soon as the t gets scheduled. The second one
23   // is executed when t is canceled, so that t stops blocking.
24   auto t = create_task(with_attrs(attrs, [] {
25         task_running = 1;
26         while(keep_blocking){}
27       },
28       []() mutable {
29         printf("Task canceled.  Stop blocking...\n");
30         keep_blocking = 0;
31       }
32       ));
33
34   // Launch long-running task
35   launch(t);
36
37   while(!task_running){};
38   // Sleep for a while
39   usleep(10000);
40   cancel(t);
41
42   // Wait for long-running task to finish
43   wait_for(t);
44
45   // Shutdown the MARE runtime.
46   runtime::shutdown();
47
48   return 0;
49 }
```

Definition at line 53 of file attr.hh.

10 Groups Reference API

Groups represent sets of tasks, which are used to simplify waiting and canceling large numbers of tasks used to implement particular algorithms or phases of the application.

10.1 Group Objects

Classes

- class mare::group_ptr

Functions

- std::string const & mare::group_name (group_ptr const &group)

10.1.1 Class Documentation

10.1.1.1 class mare::group_ptr

An std::shared_ptr-like managed pointer to a group object.

The method void mare::create_group() returns an object of type mare::group_ptr, which is a custom smart pointer to a group object. Groups are reference counted, and they are automatically destroyed when they have no tasks and there are no more mare::group_ptr pointers pointing to them. This means that even if the user has no pointers to the group, MARE will not destroy the group until all its tasks complete.

The group reference counter requires atomic operations. Copying a mare::group_ptr pointer requires an atomic increment, and an atomic decrement when the newly created mare::group_ptr pointer goes out of scope. For best results, minimize the number of times your application copies group_ptr pointers.

The current MARE release does not support mare::unsafe_group_ptr yet, but a future one will.

Warning

> Never use a reference to a mare::group_ptr pointer, as it can cause undefined behavior. This is because some MARE API methods are highly optimized to support the case where there is only one mare::group_ptr pointing to a group. Unfortunately, MARE cannot always prevent the programmer from taking a reference to a mare::group_ptr object, so it is up to the programmer to ensure that this does not happen.

See Also

> mare::task_ptr

Definition at line 303 of file mareptrs.hh.

Public member functions

- constexpr group_ptr ()
- constexpr group_ptr (std::nullptr_t)
- group_ptr (group_ptr const &other)
- group_ptr (group_ptr &&other)
- ~group_ptr ()
- group_ptr & operator= (group_ptr const &other)
- group_ptr & operator= (std::nullptr_t)

- group_ptr & operator= (group_ptr &&other)
- void swap (group_ptr &other)
- void reset ()
- operator bool () const
- size_t use_count () const
- bool unique () const

10.1.1.1.1 Constructors and Destructors

10.1.1.1.1.1 constexpr mare::group_ptr::group_ptr ( )

Default constructor. Constructs null pointer.

10.1.1.1.1.2 constexpr mare::group_ptr::group_ptr (std::nullptr_t)

Constructs null pointer.

10.1.1.1.1.3 mare::group_ptr::group_ptr (group_ptr const & *other*)

Copy constructor. Cosntructs pointer to the same location as other and increases reference count to group.

Parameters

other	Another pointer used to copy-construct the pointer.

10.1.1.1.1.4 mare::group_ptr::group_ptr (group_ptr && *other*)

Move constructor. Constructs pointer to the same location as other using move semantics. Does not increase the reference count to the group.

Parameters

other	Another pointer used to copy-construct the pointer.

10.1.1.1.1.5 mare::group_ptr::~group_ptr ( )

Destructor. It will destroy the target group if it is the last group_ptr pointing to the same group and has not yet been launched.

10.1.1.1.2 Member Function Documentation

10.1.1.1.2.1 group_ptr& mare::group_ptr::operator= (group_ptr const & *other*)

Assigns address to pointer and increases reference counting to it.

Parameters

other	Pointer used as source for the assignment.

Returns

∗this

10.1.1.1.2.2 group_ptr& mare::group_ptr::operator= (std::nullptr_t)

Resets the pointer.

Returns

∗this

10.1.1.1.2.3 group_ptr& mare::group_ptr::operator= (group_ptr && *other*)

Assigns address to pointer using move semantics.

Parameters

other	Pointer used as source for the assignment.

Returns

∗this

10.1.1.1.2.4 void mare::group_ptr::swap (group_ptr & *other*)

Swaps the location the pointer points to.

Parameters

other	Pointer used to exchange targets with.

10.1.1.1.2.5 void mare::group_ptr::reset ( )

Resets the pointer.

10.1.1.1.2.6 mare::group_ptr::operator bool ( ) const [explicit]

Checks whether the pointer is not a nullptr.

Returns

TRUE – The pointer is not a nullptr.
FALSE – The pointer is a nullptr.

10.1.1.1.2.7 size_t mare::group_ptr::use_count ( ) const

Counts number of group_ptr pointing to same group.

This is an expensive operation as it requires an atomic operation. Use it sparingly.

Returns

size_t – Number of group_ptr pointing to the same group.

10.1.1.1.2.8 bool mare::group_ptr::unique ( ) const

Checks whether there is only one pointer pointing to the group.

Returns

TRUE – There is only one pointer pointing to the group.

FALSE – There is more than one pointer pointing to the group, or the pointer is null.

10.1.2 Function Documentation

10.1.2.1 std::string const& mare::group_name (group_ptr const & *group*) [inline]

Returns the name of the group.

Parameters

group	Group.

Returns

std::string containing the name of the group.

Exceptions

api_exception	If group pointer is NULL.

Definition at line 24 of file group.hh.

10.2 Creation

Functions

- group_ptr mare::create_group (std::string const &name)
- group_ptr mare::create_group (const char *name)
- group_ptr mare::create_group ()
- group_ptr mare::intersect (group_ptr const &a, group_ptr const &b)

10.2.1 Function Documentation

10.2.1.1 group_ptr mare::create_group (std::string const & *name*) [inline]

Creates a named group and returns a group_ptr that points to the group.

Groups can be named to facilitate debugging. Two or more groups can share the same name.

Parameters

name	Group name.

Returns

group_ptr – Points to the new group.

Example

```
1 // Create group named "Example"
2 mare::group_ptr g = mare::create_group("Example");
```

See Also

mare::create_group()

Definition at line 50 of file group.hh.

10.2.1.2 group_ptr mare::create_group (const char * *name*) [inline]

Creates a named (const char*) group and returns a group_ptr that points to the group.

Groups can be named to facilitate debugging. Two or more groups can share the same name.

Parameters

name	Group name.

Returns

group_ptr – Points to the new group.

Example

```
1 // Create group named "Example"
2 mare::group_ptr g = mare::create_group("Example");
```

See Also

mare::create_group()

Definition at line 77 of file group.hh.

10.2.1.3 group_ptr mare::create_group ( ) [inline]

Creates a group and returns a group_ptr that points to the group.

In the current MARE release, there can be only 32 groups at a given time. Trying to create a 33rd group aborts the application.

Returns

group_ptr – Points to the new group.

Example

```
1 // Create group
2 mare::group_ptr g = mare::create_group();
```

See Also

mare::create_group(std::string const&)

Definition at line 100 of file group.hh.

10.2.1.4 group_ptr mare::intersect (group_ptr const & *a,* group_ptr const & *b*) [inline]

Returns a pointer to a group that represents the intersection of two groups.

Some applications require that tasks join more than one group. It is possible – though not recommended for performance reasons – to use mare::launch(group_ptr const&, task_ptr const&) or mare::join_group(group_ptr const&, task_ptr const&) repeatedly to add a task to several groups. Instead, use mare::intersect(group_ptr const&, group_ptr const&) to create a new group that represents the intersection of all the groups where the tasks need to launch. Again, this method is more performant than repeatedly launching the same task into different groups.

Launching a task into the intersection group also simultaneously launches it into all the groups that are part of the intersection.

Note: Intersection groups do not count towards the 31 maximum number of simultaneous groups in the application. Remember that group intersection is a somewhat expensive operation. If you need to intersect the same groups repeatedly, intersect once, and keep the pointer to the group intersection.

Consecutive calls to `mare::intersect` with the same groups' pointer as arguments, return a pointer to the same group. Group intersection is a commutative operation.

You can use the `&` operator instead of `mare::intersect`.

Parameters

a	Pointer to the first group.
b	Pointer to the second group.

Returns

group_ptr – Points to a group that represents the intersection of `a` and `b`.

Example

```
1 // Create groups
2
3 mare::group_ptr g1 = mare::create_group("Group 1");
4 mare::group_ptr g2 = mare::create_group("Group 2");
5
6 for(int i=0; i<100; i++)
7   mare::launch(g1, []{
8       //... Do something
9     });
10
11 for(int i=0; i<200; i++)
12   mare::launch(g2, []{
13       //... Do something
14     });
15
16 mare::group_ptr g12 = mare::intersect(g1, g2);
17
18 // Returns immediately. g12 is empty
19 mare::wait_for(g12);
20
21 // Return only after tasks in g1 and g2 complete
22 mare::wait_for(g1);
23 mare::wait_for(g2);
24
25 mare::task_ptr t = mare::create_task([] {
26     //... Calculate the Ultimate Question of Life, the Universe, and Everything
27     printf("42\n");
28   });
29
30 mare::launch(g12, t);
31
32 // All will return after the task prints 42
33 mare::wait_for(g1);
34 mare::wait_for(g2);
35
```

The previous code snippet shows an application with two groups, `g1` and `g2`, each with 100 and 200 tasks, respectively. In line 16, we intersect `g1` and `g2` into `g12`. The returned pointer, `g12`, points to an empty group because no task yet belongs to both `g1` and `g2`. Therefore, `mare::wait_for(g12)` in line 19 returns immediately. The `wait_for` calls in lines 22 and 23 return only once their tasks complete. In line 30, we launch `t` into `g12`. The `wait_for` calls in lines 31 and 32 return only after `t` completes execution because `t` belongs to both `t1` and `t2` (and, of course, `g12`).

See Also

mare::join_group(group_ptr const&, task_ptr const&)
mare::launch(group_ptr const&, task_ptr const&)

Definition at line 160 of file group.hh.

10.3 Waiting

Functions

- void mare::wait_for (group_ptr const &group)
- void **mare::spin_wait_for** (group_ptr const &group)

10.3.1 Function Documentation

10.3.1.1 void mare::wait_for (group_ptr const & *group*) [inline]

Waits until all the tasks in it have completed execution or have been canceled.

This function does not return until all tasks in the group complete their execution. If new tasks are added to the group while wait_for is waiting, this function does not return until all those new tasks also complete. If `mare::wait_for(mare::group_ptr)` is called from within a task, MARE context switches the task and finds another task to run. If called from outside a task, this function blocks the calling thread until it returns.

Waiting for a group intersection means that MARE returns once the tasks in the intersection group have completed or executed.

This method is a safe point. Safe points are MARE API methods where the following property holds:

> The thread on which the task executes before the API call might not be the same as the thread on which the task executes after the API call.

Parameters

group	Pointer to group.

Exceptions

api_exception	If group points to null.

Example 1

```
1 mare::group_ptr g = mare::create_group("Example");
2
3 // Launch 1000 tasks into g
4 for (int i = 0; i < 1000; i++){
5   mare::launch(g, []{ printf("Hello World!\n");});
6  }
7
8 // Wait for tasks to complete
9 mare::wait_for(g);
10
```

Example 2

```
1 // Create groups
2 mare::group_ptr g1 = mare::create_group("Example 1");
3 mare::group_ptr g2 = mare::create_group("Example 2");
4 mare::group_ptr g12 = g1 & g2;
5
6 // Create and launch two tasks that never end
7 mare::launch(g1, []{
```

```
8     while(1) {}
9   });
10
11 mare::launch(g2, []{
12     while(1) {}
13   });
14
15 // Returns immediately because there are no
16 // tasks that belong to both g1 and g2
17 mare::wait_for(g12);
18
19 // Never returns
20 mare::wait_for(g1);
21
```

See Also

wait_for(task_ptr const& task)

Definition at line 298 of file group.hh.

10.4 Cancelation

Functions

- void mare::cancel (group_ptr const &group)
- bool mare::canceled (group_ptr const &group)

10.4.1 Function Documentation

10.4.1.1 void mare::cancel (group_ptr const & *group*) [inline]

Cancels all tasks in a group.

Canceling a group means that:

- The group tasks that have not started execution will never execute.
- The group tasks that are executing will be canceled only when they call `mare::abort_on_cancel`. If any of these executing tasks is a blocking task, MARE executes its cancel handler if they had not executed it before.
- Any tasks added to the group after the group is canceled are also canceled.

Once a group is canceled, it cannot revert to a non-canceled state. mare::cancel(group_ptr const&) returns immediately. Call mare::wait_for(group_ptr const&) afterwards to wait for all the running tasks to be completed and for all the non-running tasks to be canceled.

Parameters

group	Group to cancel.

Example 1

```
1 // Create group
2 auto g = mare::create_group("example");
3
4 // Create lambda for task body.
5 auto body = [] () {
6   for (int i = 0; i < 5; ++i) {
7     mare::abort_on_cancel();
8     printf("Hello world %d\n", i);
9     sleep(1);
10   }
11 };
12
13 // Launch many tasks
14 for (int j = 0; j < 10000; ++j) {
15   mare::launch(g, body);
16  }
17 // Wait for a little bit, to give a
18 // few tasks time to complete their
19 // execution
20 sleep(10);
21
22 // Cancel group and wait for the
23 // running tasks to complete
24 mare::cancel(g);
25 mare::wait_for(g);
```

In the example above, we launch 10000 tasks and then sleep for some time, so that a few of those 10000 tasks are done, a few others are executing, and the majority is waiting to be executed. In line 24, we cancel the group, which means that next time the running tasks execute `mare::abort_on_cancel()`, they will see that their group has been canceled and will abort. `wait_for(g)` will not return before the running tasks end their execution – either because they call `mare::abort_on_cancel()`, or because they finish writing all the messages.

Example 2

```
1 #ifdef HAVE_CONFIG_H
2 #include <config.h>
3 #endif
4
5 #include <atomic>
6 #include <mare/mare.h>
7
8 using namespace std;
9
10 // Counts the number of tasks that execute before the group gets
11 // canceled
12 atomic<size_t> counter;
13
14 int main() {
15
16   // Initialize the MARE runtime.
17   mare::runtime::init();
18   auto group = mare::create_group();
19
20   // Create 2000 tasks that print Hello World!.
21   for (int i = 0; i < 2000; i++){
22     mare::launch(group, [] {
23         counter++;
24         usleep(7);
25       });
26   }
27
28   // Cancel group
29   mare::cancel(group);
30
31   // Wait for group to cancel
32   mare::wait_for(group);
33   MARE_ALOG("wait_for returned after %zu tasks executed", counter.load());
34
35   // Shutdown the MARE runtime.
36   mare::runtime::shutdown();
37
38   return 0;
39 }
40
```

Output

```
wait_for returned after 87 tasks executed.
```

See Also

mare::cancel(task_ptr const&)
mare::cancel(unsafe_task_ptr const&task)

Definition at line 376 of file group.hh.

10.4.1.2 bool mare::canceled (group_ptr const & *group*) `[inline]`

Checks whether the group is canceled.

Returns true if the group has been canceled; otherwise, returns false.

Parameters

group	Group pointer.

Returns

TRUE – The group is canceled.
FALSE – The group is not canceled.

Exceptions

api_exception	If group pointer is NULL.

See Also

mare::cancel(group_ptr const& group)

Definition at line 401 of file group.hh.

11 Power Management API

The MARE power management API provides programmers with a high-level interface to express desired power settings.

The MARE power API provides two operational modes, static and dynamic.

In static mode the programmer picks a predefined operational mode without any intervention from the MARE runtime.

In dynamic mode, the runtime will regulate the cores and frequencies according to a metric provided by the programmer. This mode requires that the program has a repetitive work pattern with a main loop. The code should include a call to set_goal and end_regulation at the prologue and epilogue of the loop and one call to regulate at the end of the body. The regulation has been tunned to provide a performance close to the maximum with energy savings. The programmer should pick a value in the metric close to the maximum possible.

Modes are exclusive. If an application starts an static mode and then switches to dynamic, the static mode request is automatically cancelled.

All API methods provide hints to the system and do not guarantee that the device state will be the requested state. Depending on the rest of the running applications, temperature, and other factors the number of cores and frequencies may differ from those requested.

Using the power management API requires including the following header file:

```
#include <mare/powermanager.hh>
```

Support for this API is only included on some Qualcomm devices. For other devices, the methods do not perform any operation.

11.1 Static Power Management

Enumerations

- enum mare::power::mode { mare::power::mode::normal, mare::power::mode::efficient, mare::power::mode::perf_burst, mare::power::mode::saver }

Functions

- void mare::power::request_mode (const mode m, const std::chrono::milliseconds &duration=std::chrono::milliseconds(0))

11.1.1 Enumeration Type Documentation

11.1.1.1 enum mare::power::mode [strong]

Enumerator

normal Device default power settings

efficient Enable all available cores and reduce the maximum frequency. Useful for parallel applications with work bursts

perf_burst Enable all cores at the maximum frequency for a limited amount of time in the order of seconds. Actual time will depends on the device, the rest of running applications and the environment

saver Enable half total cores with maximum frequency limited to the median of the available frequencies

Definition at line 14 of file powermanager.hh.

11.1.2 Function Documentation

11.1.2.1 void mare::power::request_mode (const mode *m*, const std::chrono::milliseconds & *duration* = `std::chrono::milliseconds(0)`)

Set the system in one of the predefined performance/power modes

Depending on the load and the system status, requests can be ignored. For example, in case of thermal throttling. The programmer can specify the duration of the mode in milliseconds or can pass a 0 in the duration parameter. In this case, the mode remains active until: the application finishes, a call to set_static_mode(mode::normal), a call to set_goal()

Parameters

m	performance/power mode. Possible values: NONE, efficient, perf_burst, and saver
duration	Time in milliseconds that the mode should be enabled. perf_burst mode has a limited maximum duration depending on the device and environment. If duration is lower than 0, the function returns immediately.

Example 1 Setting the efficent mode.

```
1 // Initialize the MARE runtime.
2 mare::runtime::init();
3
4 // Request the efficient mode
5 mare::power::request_mode(mare::power::mode::efficient
      );
6
7 // Main body of the application
8 // ...
9
10 // Optional mode release
11 mare::power::request_mode(mare::power::mode::normal);
12 // Shutdown the MARE runtime
13 mare::runtime::shutdown();
```

11.2 Dynamic Power Management

Functions

- void mare::power::set_goal (const float desired, const float tolerance=0.0)
- void mare::power::clear_goal ()
- void mare::power::regulate (const float measured)

11.2.1 Function Documentation

11.2.1.1 void mare::power::set_goal (const float *desired,* const float *tolerance = 0.0*)

Start the automatic performance/power regulation mode.

Once the regulation process starts, subsequent calls to this function are silently ignored.

Parameters

desired	User defined performance metric value to stabilize the runtime. The metric unit has to match with the unit of the measured parameter of regulate. For better results please use values ranging between ten and hundred.
tolerance	Percentage of allowed error between the desired and the measured values. If the error is less than the tolerance, the system remains in the same state. For example, in the quotient between measured and desired is 0.95 and tolerance is 0.1, state will not change.

When the tolerance is met, the system will remain in the current state. If tolerance is 0.0, the runtime tries to adjust the system at every step

11.2.1.2 void mare::power::clear_goal ( )

Terminate the regulation process and return the system to normal mode

11.2.1.3 void mare::power::regulate (const float *measured*)

Regulate the system to achieve the desired performance level

Parameters

measured	Performance metric value observed during the last iteration

Example 2 Dynamic regulation.

```
1 // Initialize the MARE runtime.
2 mare::runtime::init();
3
4 // set the goal
5 float desired = 30.0;
6 float tolerance = 0.1;
7 mare::power::set_goal(desired, tolerance);
```

```
8
9 // Main body of the application
10 while (work_to_do) {
11
12   // get the performance of the application
13   float measured = get_performance();
14
15   // Inform the MARE runtime
16   mare::power::regulate(measured);
17 }
18 // stop the regulation
19 mare::power::clear_goal();
20
21 // Shutdown the MARE runtime
22 mare::runtime::shutdown();
```

12 Heterogeneous Compute API

The MARE heterogeneous compute API provides programmers with a high-level interface to exploit the heterogeneity present in modern heterogeneous systems on a chip (SoC).

Currently the users can exploit the power of Adreno Graphics Processing Units (GPU) using this API.

12.1 Buffers

Classes

- class mare::buffer< ElementType >

Functions

- template<typename T >
  buffer_ptr< T > mare::create_buffer (size_t size)
- template<typename T >
  buffer_ptr< T > mare::create_buffer (T *ptr, size_t size)

12.1.1 Class Documentation

12.1.1.1 class mare::buffer

template<typename ElementType>class mare::buffer< ElementType >

Definition at line 26 of file buffer.hh.

Public member functions

- buffer (ElementType *ptr, size_t sz)
- buffer (size_t sz)
- void sync ()
- size_t size ()
- ElementType & operator[ ] (size_t index)
- ElementType * data ()

12.1.1.1.1 Member Function Documentation

12.1.1.1.1.1 template<typename ElementType> void mare::buffer< ElementType >::sync ( ) [inline]

Copies the data back to backing store which can be either user provided or runtime allocated or it can be host-accessible device memory. If this method is called when the kernel is still running, then the calling thread is blocked until the kernel completes execution and data is copied back.

Definition at line 40 of file buffer.hh.

12.1.1.1.1.2 template<typename ElementType> size_t mare::buffer< ElementType >::size ( ) [inline]

Returns the size in elements of the buffer.

Returns

Returns the size in elements of the buffer.

12.1.1.1.1.3 template<typename ElementType> ElementType& mare::buffer< ElementType >::operator[] (size_t *index*)

Access an element of buffer. If the buffer is used in a kernel launch, calling this function would block until the kernel completes and data is moved back to host backing store which can be either provided by the user during buffer creation or is created by MARE runtime or it simply refers to host-accessible device memory.

Parameters

index	Index of the element to access.

Returns

Returns the element stored at index.

12.1.1.1.1.4 template<typename ElementType> ElementType* mare::buffer< ElementType >::data ( )

Returns pointer to the host backing store. This is equivalent to calling operator[] with index equals 0.

12.1.2 Function Documentation

12.1.2.1 template<typename T > buffer_ptr<T> mare::create_buffer (size_t *size*) [inline]

Creates a buffer of given size with no backing store on host. If this buffer is used on the host before being used on a device, host memory is allocated and used as the backing store, so in this case the buffer behaves as though it was created with user provided data.

See Also

mare::create_buffer(T* ptr, size_t size).

If this buffer is used on a device and is never accessed on the host, no host memory will be allocated for the buffer. This may be useful in case of buffers used across consecutive kernel launches.

If this buffer is used on a device and then accessed on the host, then no new host memory will be allocated since the buffer may already be allocated in host-accessible device memory when used on the device.

Parameters

size	Size in number of elements of a given Type.

Exceptions

api_exception	if size is 0.

Example

```
1 // Create a buffer with 1k elements.
2 auto buf_a = mare::create_buffer<float>(1024);
3
```

Returns

mare_buffer_ptr<T> – Buffer pointer that points to new buffer.

Definition at line 91 of file buffer.hh.

12.1.2.2 template<typename T > buffer_ptr<T> mare::create_buffer (T * *ptr,* size_t *size*) [inline]

Creates a buffer of size elements with data provided in ptr as the backing store. Modifying the contents using ptr after this call is undefined, since it will be used as a backing store for syncing data to and from the device where this buffer is used.

Parameters

ptr	Pointer to host data to be used as backing store.
size	Size in number of elements, it's assumed that the application has allocated sz * sizeof(ElementType) bytes ptr is pointing to it.

Exceptions

api_exception	if size is 0.

Example

```
// Use the aligned allocator to allocate host data
template<class T, size_t Alignment>
using aligned_vector = std::vector<T, mare::aligned_allocator<T, Alignment> >;
template<class T>
using page_aligned_vector = aligned_vector<T, 4096>;

// allocate host memory for 1k floats.
page_aligned_vector<float> host_ptr(1024);

// Create a buffer from existing host data
auto buf_a = mare::create_buffer(host_ptr, 1024);
```

Returns

mare_buffer_ptr<T> – Buffer pointer that points to new buffer.

Definition at line 134 of file buffer.hh.

12.2 GPU Task Creation

Functions

- template<size_t DIMS, typename Body >
  task_ptr mare::create_ndrange_task (mare::range< DIMS > &r, body_with_attrs_gpu< Body, KernelPtr, Kargs...> &&attrd_body)

12.2.1 Function Documentation

12.2.1.1 template<size_t DIMS, typename Body > task_ptr mare::create_ndrange_task (mare::range< DIMS > & *r,* body_with_attrs_gpu< Body, KernelPtr, Kargs...> && *attrd_body*) [inline]

Creates a new task and returns a task_ptr that points to that task.

This template method creates a task and returns a pointer to it. The task executes the `Body` passed as parameter for each point in the iteration space of mare::range specified by the first argument. The preferred type of `<typename Body>` is a C++11 lambda, although it is possible to use other types such as function objects and function pointers. mare::range object passed is used as the global size for the OpenCL C kernel.

Parameters

r	Range object (1D, 2D or 3D) representing the iteration space.
attrd_body	Body with attributes, kernel object created using `mare::create_kernel` and actual kernel arguments.

Returns

task_ptr – Task pointer that points to the new task.

Example 1 Create an ndrange task which runs on the GPU.

```
// Use the aligned allocator to allocate host data
template<class T, size_t Alignment>
using aligned_vector = std::vector<T, mare::aligned_allocator<T, Alignment> >;
template<class T>
using page_aligned_vector = aligned_vector<T, 4096>;

// allocate host memory for 1k floats.
page_aligned_vector<float> host_ptr(1024);

// Create a buffer from existing host data
auto buf_a = mare::create_buffer(host_ptr, 1024);

//Create a gpu task attribute
auto attrs = mare::create_task_attrs(mare::attr::gpu)

//Create a kernel string containing OpenCL C code.
//The code below shows an empty OpenCL C kernel.
#define OCL_KERNEL(name, k) std::string const name##_string = #k

//Empty OpenCL C kernel, does nothing.
OCL_KERNEL(gpu_empty_fn,
  __kernel void empty(__global float* A, unsigned int size) {
  });
```

```
//Create a kernel object which has the same signature as the OpenCL C kernel.
auto gpu_empty_fn = mare::create_kernel<mare::buffer_ptr<float>,
                                        unsigned int>(gpu_empty_fn_string,
                                                      kernel_name);

//Create a 1D range, spans from [0, 5).
mare::range<1> range_1d(5);

// Create a 1D range task which runs on the gpu.
auto gpu_task = create_ndrange_task(range_1d,
                                    with_attrs(attrs, gpu_empty_fn,
                                               buf_a, size));
```

See Also

mare::launch(group_ptr const&, Body&&)
mare::create_ndrange_task(const range<DIMS>&, Body&&);
mare::create_task(Body&&)

12.3 GPU Patterns

Functions

- template<size_t DIMS, typename Body >
  void mare::pfor_each (mare::range< DIMS > &r, body_with_attrs_gpu< Body, KernelPtr, Kargs...> &&attrd_body)

12.3.1 Function Documentation

12.3.1.1 template<size_t DIMS, typename Body > void mare::pfor_each (mare::range< DIMS > & *r,* body_with_attrs_gpu< Body, KernelPtr, Kargs...> && *attrd_body*)

Parallel version of `std::for_each` that runs on the GPU.

Launches a task on the GPU with global size for the OpenCL C kernel as defined by `mare::range` parameter.

Note: This function returns only after the kernel completes execution.

In addition, the call to this function cannot be canceled by canceling the group passed as argument.

Parameters

r	Range object (1D, 2D or 3D) representing the iteration space.
attrd_body	Body with attributes, kernel object created using `mare::create_kernel` and actual kernel arguments.

Example 1 Create a pfor_each which runs on the GPU.

```
1 #ifdef HAVE_CONFIG_H
2 #include <config.h>
3 #endif
4
5 #include <mare/mare.h>
6 #include <mare/alignedallocator.hh>
7 #include <mare/patterns.hh>
8
9 //Create input vectors using an aligned allocator.
10 //The example below creates a page algined vectors. Although
11 //this step is not mandatory, an aligned vector may avoid
12 //additional copies by runtime.
13 template<class T, size_t Alignment>
14 using aligned_vector = std::vector<T, mare::aligned_allocator<T, Alignment> >;
15 template<class T>
16 using page_aligned_vector = aligned_vector<T, 4096>;
17
18 //Create a string containing OpenCL C kernel code.
19 #define OCL_KERNEL(name, k) std::string const name##_string = #k
20
21 OCL_KERNEL(vadd_kernel,
22   __kernel void vadd(__global float* A, __global float* B, __global float* C,
23                      unsigned int size) {
24     unsigned int i = get_global_id(0);
25     if(i < size)
26        C[i] = A[i] + B[i];
27   });
28
```

```
29 int
30 main(void)
31 {
32   //Initialize the MARE runtime.
33   mare::runtime::init();
34
35   //Create input vectors
36   page_aligned_vector<float> a(1024);
37   page_aligned_vector<float> b(a.size());
38
39   //Initialize the input vectors
40   for (size_t i = 0; i < a.size(); ++i) {
41     a[i] = i;
42     b[i] = a.size() - i;
43   }
44
45   //Create mare::buffers with host ptr containing
46   //the initialization data and to be used as backing store.
47   auto buf_a = mare::create_buffer(a.data(), a.size());
48   auto buf_b = mare::create_buffer(b.data(), b.size());
49
50   //buf_c has no host ptr & initialization data.
51   auto buf_c = mare::create_buffer<float>(a.size());
52
53   std::string kernel_name("vadd");
54   //Create a kernel object, note the kernel parameters used determine
55   //if the data is copied to/from device before/after the kernel
56   //launch. Here buf_a and buf_b are const buffers i.e. they are not
57   //modified by the kernel, hence when they are accessed on host
58   //after kernel launch, data is not copied back from the device.
59   //Similarly buf_c is specified as out<float> which indicates that
60   //kernel modifies the buffer and hence data is not copied to the device
61   //before kernel launch.
62   auto gpu_vadd = mare::create_kernel<mare::buffer_ptr<const float>,
63                                       mare::buffer_ptr<const float>,
64                                       mare::buffer_ptr<mare::out<float>>,
65                                       unsigned int>(vadd_kernel_string,
66                                                     kernel_name);
67
68   //Create a mare::range object, 1D in this case.
69   mare::range<1> range_1d(a.size());
70
71   //Create a task attribute to mark the task as a gpu task.
72   auto attrs = mare::create_task_attrs(mare::attr::gpu);
73
74   unsigned int size = a.size();
75
76   //Use the pfor_each pattern.
77   pfor_each(range_1d,
78             mare::with_attrs(attrs, gpu_vadd, buf_a,
79                              buf_b, buf_c, size));
80
81   //compare the results.
82   for(size_t i = 0; i < a.size(); ++i) {
83     MARE_INTERNAL_ASSERT(a[i] + b[i] == buf_c[i] && buf_c[i] == a.size(),
84                          "comparison failed at ix %zu: %f + %f == %f == %zu",
85                          i, a[i], b[i], buf_c[i], a.size());
86   }
87
88   //shutdown the mare runtime
89   mare::runtime::shutdown();
90 }
```

See Also

`mare::create_ndrange_task`

13 Data Structures

MARE provides a set of concurrent data structures that are optimized for performance using internal MARE primitives.

13.1 Concurrent Obstruction-Free DQueue

Classes

- class mare::cof_deque< T, PREDICTOR >

13.1.1 Class Documentation

13.1.1.1 class mare::cof_deque

template<class T, class PREDICTOR = internal::cof::default_predictor>class mare::cof_deque< T, PREDICTOR >

Bounded double-ended queue, inspired by:

Herlihy, Maurice, Victor Luchangco, and Mark Moir. "Obstruction-free synchronization: Double-ended queues as an example." Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on. IEEE, 2003.

Herlihy, Maurice, Victor Luchangco, and Mark Moir. "Obstruction-free synchronization: Double-ended queues as an example." Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on. IEEE, 2003.

Y. Afek and A. Morrison. Fast concurrent queues for x86 processors. In PPoPP. ACM, 2013.

H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. Michael, and M. Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In POP, 2011.

G. Bar-Nissan, D. Hendler, and A. Suissa. A dynamic elimination-combining stack algorithm. In OPODIS, 2011.

A. Haas, C. Kirsch, M. Lippautz, and H. Payer. How FIFO is your concurrent FIFO queue? In RACES. ACM, 2012.

A. Haas, T. Henzinger, C. Kirsch, M. Lippautz, H. Payer, A. Sezgin, and A. Sokolova. Distributed queues in shared memory-multicore performance and scalability through quantitative relaxation. In Computing Frontiers. ACM, 2013.

T. Harris. A pragmatic implementation of non-blocking linked-lists. In DISC. Springer, 2001.

D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In SPAA. ACM, 2004.

T. Henzinger, H. Payer, and A. Sezgin. Replacing competition with cooperation to achieve scalable lock-free FIFO queues. Technical Report IST-2013-124-v1+1, IST Austria, 2013.

M. Hoffman, O. Shalev, and N. Shavit. The baskets queue. In OPODIS, pages 401-414. Springer, 2007.

D. H. I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In SPAA, pages 355-364. ACM, 2010.

M. Michael. CAS-based lock-free algorithm for shared deques. In Euro-Par. Springer, 2003.

M. Michael and M. Scott. Simple, fast, and practical non- blocking and blocking concurrent queue algorithms. In PODC, pages 267-275. ACM, 1996.

H. Sundell and P. Tsigas. Lock-free deques and doubly linked lists. Journal of Parallel and Distributed Computing, 68, 2008.

The implementation is designed to work with values of any size; however, there is an optimized variant for values whose type is of a size less than or equal to that of the platform's size_t. This has the advantage of providing an optimized variant for native pointers.

Definition at line 78 of file cofdeque.hh.

13.2 Concurrent Obstruction-Free Queue

Classes

- class mare::cof_queue< T, PREDICTOR >

Typedefs

- typedef cof_deque< T, PREDICTOR > **mare::cof_queue< T, PREDICTOR >::container_type**
- typedef T **mare::cof_queue< T, PREDICTOR >::value_type**

Functions

- **mare::cof_queue< T, PREDICTOR >::cof_queue** (size_t qsize)
- **mare::cof_queue< T, PREDICTOR >::cof_queue** (cof_queue< T > &rhs)
- bool mare::cof_queue< T, PREDICTOR >::push (const value_type &v)
- bool mare::cof_queue< T, PREDICTOR >::pop (value_type &r)
- size_t mare::cof_queue< T, PREDICTOR >::size () const

Variables

- container_type **mare::cof_queue< T, PREDICTOR >::_c**

13.2.1 Class Documentation

13.2.1.1 class mare::cof_queue

template<class T, class PREDICTOR = internal::cof::default_predictor>class mare::cof_queue< T, PREDICTOR >

FIFO queue implemention built on top of cofdeque.hh

Note: As it is built on top of MARE's DQueue, similar to that structure, the size is bounded at creation time.

Definition at line 22 of file cofqueue.hh.

Public Types

- typedef cof_deque< T, PREDICTOR > **container_type**
- typedef T **value_type**

Public member functions

- **cof_queue** (size_t qsize)
- **cof_queue** (cof_queue< T > &rhs)
- bool push (const value_type &v)
- bool pop (value_type &r)
- size_t size () const

Private Attributes

- container_type **_c**

13.2.2 Function Documentation

13.2.2.1 template<class T, class PREDICTOR = internal::cof::default_predictor> bool mare::cof_queue< T, PREDICTOR >::push (const value_type & *v*) [inline]

See Also

cof_deque::right_push

Definition at line 31 of file cofqueue.hh.

13.2.2.2 template<class T, class PREDICTOR = internal::cof::default_predictor> bool mare::cof_queue< T, PREDICTOR >::pop (value_type & *r*) [inline]

See Also

cof_deque::left_pop

Definition at line 34 of file cofqueue.hh.

13.2.2.3 template<class T, class PREDICTOR = internal::cof::default_predictor> size_t mare::cof_queue< T, PREDICTOR >::size ( ) const [inline]

See Also

cof_deque::size

Definition at line 37 of file cofqueue.hh.

13.3 Concurrent Obstruction-Free Stack

Classes

- class mare::cof_stack< T, PREDICTOR >

Typedefs

- typedef cof_deque< T, PREDICTOR > **mare::cof_stack< T, PREDICTOR >::container_type**
- typedef T **mare::cof_stack< T, PREDICTOR >::value_type**

Functions

- **mare::cof_stack< T, PREDICTOR >::cof_stack** (size_t qsize)
- **mare::cof_stack< T, PREDICTOR >::cof_stack** (cof_stack< T > &rhs)
- bool mare::cof_stack< T, PREDICTOR >::push (const value_type &v)
- bool mare::cof_stack< T, PREDICTOR >::pop (value_type &r)
- size_t mare::cof_stack< T, PREDICTOR >::size () const

Variables

- container_type **mare::cof_stack< T, PREDICTOR >::_c**

13.3.1 Class Documentation

13.3.1.1 class mare::cof_stack

template<class T, class PREDICTOR = internal::cof::default_predictor>class mare::cof_stack< T, PREDICTOR >

LILO stack implemention built on top of cofdeque.hh

Note: As it is built on top of MARE's DQueue, similar to that structure, the size is bounded at creation time.

Definition at line 18 of file cofstack.hh.

Public Types

- typedef cof_deque< T, PREDICTOR > **container_type**
- typedef T **value_type**

Public member functions

- **cof_stack** (size_t qsize)
- **cof_stack** (cof_stack< T > &rhs)
- bool push (const value_type &v)
- bool pop (value_type &r)
- size_t size () const

Private Attributes

- container_type **_c**

13.3.2 Function Documentation

13.3.2.1 template<class T, class PREDICTOR = internal::cof::default_predictor> bool mare::cof_stack< T, PREDICTOR >::push (const value_type & *v*) [inline]

See Also

cof_deque::right_push

Definition at line 27 of file cofstack.hh.

13.3.2.2 template<class T, class PREDICTOR = internal::cof::default_predictor> bool mare::cof_stack< T, PREDICTOR >::pop (value_type & *r*) [inline]

See Also

cof_deque::right_pop

Definition at line 30 of file cofstack.hh.

13.3.2.3 template<class T, class PREDICTOR = internal::cof::default_predictor> size_t mare::cof_stack< T, PREDICTOR >::size ( ) const [inline]

See Also

cof_deque::size

Definition at line 33 of file cofstack.hh.

14 Data Sharing

14.1 Task Storage

Using task storage requires including the following header file:

```
#include <mare/taskstorage.hh>
```

Classes

- class mare::task_storage_ptr< T >

14.1.1 Class Documentation

14.1.1.1 class mare::task_storage_ptr

template<typename T>class mare::task_storage_ptr< T >

Task-local storage allows sharing of information on a per-task basis, similar as thread-local storage does for threads. The value of a 'task_storage_ptr' is local to a task. A 'task_storage_ptr<T>' stores a pointer-to-T ('T*').

Example

```
1 namespace {
2 task_storage_ptr<int> storage;
3 }; // namespace
4
5 void func() {
6   printf("%d\n", *storage);
7   ++*storage;
8 }
9
10 void run() {
11   auto g = create_group("test");
12   for (int i = 0; i < N; ++i) {
13     launch(g, [i] {
14         int v = i;
15         storage = &v;
16         func();
17         assert(v == i+1);
18         func();
19         assert(v == i+2);
20       });
21   }
22   wait_for(g);
23 }
24
```

Definition at line 23 of file taskstorage.hh.

Public Types

- typedef T const * **pointer_type**

Public member functions

- **MARE_DELETE_METHOD** (task_storage_ptr(task_storage_ptr const &))
- **MARE_DELETE_METHOD** (task_storage_ptr &operator=(task_storage_ptr const &))
- **MARE_DELETE_METHOD** (task_storage_ptr &operator=(task_storage_ptr const &) volatile)
- task_storage_ptr ()
- task_storage_ptr (T ∗const &ptr)
- task_storage_ptr (T ∗const &ptr, void(∗dtor)(T ∗))
- T ∗ get () const
- T ∗ operator-> () const
- T & operator∗ () const
- task_storage_ptr & operator= (T ∗const &ptr)
- bool operator! () const
- bool **operator==** (T ∗const &other) const
- bool **operator!=** (T ∗const &other) const
- operator pointer_type () const
- operator bool () const

Private member functions

- void **init_key** (void(∗dtor)(T ∗)=nullptr)

Private Attributes

- internal::storage_key **_key**

14.1.1.1.1 Constructors and Destructors

14.1.1.1.1.1 template<typename T > mare::task_storage_ptr< T >::task_storage_ptr ( ) [inline]

Exceptions

mare::tls_exception	if ‘task_storage_ptr‘ could not be reserved.

Definition at line 35 of file taskstorage.hh.

14.1.1.1.1.2 template<typename T > mare::task_storage_ptr< T >::task_storage_ptr (T ∗const & *ptr*) [inline]

Exceptions

mare::tls_exception	if ‘task_storage_ptr‘ could not be reserved.

Parameters

ptr	initial value of task-local storage.

Definition at line 44 of file taskstorage.hh.

14.1.1.1.1.3 template<typename T > mare::task_storage_ptr< T >::task_storage_ptr (T *const & *ptr*, void(*)(T *) *dtor*) [inline]

Exceptions

mare::tls_exception	if 'task_storage_ptr' could not be reserved.

Parameters

ptr	initial value of task-local storage.
dtor	destructor function.

Definition at line 56 of file taskstorage.hh.

14.1.1.1.2 Member Function Documentation

14.1.1.1.2.1 template<typename T > T* mare::task_storage_ptr< T >::get ( ) const [inline]

Returns

Pointer to stored pointer value.

Definition at line 63 of file taskstorage.hh.

14.1.1.1.2.2 template<typename T > T* mare::task_storage_ptr< T >::operator-> ( ) const [inline]

Returns

Pointer to stored pointer value.

Definition at line 68 of file taskstorage.hh.

14.1.1.1.2.3 template<typename T > T& mare::task_storage_ptr< T >::operator* ( ) const [inline]

Returns

Reference to value pointed to by stored pointer.

Note: No checking for 'nullptr' is performed.

Definition at line 77 of file taskstorage.hh.

14.1.1.1.2.4 template<typename T > task_storage_ptr& mare::task_storage_ptr< T >::operator= (T *const & *ptr*) [inline]

Assignment operator, stores T*.

Parameters

ptr	Pointer value to store.

Definition at line 86 of file taskstorage.hh.

14.1.1.1.2.5 template<typename T > bool mare::task_storage_ptr< T >::operator! ( ) const [inline]

Returns

True if stored pointer is ‘nullptr‘.

Definition at line 92 of file taskstorage.hh.

14.1.1.1.2.6 template<typename T > mare::task_storage_ptr< T >::operator pointer_type ( ) const [inline]

Casting operator to ‘T*‘ pointer type.

Definition at line 106 of file taskstorage.hh.

14.1.1.1.2.7 template<typename T > mare::task_storage_ptr< T >::operator bool ( ) const [inline]

Casting operator to bool.

Definition at line 111 of file taskstorage.hh.

14.2 Scoped Storage

Using scoped storage requires including the following header file:

```
#include <mare/scopedstorage.hh>
```

Classes

- class mare::scoped_storage_ptr< Scope, T, Allocator >

Variables

- static Allocator **mare::scoped_storage_ptr< Scope, T, Allocator >::_alloc**

14.2.1 Class Documentation

14.2.1.1 class mare::scoped_storage_ptr

template<template< class, class > class Scope, typename T, class Allocator>class mare::scoped_storage_ptr< Scope, T, Allocator >

Scoped storage allows sharing of information on a per-task basis, similar as thread-local storage does for threads. A 'scoped_storage_ptr<Scope,T,Allocator>' stores a pointer-to-T ('T*') for a given scope 'Scope<T,A>', defining life time, persistence, etc.. 'Allocator' controls the allocation of T objects.

Definition at line 21 of file scopedstorage.hh.

Public Types

- typedef Scope< T, Allocator > **scope_type**
- typedef T const * **pointer_type**

Public member functions

- **MARE_DELETE_METHOD** (scoped_storage_ptr(scoped_storage_ptr const &))
- **MARE_DELETE_METHOD** (scoped_storage_ptr &operator=(scoped_storage_ptr const &))
- **MARE_DELETE_METHOD** (scoped_storage_ptr &operator=(scoped_storage_ptr const &) volatile)
- **MARE_DELETE_METHOD** (scoped_storage_ptr &operator=(T *const &))
- scoped_storage_ptr ()
- T * get () const
- T * operator-> () const
- T & operator* () const
- bool operator! () const
- bool **operator==** (T *const &other)
- bool **operator!=** (T *const &other)

- operator pointer_type () const
- operator bool () const

Private member functions

- void **init_key** ()

Static Private Member Functions

- static void **maybe_invoke_dtor** (void *ptr)

Private Attributes

- internal::storage_key **_key**

Static Private Attributes

- static Allocator **_alloc**

14.2.1.1.1 Constructors and Destructors

14.2.1.1.1.1 template<template< class, class > class Scope, typename T, class Allocator> mare::scoped_storage_ptr< Scope, T, Allocator >::scoped_storage_ptr ( ) [inline]

Exceptions

mare::tls_exception	if 'scoped_storage_ptr' could not be reserved

Definition at line 38 of file scopedstorage.hh.

14.2.1.1.2 Member Function Documentation

14.2.1.1.2.1 template<template< class, class > class Scope, typename T, class Allocator> T* mare::scoped_storage_ptr< Scope, T, Allocator >::get ( ) const [inline]

Returns

Stored pointer value; a new object of type 'T' is created and stored, if it has not been stored before.

Definition at line 48 of file scopedstorage.hh.

14.2.1.1.2.2 template<template< class, class > class Scope, typename T, class Allocator> T* mare::scoped_storage_ptr< Scope, T, Allocator >::operator-> ( ) const [inline]

Returns

Stored pointer value; a new object of type 'T' is created and stored, if it has not been stored before.

Definition at line 62 of file scopedstorage.hh.

14.2.1.1.2.3 template<template< class, class > class Scope, typename T, class Allocator> T& mare::scoped_storage_ptr< Scope, T, Allocator >::operator* ( ) const [inline]

Returns

Reference to value pointed to by stored pointer; A new object of type 'T' is created and stored, if it has not been stored before.

Definition at line 71 of file scopedstorage.hh.

14.2.1.1.2.4 template<template< class, class > class Scope, typename T, class Allocator> bool mare::scoped_storage_ptr< Scope, T, Allocator >::operator! ( ) const [inline]

Returns

Constantly false.

Definition at line 78 of file scopedstorage.hh.

14.2.1.1.2.5 template<template< class, class > class Scope, typename T, class Allocator> mare::scoped_storage_ptr< Scope, T, Allocator >::operator pointer_type ( ) const [inline]

Casting operator to 'T*' pointer type.

Definition at line 98 of file scopedstorage.hh.

14.2.1.1.2.6 template<template< class, class > class Scope, typename T, class Allocator> mare::scoped_storage_ptr< Scope, T, Allocator >::operator bool ( ) const [inline]

Casting operator to bool (constantly true).

Definition at line 103 of file scopedstorage.hh.

14.3 Scheduler Storage

Using scheduler storage requires including the following header file:

```
#include <mare/schedulerstorage.hh>
```

Classes

- class mare::scheduler_storage_ptr< T, Allocator >

14.3.1 Class Documentation

14.3.1.1 class mare::scheduler_storage_ptr

template<typename T, class Allocator = std::allocator<T>>class mare::scheduler_storage_ptr< T, Allocator >

Scheduler-local storage allows sharing of information on a per-task basis, similar as thread-local storage does for threads. A 'scheduler_storage_ptr<T>' stores a pointer-to-T ('T*'). In contrast to 'task_storage_ptr', the contents are persistent between tasks. In contrast to 'thread_storage_ptr', the contents are guaranteed to be unchanged across context switches. To maintain these guarantees, the runtime system is free to create new objects of type T whenever needed.

See Also

task_storage_ptr
thread_storage_ptr

Example

```
1 #include <assert.h>
2
3 #include <algorithm>
4 #include <iterator>
5
6 #include <mare/mare.h>
7 #include <mare/schedulerstorage.hh>
8
9 using namespace mare;
10
11 template <size_t N>
12 struct image_scratchpad {
13   image_scratchpad() {
14     std::fill(begin(edge_image), end(edge_image), 0);
15   }
16   char edge_image[N];
17 };
18
19 namespace {
20 const scheduler_storage_ptr<image_scratchpad<4096> >
       image_buffers;
21 }; // namespace
22
23 void run() {
24   int const N = 200;
25
26   auto g = create_group("test");
27   for (int i = 1; i < N; ++i) {
28     launch(g, [i] {
29         // fill image buffer, which is reused across tasks
30         for (auto& slot : image_buffers->edge_image)
```

```
31           slot = i & 0xff;
32         internal::yield(); // context-switch, we expect SLS to survive this
33         // check contents
34         for (auto const& slot : image_buffers->edge_image)
35           assert(slot == char(i & 0xff));
36       });
37   }
38   wait_for(g);
39 }
```

```
1 #include <mare/schedulerstorage.hh>
2
3 namespace {
4 const scheduler_storage_ptr<size_t> s_sls_state;
5 }; // namespace
6
7 void run() {
8   auto g = create_group("test");
9
10   for (size_t i = 0; i < 200; ++i) {
11     launch(g, [i] {
12         size_t c = ++*s_sls_state;
13         // values for c are consecutive on a per-scheduler basis
14       });
15   }
16
17   wait_for(g);
18 }
```

```
1 #include <mare/schedulerstorage.hh>
2
3 namespace {
4 const scheduler_storage_ptr<size_t> s_sls_state;
5 }; // namespace
6
7 void run() {
8   auto g = create_group("test");
9   auto t = create_task([] {});
10
11   for (size_t i = 0; i < 200; ++i) {
12     launch(g, [=] {
13         size_t c1 = ++*s_sls_state;
14         launch(t);
15         wait_for(t);
16         size_t c2 = ++*s_sls_state;
17         assert(c1 + 1 == c2);
18       });
19   }
20
21   wait_for(g);
22 }
```

Definition at line 33 of file schedulerstorage.hh.

Public Types

- typedef Allocator **allocator_type**

Static Public Member Functions

- static int **key_create** (internal::storage_key ∗key, void(∗dtor)(void ∗))
- static int **set_specific** (internal::storage_key key, void const ∗value)
- static void ∗ **get_specific** (internal::storage_key key)

Additional Inherited Members

14.4 Thread Storage

Using thread storage requires including the following header file:

```
#include <mare/threadstorage.hh>
```

Classes

- class mare::thread_storage_ptr< T, Allocator >

14.4.1 Class Documentation

14.4.1.1 class mare::thread_storage_ptr

template<typename T, class Allocator = std::allocator<T>>class mare::thread_storage_ptr< T, Allocator >

Thread-local storage allows sharing of information on a per-task basis, similar as thread-local storage does for threads. A ‘scheduler_storage_ptr<T>‘ stores a pointer-to-T (‘T∗‘). In contrast to ‘task_storage_ptr‘, the contents are persistent between tasks. In contrast to ‘scheduler_storage_ptr‘, the contents might be accessed by others while a task is suspended.

See Also

task_storage_ptr
scheduler_storage_ptr

Example

```
1 #include <mare/threadstorage.hh>
2
3 namespace {
4 const thread_storage_ptr<size_t> s_tls_state;
5 }; // namespace
6
7 void run() {
8   auto g = create_group("test");
9   auto t = create_task([] {});
10
11   for (size_t i = 0; i < 200; ++i) {
12     launch(g, [=] {
13         size_t* p1 = s_tls_state.get();
14         launch(t);
15         wait_for(t);
16         size_t* p2 = s_tls_state.get();
17         // cannot assume that p1 == p2
18       });
19   }
20
21   wait_for(g);
22 }
```

Definition at line 29 of file threadstorage.hh.

Public Types

- typedef Allocator **allocator_type**

Static Public Member Functions

- static int **key_create** (internal::storage_key ∗key, void(∗dtor)(void ∗))
- static int **set_specific** (internal::storage_key key, void const ∗value)
- static void ∗ **get_specific** (internal::storage_key key)

Additional Inherited Members

15 Exceptions Reference API

In this chapter we discuss all the exceptions thrown by the MARE runtime system.

15.1 Exceptions

Classes

- class mare::mare_exception
- class mare::error_exception
- class mare::api_exception
- class mare::tls_exception
- class mare::abort_task_exception

15.1.1 Class Documentation

15.1.1.1 class mare::mare_exception

Superclass of all MARE-generated exceptions.

Definition at line 27 of file exceptions.hh.

Public member functions

- virtual ~mare_exception () MARE_NOEXCEPT
- virtual const char * what () const =0 throw ()

15.1.1.1.1 Constructors and Destructors

15.1.1.1.1.1 virtual mare::mare_exception::~mare_exception ( ) [inline], [virtual]

Destructor.

Definition at line 31 of file exceptions.hh.

15.1.1.1.2 Member Function Documentation

15.1.1.1.2.1 virtual const char* mare::mare_exception::what ( ) const throw) [pure virtual]

Returns exception description.

Returns

C string describing the exception.

Implemented in mare::abort_task_exception, and mare::error_exception.

15.1.1.2 class mare::error_exception

Superclass of all MARE-generated exceptions that indicate internal or programmer errors.

Definition at line 44 of file exceptions.hh.

Public member functions

- **error_exception** (std::string msg, const char ∗filename, int lineno, const char ∗funcname)
- virtual const char ∗ **message** () const throw ()
- virtual const char ∗ what () const throw ()
- virtual const char ∗ **file** () const throw ()
- virtual int **line** () const throw ()
- virtual const char ∗ **function** () const throw ()
- virtual const char ∗ **type** () const throw ()

Static Private Member Functions

- static std::string **compiler_demangle** (std::type_info const &mytype)

Private Attributes

- std::string **_message**
- std::string **_long_message**
- std::string **_file**
- int **_line**
- std::string **_function**
- std::string **_classname**

15.1.1.2.1 Member Function Documentation

15.1.1.2.1.1 virtual const char∗ mare::error_exception::what ( ) const throw) [inline], [virtual]

Returns exception description.

Returns

C string describing the exception.

Implements mare::mare_exception.

Definition at line 61 of file exceptions.hh.

15.1.1.3 class mare::api_exception

Represents a misuse of the MARE API. For example, invalid values passed to a function. Should cause termination of the application (future releases will behave differently).

Definition at line 84 of file exceptions.hh.

Public member functions

- **api_exception** (std::string msg, const char ∗filename, int lineno, const char ∗funcname)

15.1.1.4 class mare::tls_exception

Indicates that the thread TLS has been misused or become corrupted. Should cause termination of the application.

Definition at line 133 of file exceptions.hh.

Public member functions

- **tls_exception** (std::string msg, const char ∗filename, int lineno, const char ∗funcname)

15.1.1.5 class mare::abort_task_exception

Exception thrown to abort the current task.

See Also

mare::abort_on_cancel()
mare::abort_task()

Definition at line 148 of file exceptions.hh.

Public member functions

- **abort_task_exception** (std::string msg="aborted task")
- virtual const char ∗ what () const throw ()

Private Attributes

- std::string **_msg**

15.1.1.5.1 Member Function Documentation

15.1.1.5.1.1 virtual const char∗ mare::abort_task_exception::what ( ) const throw) [inline], [virtual]

Returns exception description.

Returns

C string describing the exception.

Implements mare::mare_exception.

Definition at line 157 of file exceptions.hh.

16 Class Documentation

16.1 mare::aligned_allocator< T, Alignment > Struct Template Reference

Classes

- struct rebind

Public Types

- typedef std::allocator< T > ::size_type **size_type**
- typedef std::allocator< T > ::pointer **pointer**
- typedef std::allocator< T > ::const_pointer **const_pointer**

Public member functions

- **MARE_GCC_IGNORE_END** ("-Weffc++")
- **aligned_allocator** (const aligned_allocator &other) throw ()
- template<class U > **aligned_allocator** (const aligned_allocator< U, Alignment > &) throw ()
- pointer **allocate** (size_type n)
- pointer **allocate** (size_type n, const_pointer)
- void **deallocate** (pointer p, size_type)

Additional Inherited Members

template<class T, size_t Alignment>struct mare::aligned_allocator< T, Alignment >

Definition at line 48 of file alignedallocator.hh.

16.1.1 Class Documentation

16.1.1.1 struct mare::aligned_allocator::rebind

template<class T, size_t Alignment>template<class U>struct mare::aligned_allocator< T, Alignment >::rebind< U >

Definition at line 59 of file alignedallocator.hh.

Data fields

Type	Field	Description
typedef aligned_-allocator< U, Alignment >	other	

The documentation for this struct was generated from the following file:

- /var/lib/jenkins/workspace/mare_installer_master/mare.git/include/mare/alignedallocator.hh

16.2 body_with_attrs_base_gpu Class Reference

The documentation for this class was generated from the following file:

- /var/lib/jenkins/workspace/mare_installer_master/mare.git/include/mare/attrobjs.hh

Index

Bibliography

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29:66–76, 1995.

[2] Gene M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485, Reston, VA, April 1967.

[3] Christopher Barton, Călin Cascaval, and José Nelson Amaral. A characterization of shared data access patterns in upc programs. In *Proceedings of the 19th international conference on Languages and compilers for parallel computing*, LCPC'06, pages 111–125, Berlin, Heidelberg, 2007. Springer-Verlag.

[4] Antoni Buades, Bartomeu Coll, and Jean-Michel Morel. A non-local algorithm for image denoising. In *Computer Vision and Pattern Recognition*, 2005.

[5] C++11 standard – working draft n3485. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3485.pdf, Nov 2012.

[6] Calin Cascaval, Seth Fowler, Pablo Montesinos-Ortego, Wayne Piekarski, Mehrdad Reshadi, Behnam Robatmili, Michael Weber, and Vrajesh Bhavsar. Zoomm: a parallel web browser engine for multicore mobile devices. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '13, pages 271–280, New York, NY, USA, 2013. ACM.

[7] Stephanie Coleman and Kathryn S. McKinley. Tile Size Selection Using Cache Organization and Data Layout. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI '95*, La Jolla, CA, June 1995. SIGPLAN.

[8] Michael J. Flynn. Some computer organizations and their effectivenes. *IEEE Transactions on Computers*, C-21(9):948–960, Sept. 1972.

[9] Benedict R. Gaster and Lee Howes. Can GPGPU programming be liberated from the data-parallel bottleneck? *IEEE Computer*, pages 42–52, 2012.

[10] Marc Gregoire, Nicholas A. Solter, and Scott J. Klepper. *Professional C++*. John Wiley and Sons, Inc., 2nd edition, 2011.

[11] John L. Gustafson. Reevaluating Amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988.

[12] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann, second edition, 1996.

[13] Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. *IEEE Computer*, July 2008.

[14] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, December 1986.

[15] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4.5):589–604, 2005. http://dx.doi.org/10.1147/rd.494.0589.

[16] Milind Kulkarni, Martin Burtscher, Rajeshkar Inkulu, Keshav Pingali, and Calin Casçaval. How much parallelism is there in irregular applications? In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 3–14, New York, NY, USA, 2009. ACM.

[17] NEON intrinsics. http://gcc.gnu.org/onlinedocs/gcc/ARM-NEON-Intrinsics.html, Apr 2013.

[18] Qualcomm Research Silicon Valley. *MARE: Multicore Asynchronous Runtime Environment User's Manual*, 0.8 edition, April 2013.

[19] Gabriel Rivera and Chau-Wen Tseng. Data transformations for eliminating conflict misses. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI '98*, pages 38–49, June 1998.

[20] Anne Rogers and Keshav Pingali. Process decomposition through locality of reference. *SIGPLAN Notices*, 24(7):69–80, July 1989.

[21] Josep Torrellas, Monica S. Lam, and John L. Hennessy. Shared Data Placement Optimizations to Reduce Multiprocessor Cache Miss Rates. In *Proceedings of the International Conference on Parallel Processing*, pages II–266–II–270, 1990.

[22] Michael Wolfe. More Iteration Space Tiling. In *Proceedings of Supercomputing '89*, pages 655–664, Reno, NV, November 1989. ACM.