SAND2022-13279C

Sandia National Laboratories

Exceptional service in the national interest

Advanced Threading Features in Open MPI

Jan Ciesko

Wednesday, 28th September, 2022

UT, Chattanooga, Tennessee,

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Agenda

Problem Statement

Solutions Today

User-level Threading Support in Open MPI

Other Techniques

Summary and Outlook

User-level Threading in Open MPI

Cooperative multithreading requires MPI to be cooperative

- **Example:**

Deadlock depends on task ordering and on the number of … ?

Rank 0

Ready-task Queue

```
1  void task_send(task_args_t *args) {
2    int ret;
3    MPI_Request request;
4    MPI_Send(&args->some_data, 1, MPI_INT, args->target,
5      0, comm, &request);
6  }
```

```
1  void task_recv(task_args_t *args) {
2    MPI_Request request;
3    MPI_Recv(&args->some_data, 1, MPI_INT, args->target,
4      0, comm, &request);
5  }
```

```
1  void task_send(task_args_t *args) {
2    int ret;
3    MPI_Request request;
4    MPI_Send(&args->some_data, 1, MPI_INT, args->target,
5      0, comm, &request);
6  }
```

Rank 1

Ready-task Queue

```
1  void task_recv(task_args_t *args) {
2    MPI_Request request;
3    MPI_Recv(&args->some_data, 1, MPI_INT, args->target,
4      0, comm, &request);
5  }
```

```
1  void task_send(task_args_t *args) {
2    int ret;
3    MPI_Request request;
4    MPI_Send(&args->some_data, 1, MPI_INT, args->target,
5      0, comm, &request);
6  }
```

```
1  void task_recv(task_args_t *args) {
2    MPI_Request request;
3    MPI_Recv(&args->some_data, 1, MPI_INT, args->target,
4      0, comm, &request);
5  }
```

https://github.com/sandialabs/MPI-Partix

Solutions today

User-level Threading Support in MPI (>=4.x)

User-level Threading Support Through MPI Continuations (events)

API Overloads for Proprietary Event Handling

Solutions today

User-level Threading Support in MPI (>=4.x)

User-level Threading Support Through MPI Continuations (events)

API Overloads for Proprietary Event Handling

User-level Threading in Open MPI

Open MPI 4.x and MPICH 3.4.x have ULT* support

Easy to use with configure options

The use of ULTs can be beneficial for performance

Hybrid applications require ULT support in the MPI implementation for correctness (progress guarantees)

**Note: User-level Threading (ULT) refers to any threading implementation where the operating system is not aware of such threads. Such threads or "tasks" are light-weight but require cooperative behavior for progress guarantees (cooperative multithreading)*

Configure options: Open MPI

```
--with-threads=TYPE      Specify thread TYPE to use. default:pthreads. Other
                         options are qthreads and argobots.
--with-argobots=DIR      Specify location of argobots installation. Error if
                         argobots support cannot be found.
--with-argobots-libdir=DIR
                         Search for argobots libraries in DIR
--with-qthreads=DIR      Specify location of qthreads installation. Error if
                         qthreads support cannot be found.
--with-qthreads-libdir=DIR
                         Search for qthreads libraries in DIR
```

MPICH

```
--with-thread-package=package
--with-argobots=[PATH]
--with-argobots-include=PATH
--with-argobots-lib=PATH
```

Note: Use *ompi_info --config* to see your MPI configuration

User-level Threading in Open MPI

Two new important changes

- Internal synchronization primitive are now cooperative
  - Implements common interface and two back-ends (Qthreads and Argobots).

Note: Easily extensible

Declarations:

```
1  static void mca_threads_mutex_constructor(opal_mutex_t *p_mutex) {
2      opal_thread_internal_mutex_init(&p_mutex->m_lock, false);
3  }
4
5  static void mca_threads_mutex_destructor(opal_mutex_t *p_mutex) {
6      opal_thread_internal_mutex_destroy(&p_mutex->m_lock);
7  }
8
9  static void mca_threads_recursive_mutex_constructor(opal_recursive_mutex_t *p_mutex){
10     opal_thread_internal_mutex_init(&p_mutex->m_lock, true);
11     opal_atomic_lock_init(&p_mutex->m_lock_atomic, 0);
12 }
13
14 static void mca_threads_recursive_mutex_destructor(opal_recursive_mutex_t *p_mutex){
15     opal_thread_internal_mutex_destroy(&p_mutex->m_lock);
16 }
17
18 int opal_cond_init(opal_cond_t *cond){
19     return opal_thread_internal_cond_init(cond);
20 }
21
22 int opal_cond_wait(opal_cond_t *cond, opal_mutex_t *lock){
23     opal_thread_internal_cond_wait(cond, &lock->m_lock);
24     return OPAL_SUCCESS;
25 }
26
27 int opal_cond_broadcast(opal_cond_t *cond){
28     opal_thread_internal_cond_broadcast(cond);
29     return OPAL_SUCCESS;
30 }
31
32 int opal_cond_signal(opal_cond_t *cond){
```

OMPI_SRC/opal/mca/threads/base/mutex.c

User-level Threading in Open MPI

OMPI_SRC/opal/mca/threads/base/mutex.c

Two new important changes

- Internal synchronization primitive are now cooperative
  - Implements common interface and two back-ends (Qthreads and Argobots).

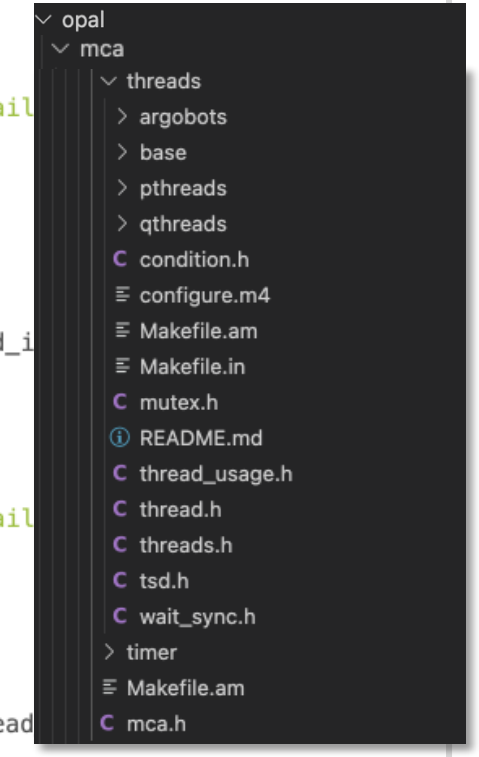
Note: Easily extensible

Definitions (Qthreads, Argobots, Pthread)

```c
static inline int opal_thread_internal_mutex_init(opal_thread_internal_mutex_t *p_mutex,
                                                  bool recursive)
{
    opal_threads_ensure_init_qthreads();
    #if OPAL_ENABLE_DEBUG
    int ret = qthread_spinlock_init(p_mutex,recursive);
    if (QTHREAD_SUCCESS != ret) {
        opal_show_help("help-opal-threads.txt", "mutex init fail
    }
    #else
    qthread_spinlock_init(p_mutex,recursive);
    #endif
    return OPAL_SUCCESS;
}

static inline void opal_thread_internal_mutex_lock(opal_thread_i
{
    opal_threads_ensure_init_qthreads();
    #if OPAL_ENABLE_DEBUG
    int ret = qthread_spinlock_lock(p_mutex);
    if (QTHREAD_SUCCESS != ret) {
        opal_show_help("help-opal-threads.txt", "mutex lock fail
    }
    #else
    qthread_spinlock_lock(p_mutex);
    #endif
}

static inline int opal_thread_internal_mutex_trylock(opal_thread
{
    opal_threads_ensure_init_qthreads();
    int ret = qthread_spinlock_trylock(p_mutex);
    if (QTHREAD_OPFAIL == ret) {
        return 1;
```



User-level Threading in Open MPI

OMPI_SRC/opal/runtime/opal_progress.c

Two new important changes

- Internal synchronization primitive are now cooperative
  - Implements common interface and two back-ends (Qthreads and Argobots).
- The progress engine support the query of task schedulers in progress loop

Progress loop

```
1  int opal_progress(void)
2  {
3      ...
4      /* progress all registered callbacks */
5      for (i = 0; i < callbacks_len; ++i) {
6          events += (callbacks[i])();
7      }
8      ...
9      if (opal_progress_yield_when_idle && events <= 0) {
10         /* If there is nothing to do - yield the processor - otherwise
11          * we could consume the processor for the entire time slice. If
12          * the processor is oversubscribed - this will result in a best-case
13          * latency equivalent to the time-slice.
14          * With some thread implementations, yielding might be required
15          * to ensure correct scheduling of all communicating threads.
16          */
17         opal_thread_yield();
18     }
19     return events;
20 }
```

Note: works with supported ULTs only

User-level Threading in Open MPI

Experiment with “MPI Partix”

- Application test suite for **user-level threading and partitioned communication.**
- Contains API examples, benchmarks and correctness tests
- Works with different threading backends

Example code:

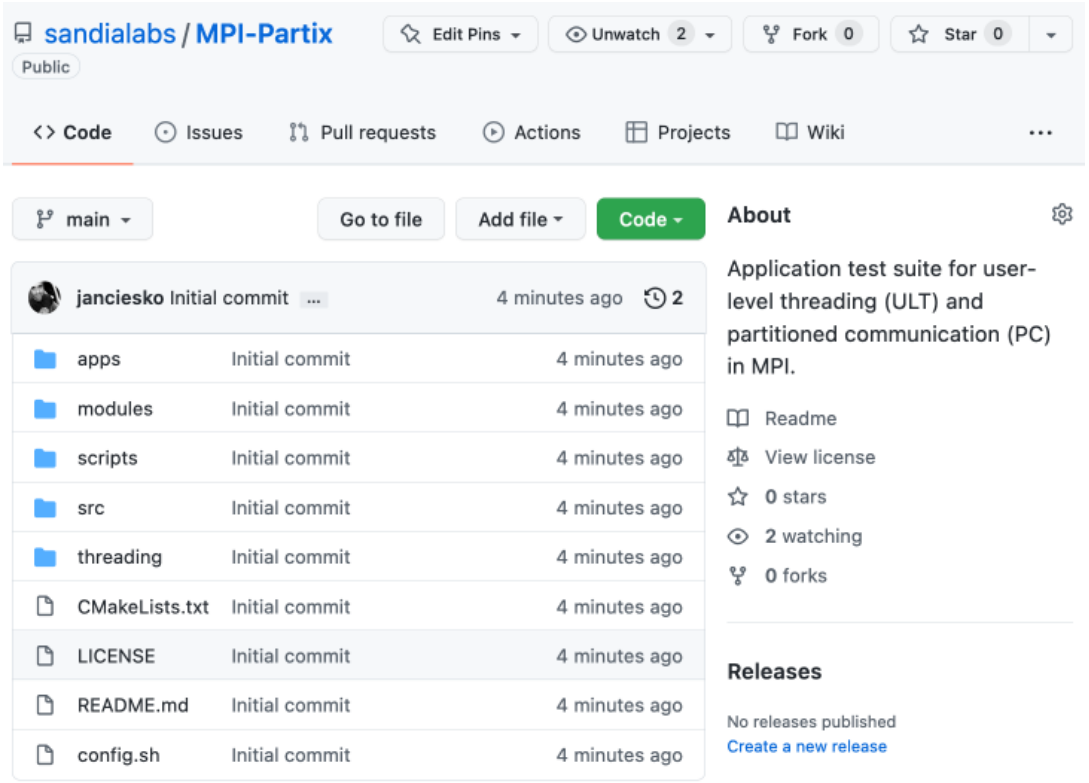
```cpp
#include <cstdio>
#include <partix.h>

void task(partix_task_args_t *args) { printf("Hello World\n"); }

int main(int argc, char *argv[]) {
  partix_config_t conf;
  partix_init(argc, argv, &conf);
  partix_library_init();

  partix_context_t ctx;

  for (int i = 0; i < conf.num_tasks; ++i) {
    partix_task(&task /*functor*/, NULL, &ctx);
  }
  partix_taskwait(&ctx);
  partix_library_finalize();
  return 0;
}
```



https://github.com/sandialabs/MPI-Partix

User-level Threading in Open MPI

Deadlocks with MPI Partix

- **Example:**

```c
//set context
partix_context_t ctx;

#if defined(OMP)
#pragma omp parallel num_threads(conf.num_threads)
#pragma omp single
#endif
  // Create in this sequence that starts and ends with recv tasks
  // This tests correct ULT functionality with ULT libs with
  // FIFO or LIFO schedulers
  for (int i = 0; i < conf.num_tasks; i += 2) {
    if (i < 2) {
      partix_task(&task_recv, &task_args, &ctx);
      partix_task(&task_send, &task_args, &ctx);
    } else {
      partix_task(&task_send, &task_args, &ctx);
      partix_task(&task_recv, &task_args, &ctx);
    }
  }

  partix_taskwait(&ctx);

  assert(reduction_var == DEFAULT_VALUE * conf.num_tasks);
```

```c
void task_send(partix_task_args_t *args) {
  int ret;
  MPI_Request request;
  task_args_t *task_args=(task_args_t *)args->user_task_args;

  MPI_Isend(&task_args->some_data, 1, MPI_INT, task_args->target,
            0, comm, &request);
  MPI_Wait(&request, MPI_STATUS_IGNORE);

  partix_mutex_enter(&mutex);
  reduction_var += task_args->some_data;
  partix_mutex_exit(&mutex);
}
```

```c
void task_recv(partix_task_args_t *args) {
  int ret, tmp;
  MPI_Request request;
  task_args_t *task_args=(task_args_t *)args->user_task_args;

  MPI_Irecv(&tmp, 1, MPI_INT, task_args->target, 0,
            comm, &request);
  MPI_Wait(&request, MPI_STATUS_IGNORE);

  partix_mutex_enter(&mutex);
  reduction_var += tmp;
  partix_mutex_exit(&mutex);
}
```

Supported for Argobots and Qthreads.

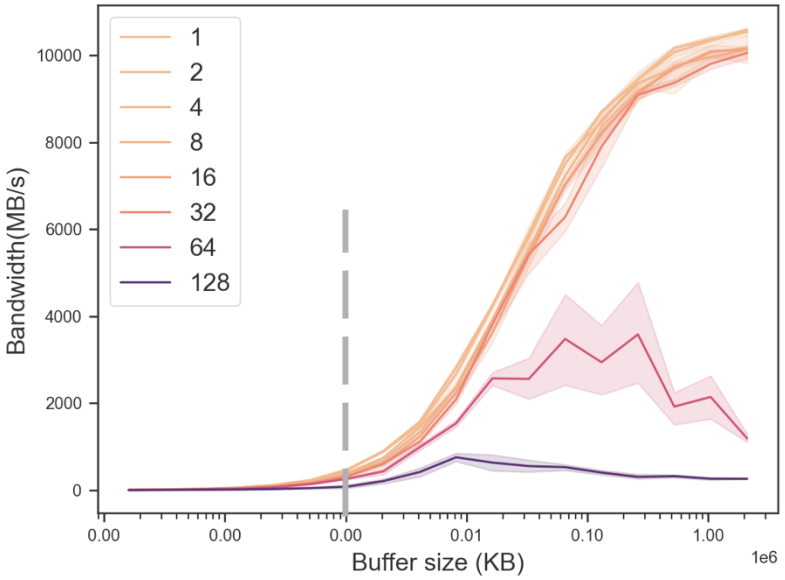
OpenMP tasking requires MPI Continuations here.

https://github.com/sandialabs/MPI-Partix

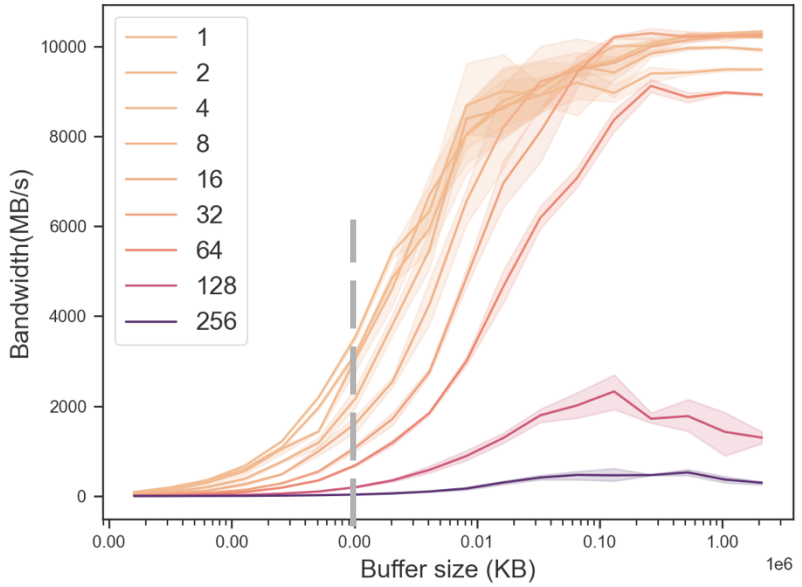
User-level Threading in Open MPI

User-level threading (ULT) + Partitioned Communication (Basic)

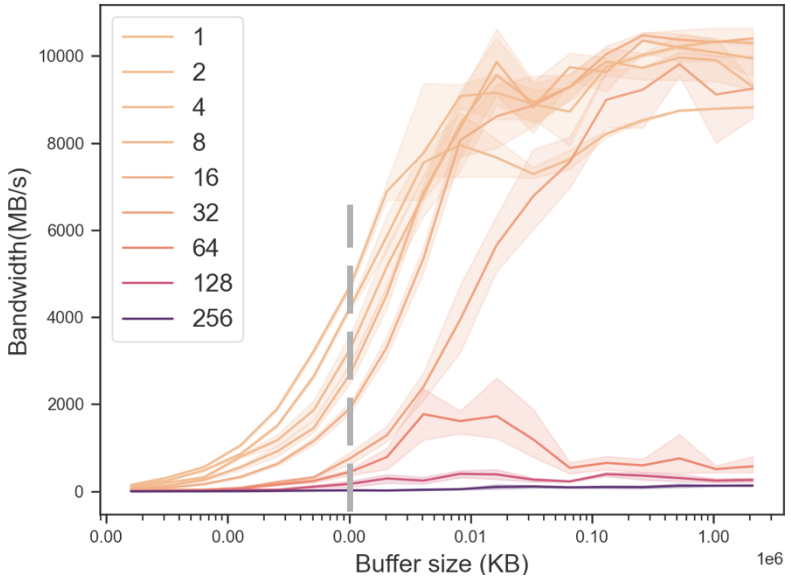
- MPI Partix: **“Bench1”,** 16KB-2GB buffer size, 1:1 partitions to task mapping



Blake, x86, UCX, OMPI 5.0.X, **Pthreads**, 1 -128 partitions, 1:1 P2T



Blake, x86, UCX, OMPI 5.0.X, **Qthreads**, 1-256 partitions, 1:1 P2T



Blake, x86, UCX, OMPI 5.0.X, **OMP Task**, 1-256 partitions, 1:1 P2T

More benchmarks included. Feel free to experiment and report back!

https://github.com/sandialabs/MPI-Partix

Solutions today

User-level Threading Support in MPI (>=4.x)

User-level Threading Support Through MPI Continuations (events)

API Overloads for Proprietary Event Handling

ULT Support through MPI Continuations

MPI Continuations is an MPI API proposal

- See Joseph Schuchart's spec draft [1]
- Allows association of completion of operation requests with callbacks (continuations)
- Upon completion of registered operation request(s), the continuation invoked
- Operation requests and callbacks are associated using the continuation request type

- `int MPI_Continue_init(MPI_Info info, MPI_Request *cont_req);`
- `int MPI_Continue(MPI_Request *op_request, MPI_Continue_cb_function cb, void *cb_data, MPI_Status *status, MPI_Request cont_request)`
- `int MPI_Continueall(int count, MPI_Request array_of_op_requests[], MPI_Continue_cb_function cb, void *cb_data, MPI_Status array_of_statuses[], MPI_Request cont_request)`

ULT Support through MPI Continuations

Example (similar like we had before):

```
1   void task_recv(task_args_t *args) {
2     //Rank 0
3     MPI_Request cont_req;
4     MPIX_Continue_init(&cont_req, MPI_INFO_NULL);
5     ...
6     {
7       MPI_Request req;
8       MPI_Irecv(&value, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &req);
9       MPIX_Continue(&req, &release_event, (void *) event,
10        MPI_STATUS_IGNORE, cont_req);
11
12    }
13  }
```

```
15  void task_recv_continuation(task_args_t *args) {
16  {
17    printf("Received: %d\n", value);
18  }
```

&

```
20  void release_event(MPI_Status *status, void *data) {
21    event_t * event = (event_t*) data;
22    runtime_associate(event, &task_recv_continuation);
23  }
```

&

```
1   void task_recv(task_args_t *args) {
2     MPI_Request request;
3     MPI_Recv(&args->some_data, 1, MPI_INT, args->target,
4       0, comm, &request);
5   }
```

```
1   void task_send(task_args_t *args) {
2     int ret;
3     MPI_Request request;
4     MPI_Send(&args->some_data, 1, MPI_INT, args->target,
5       0, comm, &request);
6   }
```

MPI C' requires to make tasks non-blocking and register a callback that associated the event completion with a scheduling decision!

ULT Support through MPI Continuations

Example with MPI Continuations + OpenMP tasks

```
1  {
2    //rank 1
3    MPI_Request cont_req;
4    MPIX_Continue_init(&cont_req, MPI_INFO_NULL);
5    ...
6    #pragma omp task depend(out:value) shared(value, comm_started_flag) detach(event)
7    {
8      MPI_Request req;
9      MPI_Irecv(&value, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &req);
10     MPIX_Continue(&req, &release_event, (void *) event, MPI_STATUS_IGNORE, cont_req);
11     comm_started_flag = 1;
12   }
13
14   #pragma omp task depend(in:value) shared(value, comm_started_flag)
15   {
16     printf("Received: %d\n", value);
17   }
18
19   #pragma omp task
20   {
21     int flag = 0;
22
23     do {
24       if (comm_started_flag) {
25         MPI_Test(&cont_req, &flag, MPI_STATUS_IGNORE);
26       }
27       #pragma omp taskyield
28     } while(!flag);
29   }
30   #pragma omp taskwait
31 }
```

```
1  void release_event(MPI_Status *status, void *data) {
2    omp_event_handle_t event = (omp_event_handle_t)(uintptr_t) data;
3    omp_fulfill_event(event);
4  }
```

- [NEW] OpenMP *detach* marks an external completion event.
- [NEW] *Omp_fulfill_event* associates MPI events with OpenMP external event (runtime_associate(...)).

Solutions today

User-level Threading Support in MPI (>=4.x)

User-level Threading Support Through MPI Continuations (events)

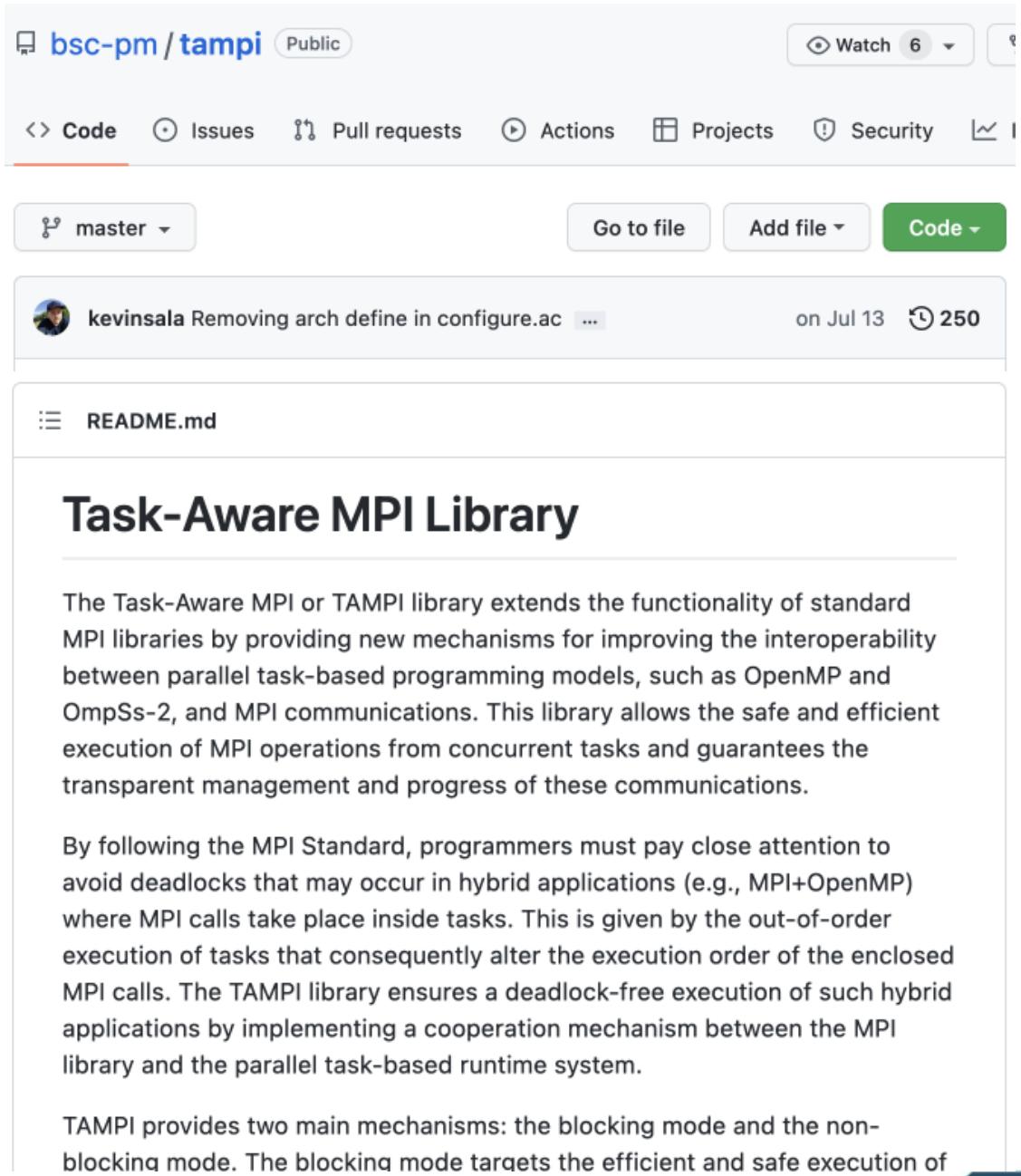
API Overloads for Proprietary Event Handling

Task-away MPI (BSC)

Adds the MPI_TASK_MULTIPLE execution mode

```
1   { //Rank 0
2   for (int n = 0; n < N; ++n) {
3     #pragma omp task depend(in: data[n]) // T1
4     {
5       MPI_Request request;
6       MPI_Issend(&data[n], 1, MPI_INT, 1, n, MPI_COMM_WORLD, &request);
7       TAMPI_Iwait(&request, MPI_STATUS_IGNORE);
8       // Data buffer cannot be reused yet!
9       // Other unrelated computation...
10    }
11    #pragma omp taskwait
12  }
```

```
14  { //Rank 1
15  for (int n = 0; n < N; ++n) {
16    #pragma omp task depend(out: data[n], statuses[n]) // T2
17    {
18      MPI_Request request;
19      MPI_Irecv(&data[n], 1, MPI_INT, 0, n, MPI_COMM_WORLD, &request);
20      TAMPI_Iwaitall(1, &request, &statuses[n]);
21      // Data buffer and status cannot be accessed yet!
22      // Other unrelated computation...
23    }
24
25    #pragma omp task depend(in: data[n], statuses[n]) // T3
26    {
27      check_status(&statuses[n]);
28      fprintf(stdout, "data[%d] = %d\n", n, data[n]);
29    }
30  }
31  #pragma omp taskwait
32  }
```



Summary and Outlook

Solutions exist for progress guarantees

- Blocking calls require polling (OMPI and MPICH ULT Support) or TAMPI
- Non-blocking can rely on callbacks (and splitting up tasks into posting and using, making it cooperative)

MPI Partix will branch off into MPI Threadx to benchmark hybrid applications

- OSU benchmark ports to OpenMP tasking and ULT libs, heat diffusion, ECP Proxies

Differences between techniques need to be quantified

- Latency exposure due to schedulers and scheduling queues

Refs:

[1] MPI Continuations: https://github.com/mpiwg-hybrid/hybrid-issues/files/7377623/mpi40-report-continuations.pdf

[2] MPI C' ref implementation:

[3] TAMPI: https://github.com/bsc-pm/tampi