Task-Based Programming at Scale: Challenges and New Approaches

ZIH Kolloquium
Joseph Schuchart
15.12.2022
Who am I?

- 2005–2012: Dipl-Inf. TU Dresden, ZIH
- 2012–2013: Oak Ridge National Laboratory
- 2013–2016: ZIH, EE-HPC
- 2016–2020: Dr.-Ing., Stuttgart University, HLRS
- Since 2020: University of Tennessee, Knoxville
  Innovative Computing Laboratory
What is ICL?

• Head: Hartwig Anzt (frmly Jack)
• Linear Algebra ([Sca]LAPACK, heFFT e, SLATE, Mixed precision computation, ...)
• Performance Analysis Tools
  • PAPI
• Distributed Computing
Distributed Computing Group (DISCO)

- Message Passing Interface
  - MPI Forum members
  - Open MPI: P2P & collective operations, datatypes, RMA
  - Fault Tolerance: User-Level Fault Mitigation (ULFM)
- Distributed Tasking:
  - PaRSEC: distributed task-based runtime system
  - DPLASMA: task-based replacement for ScaLAPACK
Task-Based Programming

Traditional MPI programs

- **Sequential** control flow
- Easier to write, harder to scale
- Limited latency hiding potential

Task-based programs

- Oversubscription of work
- Scheduler-managed control flow
  - Harder to reason about
- Significant **latency hiding** potential

Figure 10: Screenshot of Vampir visualizing a trace of the S3D application using 200,448 cores on Jaguar PF. User functions are shown in green, MPI operations in red, and activities of the measurement environment in yellow (file open), light blue (trace I/O) and dark blue (synchronization).
Levels of Task-Based Programming

Shared Memory

- All tasks local
- Communication between threads on the same process
- OpenMP, OmpSs, CUDA graphs
- Coupling with distributed models like MPI

```c
#pragma omp parallel master
for (step in 0:NUM_STEPS) {
    for (field in 0:NUM_LOCAL_FIELDS)
        #pragma omp task depend(inout: data[field], in: data[field-1])
        compute_field_in_step(data, field, step);
}
```

Distributed Memory

- Data-flow across process boundaries
- Distributed scheduling decisions
- PaRSEC, HPX, UPC++, Legion, StarPU, TTG, DASH, ...
- Different models for different types of applications

```c
for (step in 0:NUM_STEPS) {
    for (field in 0:NUM_GLOBAL_FIELDS)
        insert_task(
            &compute_field_in_step, data, field, step,
            INOUT(data[field]), IN(data[field-1]));
}
```
Part One

Coupling asynchronous programming models with MPI
Nonblocking MPI Operations

- MPI provides nonblocking P2P, Collective, RMA, I/O
- Completion detection through polling only
  - MPI_Test, MPI_Wait and friends
- Applications manage requests
  - Challenging and error-prone in asynchronous/irregular applications
- Prior proposals are **not portable**
  - ULT integration with MPI (Qthreads, Argobots)
  - TAMPI
Motivation: Communicating OpenMP Tasks

• Polling is insufficient
• Not all yields are created equal

```c
/* task that receives values */
double *vars;
#pragma omp task depend(out: vars)
{
    MPI_Request op_request;
    int flag;
    vars = malloc(sizeof(double)*NUM_VARS);
    MPI_Irecv(vars, NUM_VARS, MPI_DOUBLE, 0, TAG, comm, &op_request);
    do {
        MPI_Test(&op_request, &flag, MPI_STATUS_IGNORE);
        if (flag) {
            #pragma omp taskyield
        }
    } while (flag);
}

#pragma omp task depend(in: vars)
{
    compute_vars_from(vars, 0);
    free(vars);
}
```
Motivation: Communicating OpenMP Tasks

- Polling is insufficient
- Not all yields are created equal
Motivation: Communicating OpenMP Tasks

• Polling is insufficient
• Not all yields are created equal
• Alternative 1:
  • Central polling infrastructure
• Alternative 2:
  • **Return request to MPI**, for good

```c
/* task that receives values */
double *vars;
#pragma omp task depend(out: vars)
{
    MPI_Request op_request;
    int flag;
    vars = malloc(sizeof(double)*NUM_VARS);
    MPI_Irecv(vars, NUM_VARS, MPI_DOUBLE, 0, TAG, comm, &op_request);
    do {
        MPI_Test(&op_request, &flag, MPI_STATUS_IGNORE);
        if (flag) {
            #pragma omp taskyield
        }
    } while (!flag);
}

#pragma omp task depend(in: vars)
{
    compute_vars_from(vars, 0);
    free(vars);
}
```
Motivation: Communicating OpenMP Tasks

- Polling is insufficient
- Not all yields are created equal
- Alternative 1:
  - Central polling infrastructure
- Alternative 2:
  - Return request to MPI, for good

ONE DOES NOT SIMPLY
USE MPI IN OPENMP TASKS
The Hollywood Principle

- Requests are created and free’d by MPI
- Applications receive, store, and test requests
- **Observations:**
  - Requests are only a control device
  - Applications care about **completion of operations**
The Hollywood Principle

- Requests are created and free’d by MPI
- Applications receive, store, and test requests

**Observations:**
- Requests are only a control device
- Applications care about completion of operations

- Let’s stop managing requests and focus on operations
  - Return requests to MPI
  - Wait for a call signalling completion
The Hollywood Principle

- Requests are created and free’d by MPI
- Applications receive, store, and test requests
- **Observations:**
  - Requests are only a control device
  - Applications care about **completion of operations**

- Let’s stop managing requests and focus on operations
  - Return requests to MPI
  - Wait for a call signalling completion

*Don’t call us, we’ll call you back.*
Introducing: MPI Continuations

- Continuations are attached to one or more operations
- MPI takes back request ownership*
- Invokes callback once the operation is complete

**Inputs:**
- Flags
- Callback function pointer
- User-data pointer
- Status (optional)
- Continuation request

```c
int recv_completion_cb(int rc, void *user_data)
{
    omp_fulfill_event((omp_event_t) user_data);
    return MPI_SUCCESS;
}
```

```c
/* task that receives values */
double *vars;
#pragma omp task depend(out: vars) detach(event)
{
    MPI_Request op_request;
    vars = malloc(sizeof(double)*NUM_VARS);
    MPI_Irecv(vars, NUM_VARS, MPI_DOUBLE, 0, TAG, comm, op_request);
    MPI_Continue(&op_request, &recv_completion_cb, event, 0,
                  MPI_CONT_RECVBUF_VOLATILE,
                  MPI_STATUS_IGNORE, cont_request);
}
/* task processing values, executed once the receiving task’s
dependencies are released */
#pragma omp task depend(in: vars)
{
    compute_vars_from(vars, 0);
    free(vars);
}
```
Introducing: MPI Continuations

- Continuations are attached to one or more operations
- MPI takes back ownership
- Invokes callback once the operation is complete

**Inputs:**
- Flags
- Callback function pointer
- User-data pointer
- Status (optional)
- Continuation request

```c
int recv_completion_cb(int rc, void *user_data)
{
    omp_fulfill_event((omp_event_t) user_data);
    return MPI_SUCCESS;
}

/* task that receives values */
double *vars;
#pragma omp task depend(out: vars)
detach(event)
{
    MPI_Request op_request;
    vars = malloc(sizeof(double)*NUM_VARS);
    MPI_Irecv(vars, NUM_VARS, MPI_DOUBLE, 0, TAG, comm, &op_request);
    MPI_Continue(&op_request, &recv_completion_cb, event, 0,
                  MPI_CONT_REQBUF_VOLATILE,
                  MPI_STATUS_IGNORE, cont_request);
}

/* task processing values, executed once the receiving task's dependencies are released */
#pragma omp task depend(in: vars)
{
    compute_vars_from(vars, 0);
    free(vars);
}
```
Continuations Control Flow

Application

MPI

MPI_Isend

Request

op₁

Request
Continuations Control Flow

Application

- MPI_Isend
- MPI_Continue

MPI

- Request
- Continuation

Op1

Op2

C1
Continuations Control Flow

Application

- MPI_Isend
- MPI_CONTINUE

MPI

- Request
- Continuation

Op1

Op1
C1

Op2

Op2
C2

Op2

Op2
Continuations Control Flow

Application

- MPI_Isend
- MPI_Continue
- MPI_Irecv
- MPI_Continue
- MPI_Wait

MPI

- Request
- Continuation
- Request
- Continuation
- C1
- C2
- Op_1
- Op_2
Attaching a Continuation: MPI_Continue

- **Attaches** a continuation to **one** operation
- Flags control behavior
- Status filled before callback is invoked

```c
MPI_CONTINUE(op_request, cb, cb_data, flags, status, cont_request)
```

- **INOUT** `op_request`  
 operation request (handle)
- **IN** `cb`  
  callback to be invoked once the operation is complete (function)
- **IN** `cb_data`  
  pointer to a user-controlled buffer
- **IN** `flags`  
  flags controlling aspects of the continuation (integer)
- **IN** `status`  
  status object (array of status)
- **IN** `cont_request`  
  continuation request (handle)

**C binding**

```c
int MPI_Continue(MPI_Request *op_request, MPI_Continue_cb_function cb, 
                 void *cb_data, int flags, MPI_Status *status, 
                 MPI_Request cont_request)
```
Attaching a Continuation: MPI_Continueall

- **Attaches** a continuation to **multiple** operations
- Flags control behavior
- Status filled before callback is invoked

```c
int MPI_Continueall(int count, MPI_Request array_of_op_requests[],
                     MPI_CONTINUE cb_function cb, void *cb_data, int flags,
                     MPI_Status array_of_statuses[], MPI_Request cont_request)
```
Callback Functions

**Inputs:**
- Error code: MPI_SUCCESS (or error if operation failed)
- User-data provided during creation

**Returns:**
- Error code: MPI_SUCCESS or error code
- May call MPI procedures
  - Blocking procedures discouraged

```c
typedef int MPI_Continue_cb_function(int error_code, void *user_data);

ABSTRACT INTERFACE
SUBROUTINE MPI_Continue_cb_function(error_code, user_data, ierror)
  INTEGER :: error_code
  INTEGER(KIND=MPI_ADDRESS_KIND) :: user_data
  INTEGER, OPTIONAL :: ierror
SUBROUTINE MPI_CONTINUE_CB_FUNCTION(ERROR_CODE, USER_DATA, IERROR)
  INTEGER ERROR_CODE, IERROR
  INTEGER(KIND=MPI_ADDRESS_KIND) USER_DATA
```
Evaluation: ExaHype

- Compressible Navier Stokes equations for cloud simulation
- Dynamic load balancing via task migration
- Continuations simplified request handling, leading to 25% higher offload rates and improved balancing

Reference implementation  
Using MPI Continuations
Part One: Summary

• MPI must better support asynchronous programming models
• Requests are merely a device
• Callbacks are flexible and allow for fast reaction to state changes
• Fine-grain control over execution behavior
Part Two

A new task model for distributed memory task programming
Why distributed task models?

- Shared memory tasking models have serious limitations
  - Local-only scheduling decisions
  - Communication managed by user
  - Separation problematic

- Distributed Models provide
  - Managed communication
  - Global view scheduling decisions
  - Minimal synchronization
Distributed Task Graph Requirements

1. Task graph discovery (DAGs of tasks)
2. Data flow (moving data between processes)
3. Task Execution (along the critical path)
Distributed Task Graph Discovery

- **Insert task:**
  - Sequential discovery of global task graph on all processes
  - **Limited scalability**
- **Continuation-based** programming:
  - Explicit spawning of activities at places
  - **Handle for each data** flowing through the graph
- **Abstract Task Graphs**
  - Compact representation of potential data flow
  - **Scoped discovery** of tasks
TTG: Template Task Graphs

- Abstract task graph unfolds into DAG during execution
  - **Template Tasks**: instantiated at execution
  - **Terminals**: input/output points
  - **Edges**: connecting input/output terminals
- Data-dependent selection of successors

TTG: Template Task Graphs

- Abstract task graph unfolds into DAG during execution
  - **Template Tasks**: instantiated at execution
  - **Terminals**: input/output points
  - **Edges**: connecting input/output terminals
  - Data-dependent selection of successors
Data Moves through the Graph

- Tasks send data to successors
  - Become eligible for execution once all input terminals received value
  - Unaware of predecessors
- Scalable distributed task discovery
- Flexible backend implementation
  - Available: PaRSEC, MADNESS
  - MPI main communication backend
Data Movement

- Data traverses through unfolding task graph
- Goal: minimize number of data copies
  - Utilize C++ `move` and `const` semantics
  - Avoid copying data if we know its is immutable
- Zero-copy transfer mechanism
  - Serialize meta-data, copy payload directly

```c++
void taskfn(const TaskID& task_id, const MatrixTile& input) {
    MatrixTile output = compute_output_tile(input);
    send<0>(task_id, output); // new copy required
    send<1>(task_id, move(output)); // no copy due to move
    send<2>(task_id, input); // no copy as input is const
}
```
TTG Task Functions

- Function object invoked once all inputs are satisfied.
- Key is optional (for task templates with single task instance)

```cpp
[](const keyT &k) {
    ttg::print("This is task B(" , k, ")");
}

[](const keyT &k, const T& val) {
    ttg::print("This is task B(" , k, ", " val, ")");
}

[](const keyT &k, T&& val) {
    ttg::print("This is task B(" , k, ", " val, ")");
}

[](const keyT &k, auto& val) {
    ttg::print("This is task B(" , k, ", " val, ")");
}

[](const keyT &k, auto&& val) {
    ttg::print("This is task B(" , k, ", " val, ")");
}
```

- control flow (= flow of "void" data)
- immutable data
- mutable data
- immutable generic data
- mutable generic data

We need C++ introspection!
Example: Data Movement

- Data is sent or broadcast through the graph
- POTRF kernel:
  - Invoke kernel
  - Populate successor keys
  - Broadcast keys and data

```cpp
tgt::Edge<Key1, MatrixTile<double>> syrk_potrf("syrk_potrf");
tgt::Edge<Key2, MatrixTile<double>> potrf_trsm("potrf_trsm");
auto f = [=](const Key1& key, MatrixTile<T>& tile_kk) {
    const int K = key.K;
    /* invoke POTRF kernel */
    lapack::potrf(lapack::Uplo::Lower, tile_kk.rows(), tile_kk.data(), tile_kk.rows());
    /* send the tile TRSMs and SYRK */
    std::vector<Key2> keylist;
    keylist.reserve(A.rows() - K);
    for (int m = K+1; m < A.rows(); ++m) {
        keylist.push_back(Key2(m, K));
    }
    tgt::broadcast<0, 1>(std::make_tuple(Key2(K, K), keylist), std::move(tile_kk));
    tgt::make_tt(f, tgt::edges(input), tgt::edges(output_result, output_trsm), "POTRF",
                 {"tile_kk", {"output_result", "output_trsm"}});
Cholesky Factorization: Weak Scaling

- Hawk, 1 – 256 nodes
- Matrix: 30k per node, tiles size 512

Performance of TTG matches DPLASMA
Task Graph Composition

- Assume N consecutive functions implemented using the same task programming model
  - N forks and joins without composition
  - Write-back to data structures instead of direct flow
  - Gene Amdahl says that’s bad

https://upload.wikimedia.org/wikipedia/commons/1/1a/Gene_Amdahl_on_a_classic_grey_Ferguson_tractor_at_Amdahl.JPG

https://en.wikipedia.org/wiki/Amdahl%27s_law#/media/File:AmdahlsLaw.svg
Example: Cholesky Matrix Inversion

- **Cholesky Factorization (POTRF) followed by matrix inversion**
  - Given $A$, compute $A^{-1}$
  - $A$: Hermitian positive-definite matrix

- **Inversion: Given $L$ from POTRF**
  - Compute $L^{-1}$ from $L$ (TRTRI)
  - Compute $A^{-1} = (L^{-1})^T L^{-1}$ (LAUUM)
  - $\text{POTRI} = \text{TRTRI} \oplus \text{LAUUM}$

- **POINV = POTRF \oplus \text{POTRI} = POTRF \oplus \text{TRTRI} \oplus \text{LAUUM}**
Connecting Graphs: Edges as Composition Devices

- Use Edges to connect algorithm graphs
  - Algorithms as black bloxes
  - Data flows in through Edge, comes out through Edge
Task Graph Composition at Work
POINV Composition

- 16 nodes on Hawk, 64 threads each
- Full composition beneficial for small tile sizes
  - Fine-grain composition helps hide communication latency
  - Beats both DPLASMA (based on PaRSEC PTG) and SLATE

POTRI: Comparison with Chameleon

- 128 nodes on Hawk
- Chameleon (v1.1.0, using StarPU 1.3.9)
- POTRI : TRTRI ⊕ LAUUM
- TTG performance benefits
  - Depth-first execution
  - Parallel distributed task discovery
Part 2: Summary

• Global task discovery limits scalability
• Template Task Graphs provide
  • **Scoped task discovery**, with C++ semantics
  • Without the baggage of futures
• Edges represent sets of values and simplify composition
• Future work includes
  • Integrated device support (using C++ coroutines)
  • Porting of MADNESS/TiledArray on top of TTG
Part Three

Why are asynchronous models so hard to use?
Challenges and Benefits of Tasks

• S3D using MPI vs Legion

Figure 10: Screenshot of Vampir visualizing a trace of the S3D application using 200,448 cores on Jaguar-PF. User functions are shown in green, MPI operations in red, and activities of the measurement environment in yellow (file open), light blue (trace I/O) and dark blue (synchronization).

Challenges of Programming in Tasks

- Porting existing applications to new programming models is a significant investment (>1PY)
- Higher level abstractions → more constraints:
  - Less flexible than MPI
  - Likely to run into (non-MPI) barriers at some point
- Clear separation of concerns with MPI/OpenMP
- Flag-ship task applications vs broad acceptance?
  - Octotiger (HPX)
  - S3D (Legion)
  - ExaGeoStat (PaRSEC)
A Plethora of Models

- Which is the right model for my application?
- How long will that model be supported?
- How much flexibility do I need?
- How many constraints can I accept?
- Can I maintain the code once the PhD student is gone?
The Complexity of Models

- Are developers able to grasp the complexity of fully asynchronous programming?
  - Just a matter of teaching?
- We typically think sequentially
  - Task-based programming like juggling
- We need better tools for debugging & performance analysis
Conclusions

• Task-based programming comes with significant benefits and challenges
• We need better **support from MPI** to support async models
• Better **tool support** for application & runtime developers
• There is (likely) no single model to rule all applications
• But: Can we establish interoperability between models?
  • Past efforts had limited success
A Shameless Plug

• ICL is hiring
  • PhD students
  • PostDocs
  • Visitors (3-12 months)
• Talk to me if you’re interested :)

51
Discussion