Global Task Data Dependencies for PGAS Applications

<Author Names Hidden for Double-blind Review>

Background

<Project> provides an abstraction of the **Partitioned Global Address Space** (PGAS) programming paradigm based on C++11 by providing distributed data structures.

The PGAS paradigm decouples data transfer and synchronization and thus requires explicit synchronization primitives. So far, <Project> relied on **collective operations for synchronization**.

```
Array<int> arr(size());
// fill local part
arr.local[0] = myid();
// wait for everyone to finish
barrier();
// read value from unit 0
auto root_id = arr[0];
```

Figure 1: Use of collective synchronization in <Project>

Research Goals

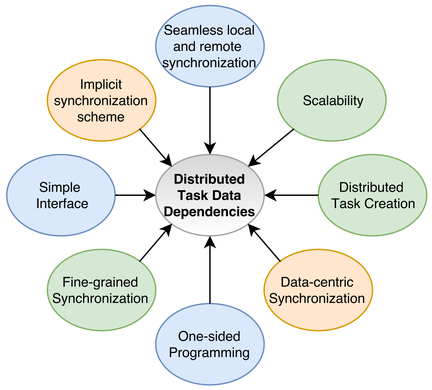
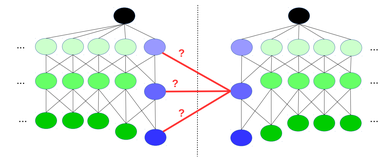
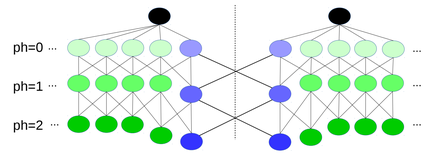


Figure 2: Goals and requirements for the proposed distributed task data dependencies.

Methodology



Above: distributed task graph of a regular 1D stencil application running on 2 processes. **Local dependencies are created in serial order** and are thus trivial to match (depicted as black lines). Dependencies **across process boundaries** (red lines) cannot be matched reliably without additional information from the user as there may be **multiple tasks with output dependencies** on the same memory location.



Our proposal: Task Phases

Task phases serve as a **logical clock** for tasks and task dependencies that provide the necessary information to reliably match dependencies across process boundaries. Similar to `barriers`, task phases restore *happens-before* relations between tasks, i.e., any task that produces data in memory location L_k in phase ph_{i-1} has to be executed before tasks in phase ph_i that depend on L_k.

Distributed Task Data Dependencies

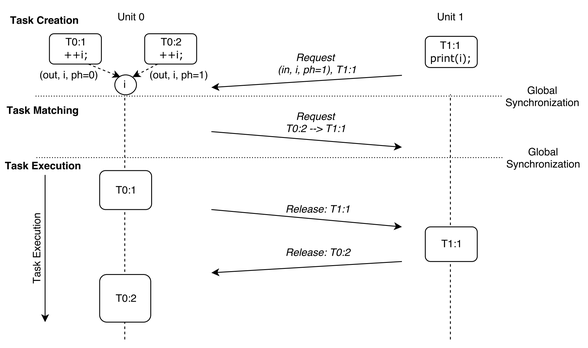


Figure 3: Communication scheme between schedulers on two processes. Dependencies are handled by the process that "owns" the referenced memory location.

Local scheduler handle locally created tasks as well as the dependencies on the local portion of the global memory space (i in Figure 3). Upon encountering a remote dependency, the local scheduler informs the respective scheduler, which **coordinates** the global execution of tasks with dependencies on this memory location. The scheduler also handles data hazards by observing WAR and WAW dependencies.

Regardless of the number of tasks **only two global synchronizations** are required to ensure that 1) all user-specified dependencies and 2) the resulting data hazards have been communicated. The task execution happens fully asynchronous.

Implementation and Evaluation

```
for (size_t k = 0; k < num_blocks; ++k) {
  if (mat.block(k,k).is_local()) {
    async([=](){ potrf(matrix.block(k,k)); },
      out(mat.block(k,k)));
  }

  // advance to next phase
  async_barrier();
  for (int i = k+1; i < num_blocks; ++i)
    if (mat.block(k,i).is_local())
      async([=](){
        trsm(cache[k][k], matrix.block(k,i)); },
        copyin(mat.block(k,k), cache[k][k]),
        out(mat.block(k,i)));

  // advance to next phase
  async_barrier();
  for (size_t i = k+1; i < num_blocks; ++i) {
    for (size_t j = k+1; j < i; ++j)
      if (mat.block(j,i).is_local())
        async([=](){
          gemm(cache[k][i], cache[k][j], mat.block(j,i)); },
          copyin(mat.block(k,i), cache[k][i]),
          copyin(mat.block(k,j), cache[k][j]),
          out(mat.block(j,i)));

    if (mat.block(i,i).is_local())
      async([=](){ syrk(cache[k][i], mat.block(i,i)); },
        copyin(mat.block(k,i), cache[k][i]),
        out(mat.block(i,i)));
  }
  // advance to next phase
  async_barrier();
}
// wait for all tasks to execute
complete();
```

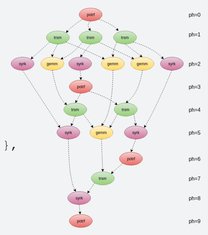


Figure 4: Blocked Cholesky factorization implemented using task data dependencies in <Project> (simplified).

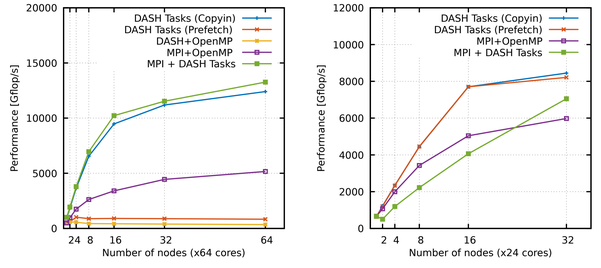


Figure 5: Blocked Cholesky Factorization (N=64k, BS=512) on Oakforest PACS (KNL+Omnipath, left) and a NEC Linux cluster (Haswell+Infiniband, right)

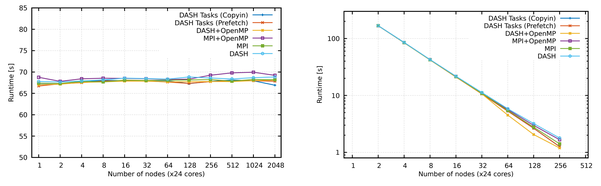


Figure 6: Weak scaling (4800 rows/node, left) and strong scaling (48000 rows, right) of a simple 2D stencil with 409600 elements per row on a Cray XC40.

Current Status

Distributed data dependencies fulfilling the requirements depicted in Figure 2, supporting:

- Seamless global **`in`/`out` dependencies**
- Flexible **`copyin` dependency** (implemented using `get` or `send`/`recv`)
- Task scheduling **priorities**
- Global/local **task cancellation**
- Rescheduling **task-yield**
- Intermediate dependency matching
- MPI-based **active messages** for inter-scheduler communication
- Poll-free waiting for data transfers

Future Work

- Support for advanced dependencies types (`commutative`, `concurrent`)
- Performance optimizations
- Tool support
- Porting target applications

References

<References removed for double-blind review>

Contact Information

Web	<hidden>
Email	<hidden>
Phone	<hidden>
Github	<hidden>