SuperGlue and DuctTEiP: Using data versioning for dependency-aware task-based parallelization

Elisabeth Larsson Martin Tillenius Afshin Zafari

UPMARC Workshop on Task-Based Parallel Programming
September 28, 2012

UPMARC
Uppsala University

Outline

- SuperGlue: The Shared Memory Framework
- DuctTEiP: The Distributed Memory Framework
- The shallow water equations on the sphere

Context and motivation

Scientific Computing & Computational Science

- Performance is crucial
- Portability is desired
- Programming must be facilitated

Task Based Abstractions

- **Mapping to hardware is hidden**
  - Portability
- **Dependency Management and Scheduling**
  - Performance
  - Ease-of-programming
  - Correctness

The SuperGlue Task Universe

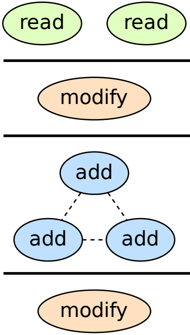
- Dependencies are deduced at run-time
- One queue per worker thread for ready tasks
- Waiting tasks are queued at data
- Scheduling with rules to promote locality
- Task stealing for load balancing

Data Versioning

Example

8 tasks accessing the same handle x:

read x, read x, modify x, add x, add x, add x, modify x

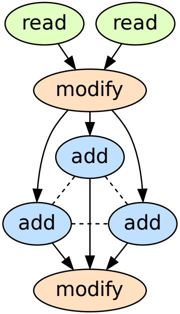


Requires version 0
(Run all at once)

Requires version 2

Requires version 3
(Any order)
(One at a time)

Requires version 6



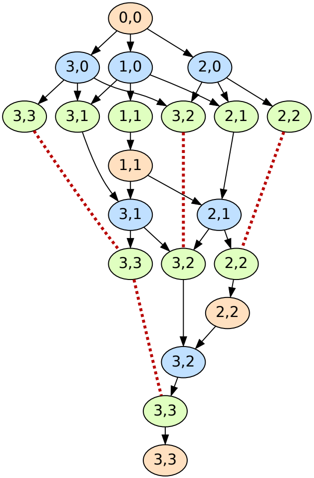
Graph View
(Not a DAG)

Example Uses: Cholesky

Cholesky

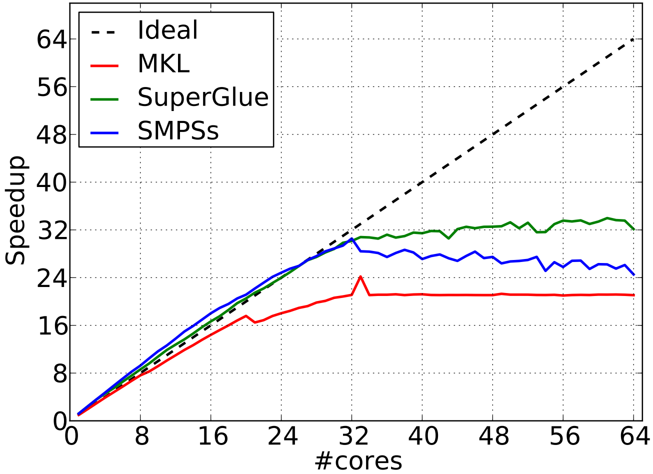
```
for (int i = 0; i < N; i++) {
  potrf(i); // Aii = Cholesky(Aii)

  for (int j = i+1; j < N; j++)
    trsm(i,j); // Aji = AjiAii^-T

  for (int j = i+1; j < N; j++)
    for (int k = i+1; k <= j; k++)
      gemm(i,j,k); // Ajk = Ajk - AjiAki^T
}
```

The **gemm()** tasks can be reordered.



N=4

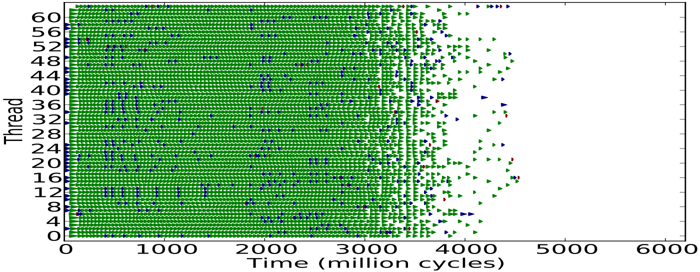
Cholesky Speedup



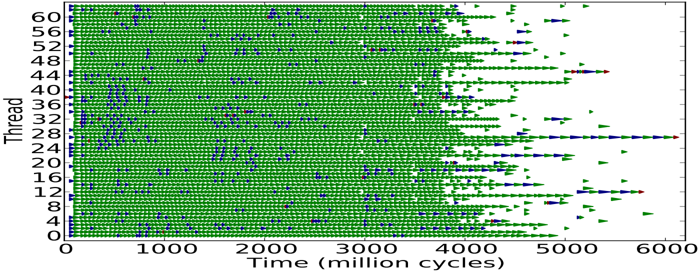
AMD Bulldozer, 2 cores share 1 FPU
8192 x 8192 Matrix in blocks of 256 x 256.

Cholesky Execution Traces: 64 cores

Our:



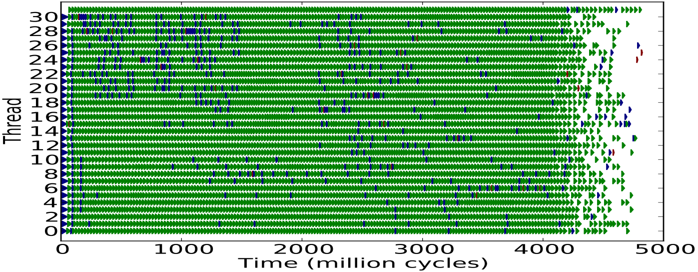
SMPSs:



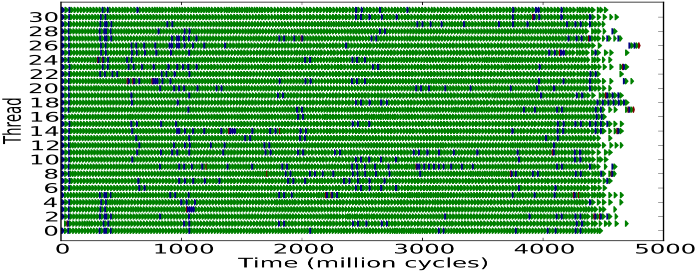
8192 x 8192 Matrix in blocks of 256 x 256.

Cholesky Execution Traces: 32 cores

Our:



SMPSs:



8192 x 8192 Matrix in blocks of 256 x 256.

Features Of DuctTeip

- Dependencies are deduced at run-time
- Tasks are hierarchical

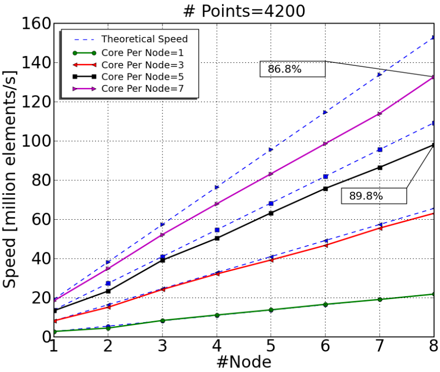
- One MPI process per node passes ready tasks to SuperGlue for local execution.
- When a task needs remote data, a listener is sent to the data host node
- Ready data versions are sent to nodes that have placed listeners

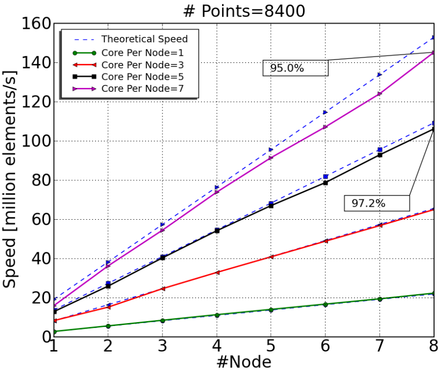
- Scheduling of the global tasks is right now static
- Task stealing between nodes has not been implemented

Strong scalability results

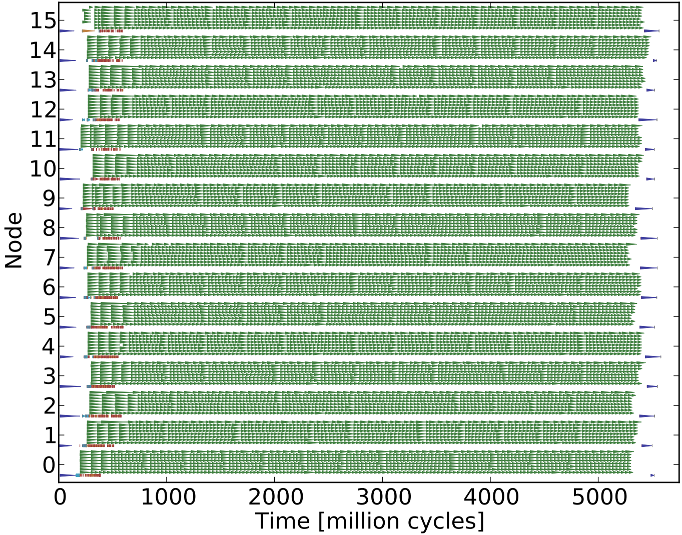
Sample problem $A_{ij} = f(x_i, x_j)$

Building a (distributed) matrix from a distributed vector.





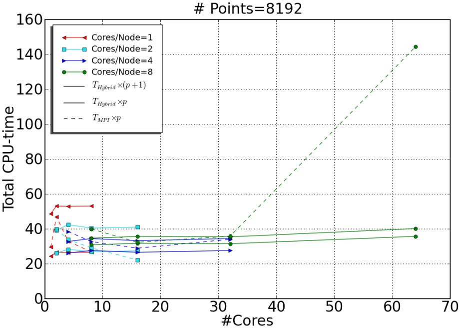
Execution Trace: DuctTeip Hybrid Model

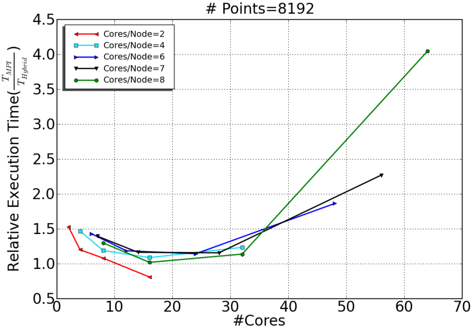


Each node consists of two Quad-core Intel Xeon 5520.

Speedup: 97.7x (76%) on 128 cores (over best serial speed)

Comparison Hybrid Versus Pure MPI





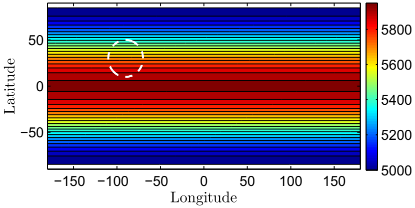
- # Messages $\propto$ # (MPI processes)^2
- Hybrid $\Rightarrow$ Dynamical load balancing within a node

Future development of DuctTeip

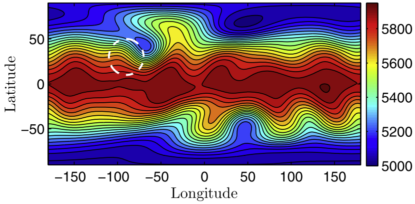
- High-level communication operations
  - Related messages can be coordinated.
  - Implementation of global reductions e.g.
- Load-balancing at the global level
  - Scheduling principles
  - Task-stealing between nodes, including tasks, listeners, data.
- Heterogeneity

Application: Global Climate Simulations

Shallow water simulations on a sphere
Test Case: Flow over an isolated mountain



Day 0



Day 15

N. Flyer, E. Lehto, S. Blaise, G.B. Wright, A. St-Cyr, **A guide to RBF-generated finite differences for nonlinear transport: Shallow water simulations on a sphere**, J. Comput. Phys. 231 (2012) 4078-4095.

Results (on 4 cores)

Results

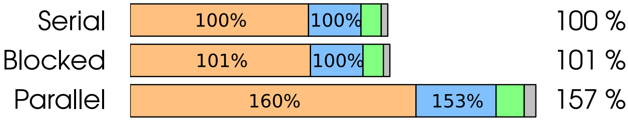
- Achieved parallelism: 3.5
- Speedup over best serial: 2.3 x
- Speedup over MATLAB: 7.3 x

Total Execution Time Comparison (million cycles)

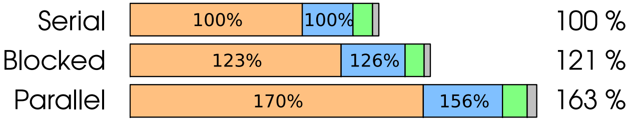
	MATLAB	**C++**	**Blocked**	**Parallel**
Row	12062	3710	3762	1644
2D	12062	3710	4475	1699

Results (on 4 cores)

Increase in execution time: Row (7 blocks)



Increase in execution time: 2D (3 $\times$ 3 blocks)



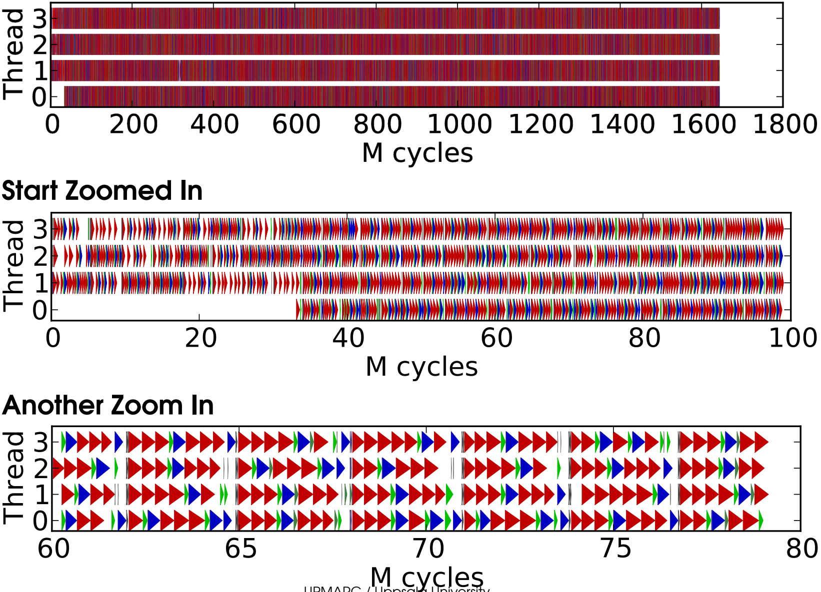
Increased task management overhead for blocking was $< 1\%$.

Results (on 4 cores)

Full Run

Start Zoomed In

Another Zoom In



Conclusions

Dependency-Aware Task Based Models are:

- Efficient
- User friendly

Version Driven Dependency Management has nice properties:

- Easy, Efficient, and Flexible
- No global view:
  - A tasks only knows the handles it accesses
  - A handle only know tasks waiting for it

The hierarchical SuperGlue + DuctTeip hybrid performs well:

- Similar user programming effort as for SuperGlue
- Good scalability for single and multiple nodes
- Using a hybrid approach can give performance benefits

Thank you!

Questions?

Code

```
#include "tasklib.hpp"
#include "options/defaults.hpp"
#include "options/prioscheduler.hpp"

// Custom handle type to include indices
template<typename Options>
struct MyHandle : public Handle_<Options> {
    size_t i, j;
    void set(size_t i_, size_t j_) { i = i_; j = j_; }
    size_t geti() { return i; }
    size_t getj() { return j; }
};

struct Options : public DefaultOptions<Options> {
    typedef MyHandle<Options> HandleType; // Override handle type
    typedef PrioScheduler<Options> Scheduler; // Override scheduler
    typedef void TaskPriorities; // Enable task priorities
};
```

Code

```
struct gemm : public Task<Options, 3> {
    gemm(Handle<Options> &h1, Handle<Options> &h2,
         Handle<Options> &h3) {
        // register data accesses to manage, with direction
        registerAccess(0, ReadWriteAdd::read, &h1);
        registerAccess(1, ReadWriteAdd::read, &h2);
        registerAccess(2, ReadWriteAdd::add, &h3);
    }
    void run() {
        Handle<Options> &h1(getAccess(0).getHandle());
        Handle<Options> &h2(getAccess(1).getHandle());
        Handle<Options> &h3(getAccess(2).getHandle());

        double *a(Adata[h1->geti()*DIM + h1->getj()]);
        double *b(Adata[h2->geti()*DIM + h2->getj()]);
        double *c(Adata[h3->geti()*DIM + h3->getj()]);

        double DONE=1.0, DMONE=-1.0;
        dgemm("N", "T", &nb, &nb, &nb, &DMONE, a, &nb, b, &nb, ...
    }
    int getPriority() const { return 0; }
};
```

Code

```
static void cholesky(const size_t numBlocks) {
    ThreadManager<Options> tm;    // Starts the system
    // Create handles, and set the custom indices
    Handle<Options> **A = new Handle<Options>*[numBlocks];
    for (size_t i = 0; i < numBlocks; ++i) {
        A[i] = new Handle<Options>[numBlocks];
        for (size_t j = 0; j < numBlocks; ++j)
            A[i][j].set(i, j);
    }

    // Main code: Generate tasks
    for (size_t j = 0; j < numBlocks; j++) {
        for (size_t k = 0; k < j; k++)
            for (size_t i = j+1; i < numBlocks; i++)
                tm.addTask(new gemm(A[i][k], A[j][k], A[i][j]), i);
        for (size_t i = 0; i < j; i++)
            tm.addTask(new syrk(A[j][i], A[j][j]), j);
        tm.addTask(new potrf(A[j][j]), j);
        for (size_t i = j+1; i < numBlocks; i++)
            tm.addTask(new trsm(A[j][j], A[i][j]), j);
    }

    tm.barrier();
}
```

Generalization: More types

Can have several different reorderable access types.

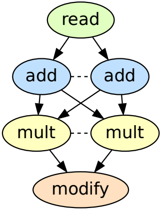
Example: **read**, **modify**, **add**, **mult**

Can be reordered:

- read - read
- add - add
- mult - mult

Example Sequence

read x, add x, add x, mult x, mult x, modify x



Generalization: More types

Limitation: Can only reorder accesses of same type.

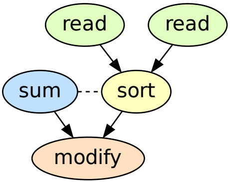
Example: **read**, **modify**, **sort**, **sum**

Can be reordered:

- read - read
- read - sum
- sort - sum

Example Sequence

read x, sum x, read x, sort x, modify x

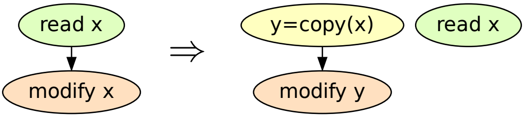


- **`Sort`** must wait for both **`reads`** to finish
- **`Sort`** need not wait for the **`sum`** task

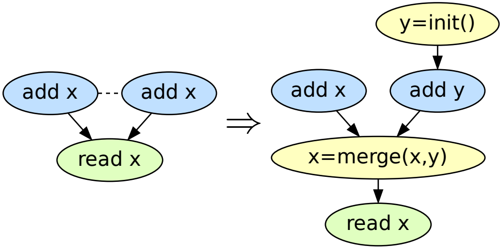
This requires more than one version counter per handle.

Renaming

Avoid write-after-read dependencies by duplicating data:



So far only automatic for **add** accesses.



Implementation

- Handles can keep a temporary copy
- Attach or merge copies when task finishes
- Lazy merge

More Code

```
void evalForce(ThreadManager<Options> &tm,
               particle_t *particles, handle_t *part,
               vector_t *forces, handle_t *forc,
               const size_t blockSize, const size_t numBlocks) {

    for (size_t i = 0; i < numBlocks; ++i) {
        tm.addTask(new EvalWithinTask(
            &particles[i*blockSize], &part[i],
            &forces[i*blockSize], &forc[i],
            blockSize), i);
    }
    for (size_t i = 0; i < numBlocks; ++i) {
        for (size_t j = i + 1; j < numBlocks; ++j)
            tm.addTask(new EvalBetweenTask(
                &particles[i*blockSize], &part[i],
                &particles[j*blockSize], &part[j],
                &forces[i*blockSize], &forc[i],
                &forces[j*blockSize], &forc[j],
                blockSize), i);
        }
    }
```

More Code

```
class EvalBetweenTask : public Task<Options, 4> {
private:
    particle_t *p0_, *p1_;
    vector_t *f0_, *f1_;
    size_t sliceSize_;
public:
    EvalBetweenTask(particle_t *p0, handle_t *hp0,
            particle_t *p1, handle_t *hp1,
            vector_t *f0, handle_t *hf0,
            vector_t *f1, handle_t *hf1,
            size_t sliceSize) {
        // register accesses
        registerAccess(0, ReadWriteAdd::read, hp0);
        registerAccess(1, ReadWriteAdd::read, hp1);
        registerAccess(2, ReadWriteAdd::add, hf0);
        registerAccess(3, ReadWriteAdd::add, hf1);
        // store data needed to execute the task
        p0_ = p0; p1_ = p1; f0_ = f0; f1_ = f1;
        sliceSize_ = sliceSize;
    }
    virtual void run() { ... }
};
```