

LibVerticalParallelization

Vertical Parallelization using memory access classification

Maxime Mogé – Inria teams CAMUS and MIMESIS

Project partners: J. Gustedt, P. Clauss, S. Cotin, H. Courtecuisse

03/08/2018

1 – General idea

A detailed description of the underlying theory as well as the implementation can be found in « Jens Gustedt, Maxime Mogé. Memory access classification for vertical task parallelism. [Research Report] RR-9182, Inria Nancy - Grand Est. 2018, pp.1-20. »

Target programs should have the following properties:

- They have an outer iteration loop (e.g. time loop).
- Each iteration consists of multiple computing tasks with a constant data access pattern over the iterations.

This situation can be found in FEM codes: computations on a mesh with unchanged topology at each iteration

Using the assumption of a constant data access pattern, we detect dependencies and derive an implicit scheduling at runtime using Ordered Read-Write Locks (ORWL). We use the EiLck library that implements this concept of ORWL.

With `libVerticalParallization` (which corresponds to the C++ implementation described in the paper), the computing tasks should be enclosed in a function call.

2 – Sofa implementation

To use `libVerticalParallization` in **Sofa**, very few changes are necessary :

- Compile using `-std=c++14`
- Define a custom parallel version of `Visitor::for_each_r`
- Define a flag in Simulation to allow parallel execution of current step
- Use `DataVectorWrapper` to overload `operator[]` for the data vector that are shared between the tasks (or for the all data vector). This can be done in a plugin if one defines a new vector type (ex: `ParallelVec3d`)
- Set the size of the meta-steps / parallel phases (the number of memory accesses in a meta-step)

The key changes are in `Visitor::for_each_r`.

Original sequential code:

/// Helper method to enumerate objects in the given list. The callback gets the pointer to node

```
template < class Visit, class VContext, class Container, class Object >
Visitor::Result for_each_r(Visit* visitor, VContext* ctx, const Container& list,
Visitor::Result (Visit::*fn)(VContext*, Object*))
{
    Visitor::Result res = Visitor::RESULT_CONTINUE;
    for (typename Container::iterator it=list.begin(); it != list.end(); ++it)
    {
        typename Container::pointed_type* ptr = &(*it);
        if(testTags(ptr))
        {
            debug_write_state_before(ptr);
            ctime_t t=begin(ctx, ptr);
            std::cout << "--for_each_r launch task" << std::endl;
            res = (visitor->*fn)(ctx, ptr);
            end(ctx, ptr, t);
            debug_write_state_after(ptr);
        }
    }
    return res;
}
```

Parallel code using libVerticalParallization:

/// Helper method to enumerate objects in the given list. The callback gets the pointer to node

```
template < class Visit, class VContext, class Container, class Object >
Visitor::Result for_each_r_para(Visit* visitor, VContext* ctx, const Container& list,
Visitor::Result (Visit::*fn)(VContext*, Object*))
{
    Result (*fn_wrapper)(Visit*,VContext*,Object*) = reinterpret_cast< Result (*)
(Visit*,VContext*,Object*)>(visitor->*fn);
    //Parallel section definition
    std::tuple< std::uintptr_t, std::uintptr_t, std::uintptr_t, const char* >
psid( (std::uintptr_t) ctx, (std::uintptr_t) fn_wrapper, (std::uintptr_t) &list, visitor-
>getClassName());
    std::vector< std::function<Result()*>* > computefunctions;
    if( sofa::simulation::sofaParallelization::parallelExecAllowed )
    {
        sofa::verticalparallelization::startParallelization< std::tuple< std::uintptr_t,
std::uintptr_t, std::uintptr_t, const char* > >>( psid );
    }
    sofa::verticalparallelization::beginParallelSection< Result, std::tuple< std::uintptr_t,
std::uintptr_t, std::uintptr_t, const char* > >>( psid );
    int counter = 0;
    Visitor::Result res = Visitor::RESULT_CONTINUE;
    for (typename Container::iterator it=list.begin(); it != list.end(); ++it)
    {
        typename Container::pointed_type* ptr = &(*it);
        if(testTags(ptr))
        {
            if( sofa::simulation::sofaParallelization::parallelExecAllowed )
            {
                //Task creation
                computefunctions.push_back( new
std::function<Result()*>( std::bind(fn_wrapper,visitor,ctx,ptr ) ) );
                sofa::verticalparallelization::runTask< Result >( *(computefunctions.back()),
res );
            }
            else
            {
                res = fn_wrapper(visitor,ctx,ptr);
            }
        }
    }
    sofa::verticalparallelization::endParallelSection< Result, std::tuple< std::uintptr_t,
std::uintptr_t, std::uintptr_t, const char* > >>( psid, res );
    for (std::vector< std::function<Result()*>* >::iterator it = computefunctions.begin()
```

```

        ; it != computefunctions.end() ; ++it)
    delete *it;
    return res;
}

```

In directory `sofaplugin` are the file needed to use `libVerticalParallelization` in `Sofa`. It consists in

1. a plugin `ParallelVectors`: It defines a new vector type (`ParallelVec3d`, etc.) that is just a wrapper around `Vec3d` with operator[] overloaded (using `libverticalparallelization::DataWrapper`) + the instantiation of all necessary classes.
2. a slightly modified version of `Simulation.cpp`: creation of the `ThreadPool`, set flags to enable parallel execution when initialization steps are done
3. `Visitor.h`: function `for_each_r_para` to execute the visitor on multiple components in parallel using `libverticalparallelization`

3 – General Usage

1. Identify a section of code with multiple tasks you want to execute in parallel. This will define a parallel section, *i.e.* a subpart of the program that is repeated at each iteration with the same data access pattern.
A Task corresponds to a function call, so you should encapsulate the tasks in functions.
WARNING: The return value of the task is not valid at the moment...
2. Identify the data vectors/arrays that are shared between the tasks, and wrap them using `DataWrapper` so that the operator[] is overloaded to track memory accesses and manage locks.
WARNING: when your data is a multidimensional array (e.g. a matrix), overloading the operator[] on the "innermost" dimension can lead to degraded performances. Overloading a "outer" dimension can lead to better performances. See example `rodinia/hotspot3D_para.cpp`, where we wrap the 2nd dimension of a 3-dimensional matrix.
3. Define a unique identifier for your parallel section. Call `startParallelization` and `beginParallelSection` to get the `ParallelSection` object and set the flags to enable parallelization.
WARNING: if there are initialization steps in your program before the memory access become constant across iterations, Call `startParallelization` only after the initialization iterations are done.
4. Bind the values of the functions/Tasks parameters, using `std::ref` for reference parameters, to get a function of type `Result(void)`. Call `runTask()` to add the Task to the `ThreadPool` and start its execution.
5. Call `endParallelSection` to resume sequential execution at the end of the parallel section.

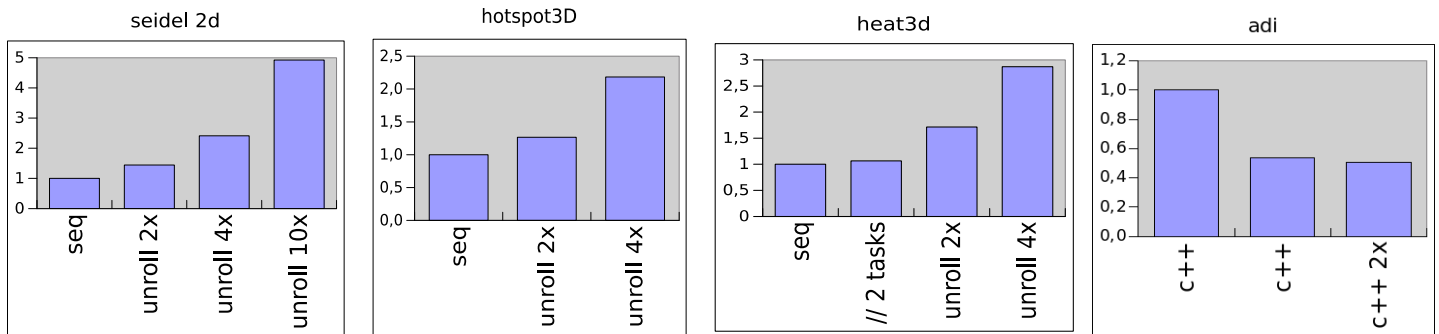
Remark : As stated in the paper, we have to group step in meta-steps (also called `ParallelPhase` in `libVerticalParallelization`) to decrease the overhead induced by the FIFO mechanism and the locks. The optimal size of the meta-steps/phases is not computed by `libVerticalParallelization`, and with current implementation the size is set arbitrarily in `memory_var.cpp`. It needs to be tuned and the lib recompiled, for each application.

4 – Performances on classical benchmarks

These results can also be found in the paper, with more precise description and comments.

We use classical benchmarking suites : rodinia (hotspot3D) and polybench (seidel2D, heat3d, adi). These are simple and computation intensive benchmarks.

In the original code, we have only 1 or 2 tasks that can be parallelized in an iteration, so I unroll the loops to have more tasks and parallelism opportunities.



adi is actually slower using the parallel version. There are multiple reasons for that:

- The tasks are not well balanced.
- The overhead of operator[] overloading is too high: too many accesses compared to computing work, and it prevents some compiler optimizations.

5 – Performances on a simple simulation in Sofa

We did not find any realistic scenario in Sofa where vertical parallelization could be successfully applied.

So the tests were done using a simple meaningless simulation of a 3D cubic object on which we apply 4 identical force fields: TetrahedronFEMForceField.

We define our parallel section as the execution of the addMBKDxVisitor on the Force Fields. This gives us 4 identical tasks: TetrahedronFEMForceField::AddDForce.

Each Task is decomposed in 32 meta-steps (ParallelPhases) of ~60000 vector accesses.

| | Sofa Master sequential | Vertical parallelization 1 Thread | Vertical parallelization 4 Threads |
|-----------|------------------------|--------------------------------------|---------------------------------------|
| Exec time | 63.44 s. | 71.47 s. | 30.80 s. |
| speedup | 1 | 0.89 | 2.06 |

Note that a sequential execution of our simulation using our modified version of Sofa with a custom vector type with operator[] overloading leads to degraded performances. Indeed, the additional operations performed prevent some compiler optimization, notably vectorization in some cases.

To compensate for the overhead, we need to extract sufficient parallelism from our parallel section.

For this simulation, we have a speedup of 2.06 when using 4 threads to run our 4 parallel tasks.

6 – Conclusion

We have mixed results. We have encouraging results on some cases, but the overhead of the overloading of `operator[]` and the prevented optimizations can be too high to get any speedup.

Before releasing this library, two things should be improved :

1. The size of the meta-steps should be computed automatically.
2. There should be a way to set the number of threads to use without having to compile the library again.