Imperial College
London

Hybrid Parallelization of SU2

A Comprehensive Introduction

Pedro Gomes, Rafael Palacios

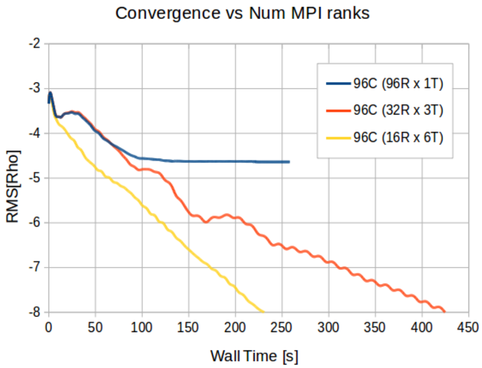
1st Annual SU2 Conference, 10-12 June 2020

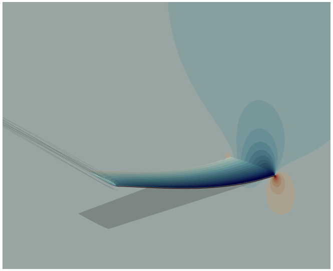
Contents

Motivation

Faster and more robust code, that scales better.

- Algorithms work better;
- Dynamic load balancing;
- Reduced communication overhead;





Objective, fast medium scale optimizations.

Motivation

Q: What is covered by the implementation?

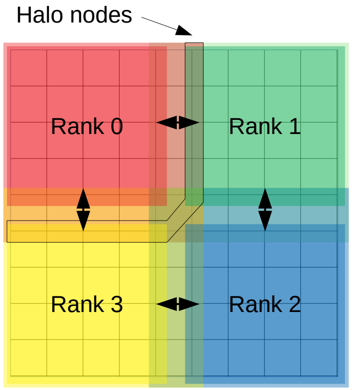
A: Primal and forward AD compressible URANS FSI (and subsets).

Q: How do I use it?

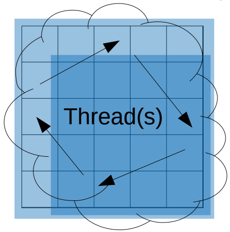
```
./meson.py ... -Dwith-omp=true ...

# auto number of threads/rank
SU2_CFD config.cfg
# 8 threads total
mpirun -n 2 --bind-to numa SU2_CFD -t 4 ...
# never --bind-to core
# mileage may vary, e.g. 2*4 != 4*2

# Useful environment variables:
# overrides default threads/rank
export OMP_NUM_THREADS=4
# better performance on some systems
export OMP_WAIT_POLICY=ACTIVE
```

The hybrid parallel model

Domain decomposition for MPI (static) vs the (possibly) dynamic movement of threads within sub-domains.



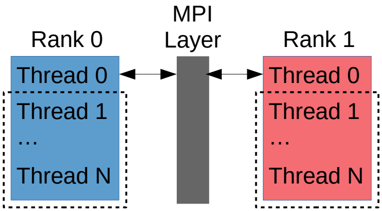
Data from other ranks is obtained **indirectly** via messages.



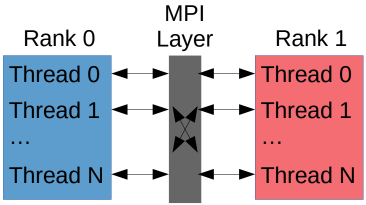
Threads within each partition can **directly** access any part of it.

The hybrid parallel model

The threads can interact with MPI in different ways, currently communications are **funneled** (multiple is WIP).



Funneled communication
(only the main thread uses MPI)



Multiple communication
(any thread at any time can use MPI)

OpenMP, an overview

An API that provides a simple and flexible interface (mostly in the form of pragmas) to develop portable parallel applications.

```c
const int N = 1024; // a shared variable
// start some threads
#pragma omp parallel
{
  int i; // a private variable
  // distribute loop indexes over threads
  #pragma omp for schedule(dynamic,32)
  for(i=0; i<N; ++i)
    myThreadSafeFunction(i);
}
```

We want to use this API to distribute the work (loops) in each MPI partition over its threads.

OpenMP, an overview

How do threads "communicate" between themselves?

```
// a function called by multiple threads
// x,y shared variables
void axpy(int N, double a, const double* x, double* y)
{
  // here there is no guarantee that the threads
  // have a consistent view of the arrays
  #pragma omp barrier
  // now there is
  #pragma omp for simd schedule(static,1024)
  for(int i=0; i<N; ++i) y[i] += a*x[i];
  // there are implicit barriers after most
  // worksharing directives
}
```

Identifying and counting threads:

```
omp_get_num_threads() <=> "size"
omp_get_thread_num() <=> "rank"
```

OpenMP, an overview

Shared vs private variables

```
#pragma omp parallel num_threads(4)
{
  // each thread has its own stack -> private
  double x[64] = {1.0};
  // the heap is shared, but we made 4 y's...
  vector<double> y(64,2.0);
  // this will not do what we want...
  axpy(64, 0.5, x, y.data());
}
```

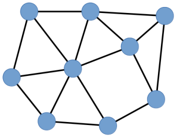
So do we need to declare/allocate everything outside parallel regions? Yes, and no.

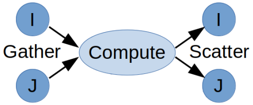
```
vector<double> y;
#pragma omp parallel num_threads(4)
{
  #pragma omp master
  y.resize(64,2.0); // only one thread allocates
  #pragma omp barrier
  ...
```

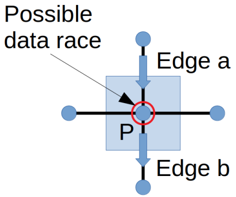
Challenges

Data races

When multiple threads simultaneously modify the same memory location (in an unregulated manner).







Builtin (OpenMP) solutions:

- Atomic operations;
- Critical directive;
- Locks;

Algorithmic solutions:

- Coloring / Partitioning;
- Scatter to Gather transformations;

Challenges

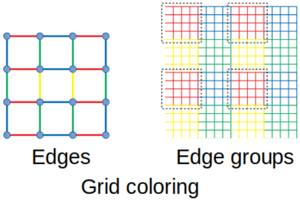
Builtin solutions:

```
// atomics are good for reduction operations
auto mySum = f(); // a private variable
#pragma omp atomic
ourSum += mySum; // safe update of shared variable

// critical for global resources
#pragma omp critical
cout << mySum << endl; // serialize output (unordered)

// locks for specific resources
const auto j = selectResource(omp_thread_num());
omp_set_lock(fileLocks[j]);
files[j] << mySum << endl;
omp_unset_lock(fileLocks[j]);
```

Pros: Small modifications to existing algorithms.
Cons: Overhead, poor scaling for resources used intensively.

Challenges

Partitioning, re-partition the MPI sub-domains, not what we want.
Coloring, create non-intersecting sets of entities (data race free).



```
for(auto c : EdgeColoring) {
  SU2_OMP_FOR_DYN(groupSize)
  for(int k=0; k<c.size; ++k) {
    auto idx = c.indices[k];
    ...
```

Pros: Load balancing via dynamic scheduling.
Cons: Reduced locality, parallel inefficiency (not enough work chunks for all threads).

This is our first choice for residual loops (use option EDGE_COLORING_GROUP_SIZE [512] to tune it).

Challenges

Scatter to Gather transformations:

```
for(edge : Edges) {
  // gather
  auto f = y[iPt]+y[jPt];
  // scatter
  x[iPt] += f;
  x[jPt] += f;
}
```

```
for(iPt : Points) {
  // gather
  for(jPt : neighbors(iPt))
    x[iPt]+=y[iPt]+y[jPt];
}
```

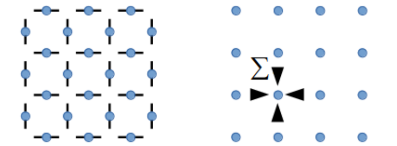
Pros: Embarrassingly parallel code, if the FLOP/BYTE ratio is O(1) the code may perform better.
Cons: Needs adjacency matrix, 2x slower if FLOP/BYTE $>> 1$.

This is what was done for preprocessing-type routines (gradients, limiters, sensors, etc.).

Challenges

Scatter to Gather transformations (two loop approach):

```
for(edge : Edges) {
  // gather
  f[edge] = y[iPt]+y[jPt];
}
for(iPt : Points) {
  // gather
  for(edge : Edges(iPt))
    x[iPt] += f[edge];
}
```



Step 1- Fluxes Step 2- Reduce

Reduction strategy

Pros: Approximately the same number of flops.
Cons: Extra storage needed for temporary variables, reduction loop has very low FLOP/BYTE ratio.

When edge coloring fails, SU2 falls back to this approach. About 20% slower worst case. A hybrid approach would probably be optimal.

Implementation overview

All pragmas and functions used throughout the code are wrapped in omp_structure.hpp, this allows disabling everything when -Dwith-omp=false (default).

```
#define SU2_OMP_SIMD SU2_OMP(simd)
#define SU2_OMP_MASTER SU2_OMP(master)
#define SU2_OMP_ATOMIC SU2_OMP(atomic)
#define SU2_OMP_BARRIER SU2_OMP(barrier)
#define SU2_OMP_CRITICAL SU2_OMP(critical)
#define SU2_OMP_PARALLEL SU2_OMP(parallel)
...
```

SU2_OMP_ ≈ SU2_MPI::

Implementation overview

Threads are started once per iteration (and per integration) in CIntegration (single or multi grid), output, to screen and file, is not multi-threaded.

```
/*--- Start an OpenMP parallel region covering the
    entire MG iteration, if the solver supports it.
    ---*/
SU2_OMP_PARALLEL_(if(solver_container[iZone][iInst][
    MESH_0][Solver_Position]->GetHasHybridParallel()))
{
...
```

All routines that are part of one iteration must be thread-safe (i.e. no unguarded writes to colliding memory locations).

Implementation overview

The "one numerics per thread" paradigm:
Numerics are shared objects (instantiated outside parallel regions) with mutable state and thus cannot be used by multiple threads.

```
/*--- Pick one numerics object per thread. ---*/
CNumerics* numerics = numerics_container[CONV_TERM +
    omp_get_thread_num()*MAX_TERMS];
```

This kind of temporary variable must also be avoided:

```
class CSolver {
  su2double *Solution, /*!< \brief Auxiliary ... */
  *Solution_i,         /*!< \brief Auxiliary ... */
  ...
```

What about using one per thread too? Bad idea due to false sharing.

Implementation overview

Grid coloring or fallback strategies are setup in solver constructors, then in residual loops:

```
  if (ReducerStrategy) {
    EdgeFluxes.SetBlock(iEdge, residual);
    Jacobian.SetBlocks(iEdge, ...
  }
  else {
    LinSysRes.AddBlock(iPoint, residual);
    LinSysRes.SubtractBlock(jPoint, residual);
    Jacobian.UpdateBlocks(iEdge, iPoint, jPoint, ...
  }
...
} // end color loop

if (ReducerStrategy) {
  SumEdgeFluxes();
  Jacobian.SetDiagonalAsColumnSum();
}
```

Implementation overview

Other tricky areas

To go around a barrier, we need two barriers:

```
if(condition) {
  SU2_OMP_BARRIER // wait for all threads to enter
  SU2_OMP_MASTER {condition = f();} // before updating
  SU2_OMP_BARRIER // or some might skip this barrier
}
```

Not so obvious deadlocks:

```
axpy(N,a,x,y); // this is fine, works in serial
SU2_OMP_PARALLEL {
  axpy(N,a,x,y); // this is fine, works in parallel
  SU2_OMP_MASTER {axpy(N,a,x,y);} // deadlock
  // other threads missed the barrier inside axpy
}
```

Implementation overview

AD-compatible funneled reductions (would be simpler with multiple communication):

```
su2double minElem(int N, const su2double* x) {
  static su2double ourMin; // global var!!
  SU2_OMP_BARRIER // consistent view of x
  SU2_OMP_MASTER {ourMin = 1e30;} // init global
  su2double myMin = 1e30; // init local
  SU2_OMP_FOR_STAT(256)
  for(int i=0; i<N; ++i) myMin = min(myMin, x[i]);
  SU2_OMP_CRITICAL // serialize update of shared var
  ourMin = min(ourMin, myMin);
  SU2_OMP_BARRIER // wait for all updates
  SU2_OMP_MASTER { // master communicates
    myMin = ourMin;
    SU2_MPI::Allreduce(&myMin, &ourMin,...
  }
  SU2_OMP_BARRIER // consistent view of ourMin
  return ourMin; // same on all threads and ranks
}
```

Concluding remarks

- Small set of OpenMP features used (also for eventual compatibility with reverse AD);
- That are still enough to improve scalability;
- Somethings require a bit more care, but essentially just be careful when writing to shared variables;
- Still lots of WIP, the solvers currently covered are a test bed for hybrid parallel strategies, it will take some time to cover everything.