Profiling Tutorial #1

Introduction and TINYPROFILER

Profiling… do it.

Here’s why:

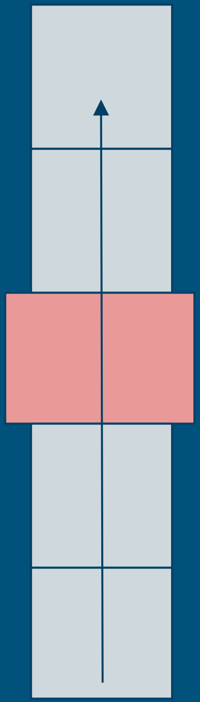
Is it doing what I want it to do?

Is it behaving at 4000 nodes?

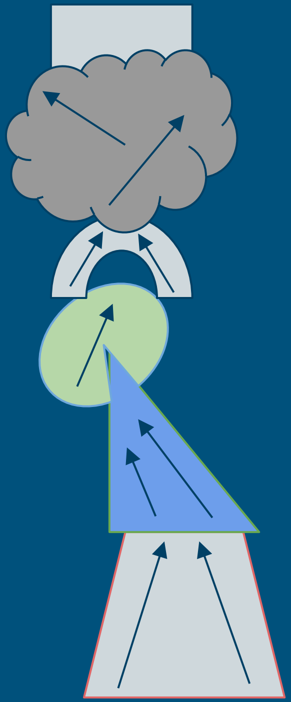
Where is my code slow?

Look at what I did!

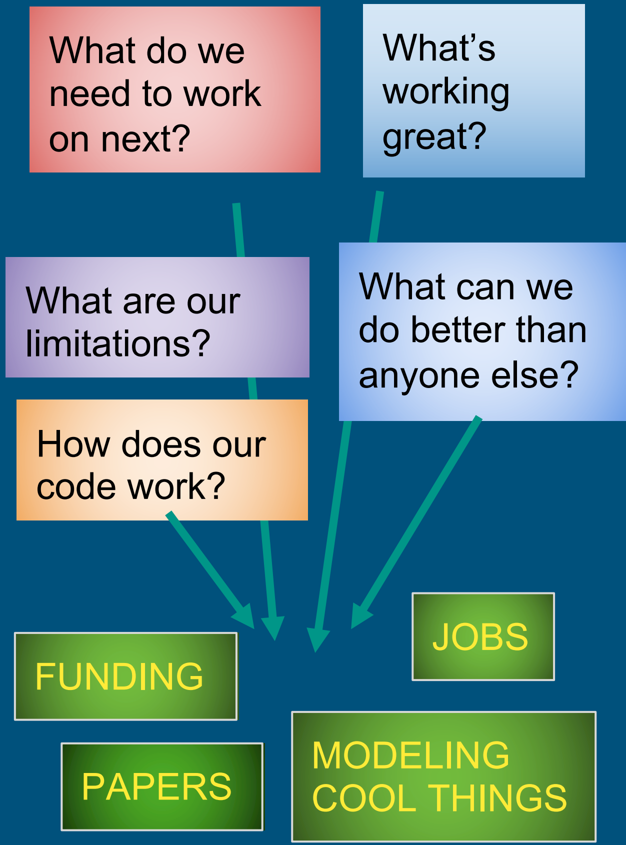
Get YOUR features into the profiling!



Profiled Code



“Just Work” Code



What is Profiling?

"Profiling": Information on what parts of the libraries your application uses, which MPI ranks use them, when they are used and how long they are used.

- Where does your application spend its time?
- Is your application load balanced?
- What changes when you run on 1000 ranks? When you turn on a different model? When you change the domain of your problem?
- Where are my communication barriers? Are those necessary?

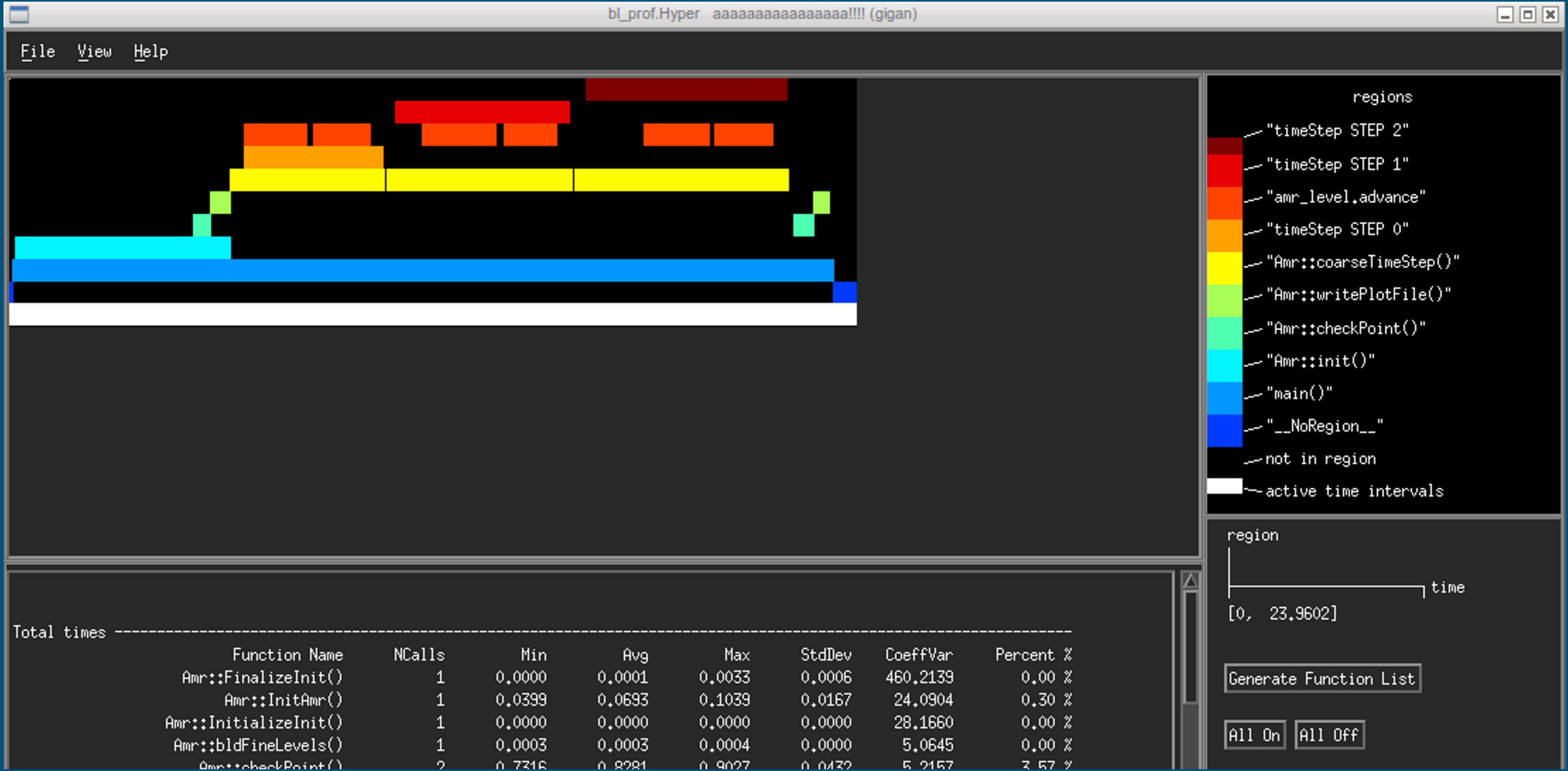
Typically try to work on the scale of "functions" or substantial, describable pieces of the code. (Order of hundred of calls, not millions, per profiled region.)

What is AMReX Profiling?

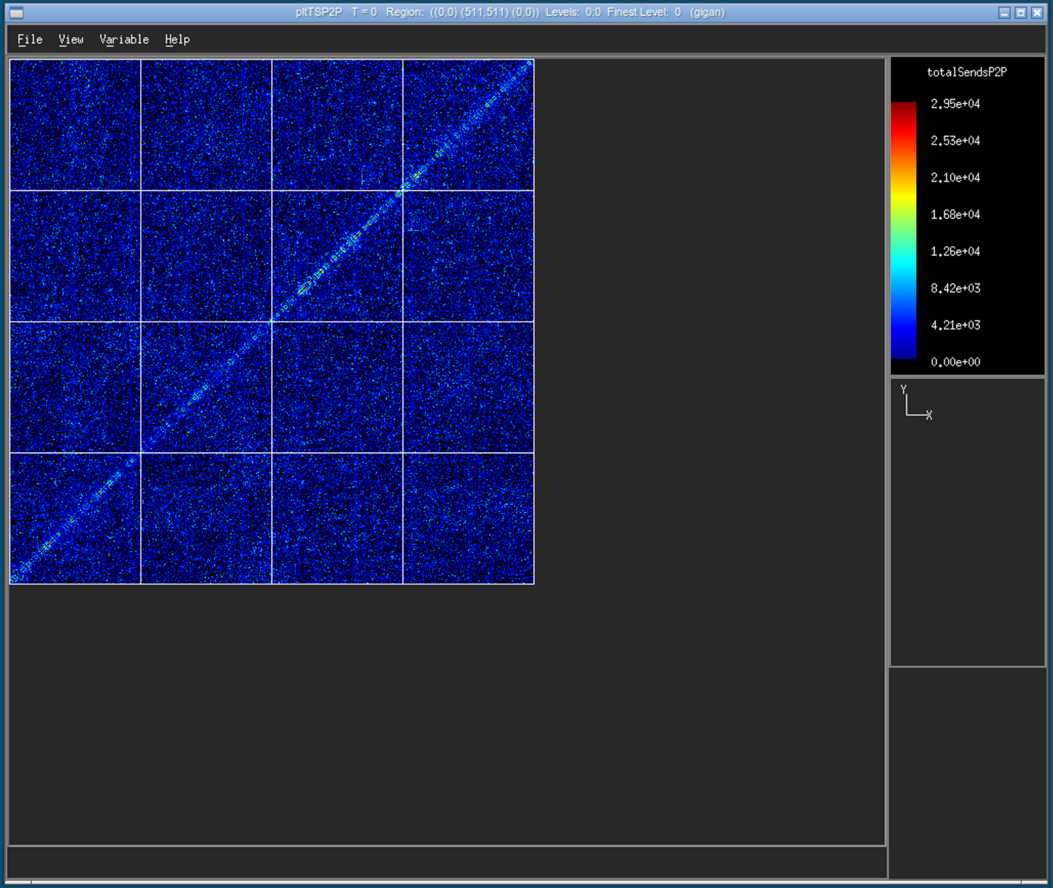
- Already built into AMReX in the form of C++ functions/macros.
- At runtime: Collects and writes a database of profiling information: “bl_prof”. Works at scale with minimal/no comms.
- Yields a minimal overhead to your overall code (1-2% at most.)
- Database is analyzed with AMRVis.
    - Allows visualization, breakdown and filtering of the data.
    - Creates plotfiles of profiling data for additional, more detailed analysis.
    - New features and visualization methods are made right here.

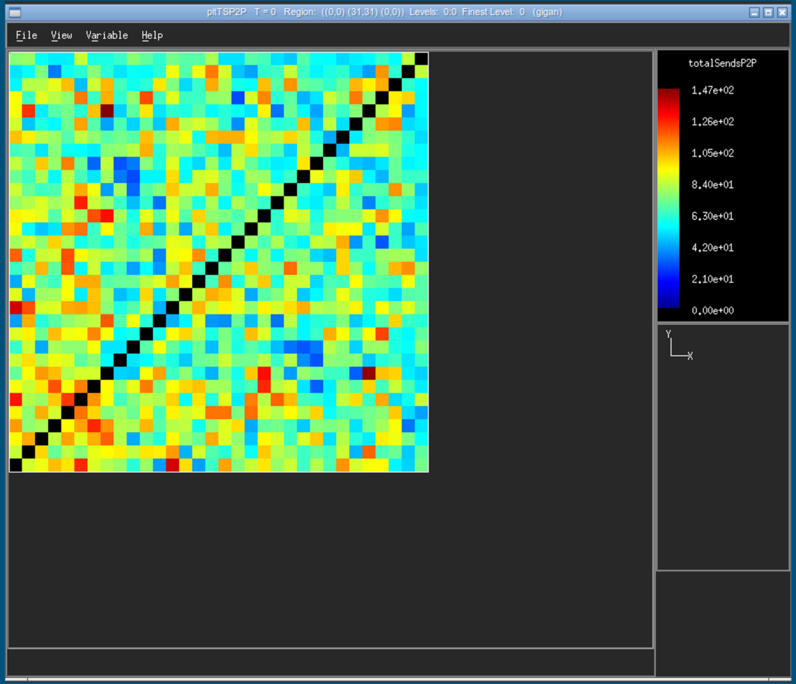
★ Features are implemented with just a couple Make flags and function calls.

ProfParser Current Features - ProfParser

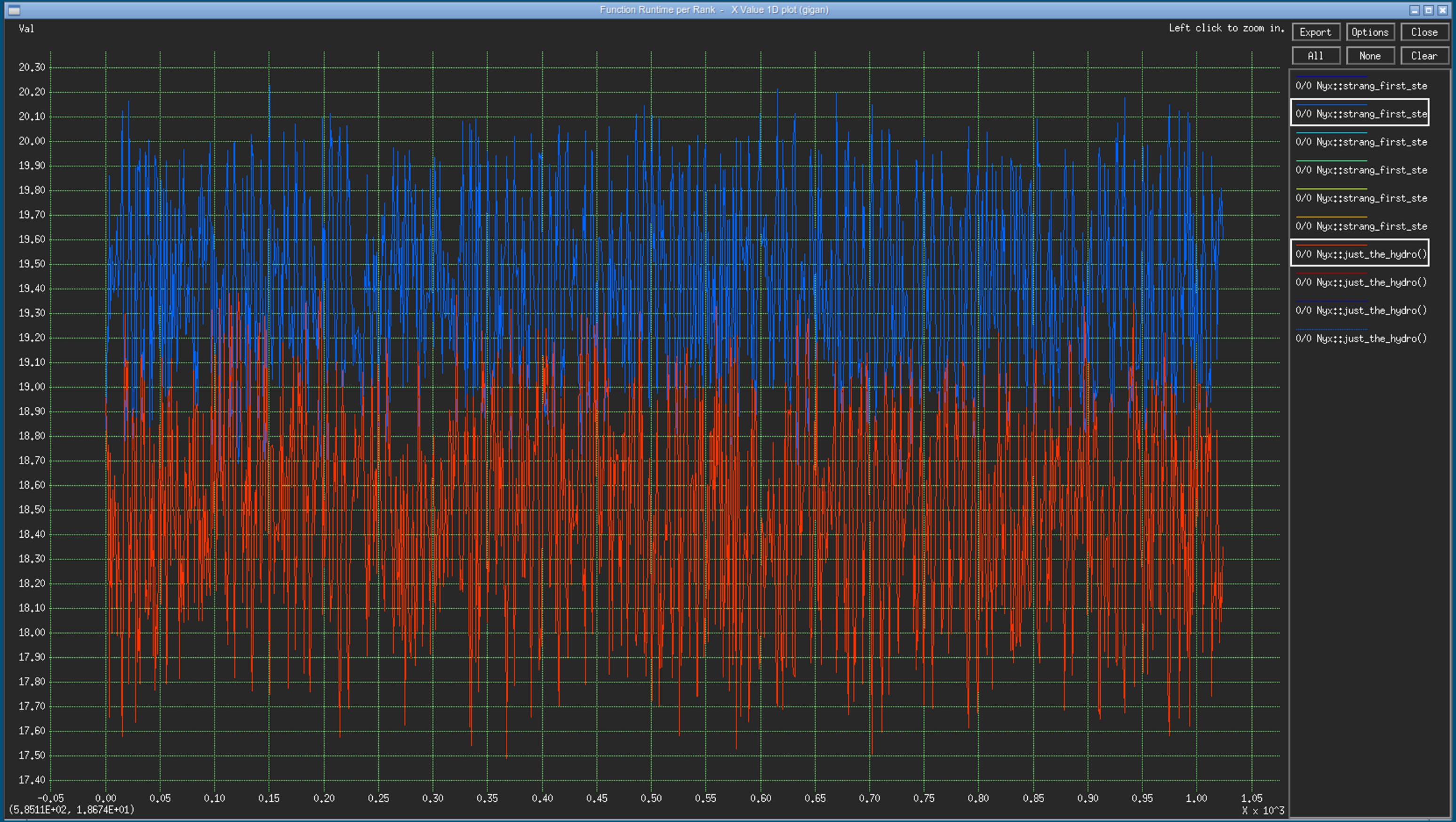


ProfParser Current Features - Sends Plotfile

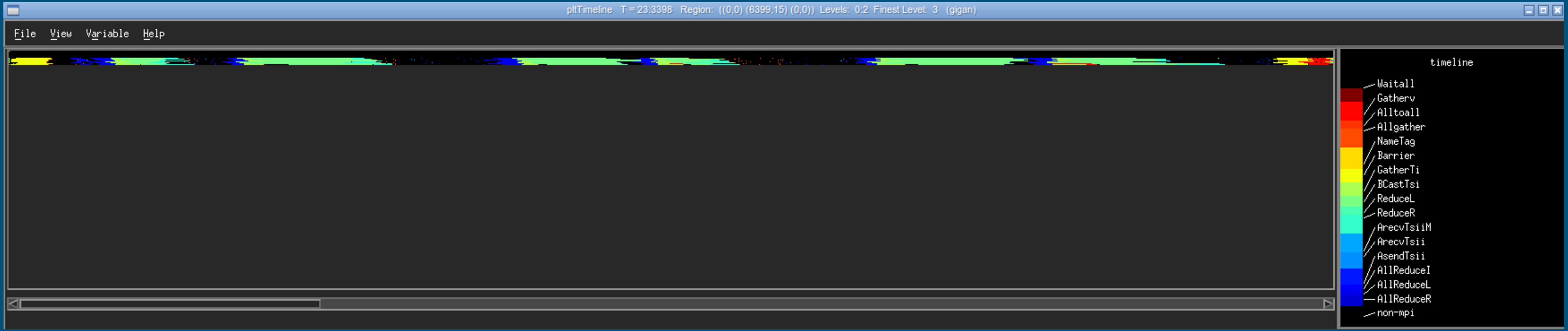


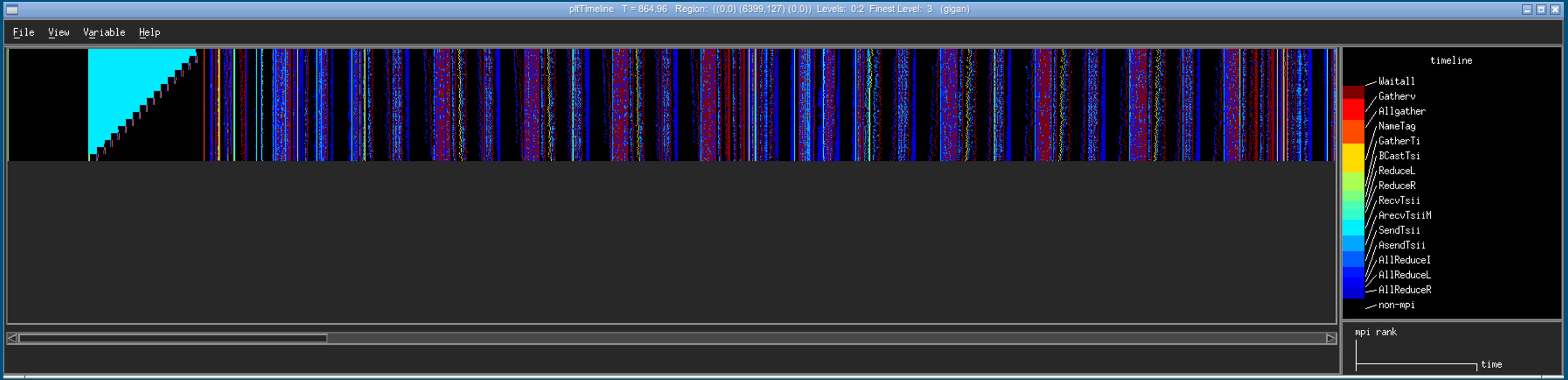


ProfParser Current Features - Function Plot



ProfParser Current Features - Timeline





ProfParser Future Plan

Currently Being Developed:

1. Get ProfParser GUI working at scale.
2. Move all features into the GUI.
3. User suggested features.
4. Complex filtering: Regions, Time Ranges, Processor Lists, Communicators.
5. GPU Profiling

TINYPROFILER

TINYPROFILER

Introduction, baseline AMReX profiler.

Adds timers to user-defined C++ sections of code.

Results put into stdout at the conclusion of the simulation:

- # of calls.
- Timer results across MPI ranks.
- Total % time in that function.

TinyProfiler total time across processes [min...avg...max]: 64.1 ... 64.22 ... 64.28

Name	NCalls	Excl. Min	Excl. Avg	Excl. Max	Max %
Amr::readProbinFile()	1	0.7714	12.24	23.37	36.36%
DistributionMapping::LeastUsedCPUs()	1	0.3214	11.03	22.16	34.48%
NavierStokesBase::velocity_advection()	12	5.814	6.796	7.414	11.53%
FabArray::ParallelCopy()	1889	2.595	3.93	6.733	10.47%
NavierStokes::scalar_advection()	12	3.721	4.035	4.944	7.69%
NavierStokes::predict_velocity()	12	3.956	4.221	4.429	6.89%
FabArray::FillBoundary()	447	1.393	2.296	3.405	5.30%
NavierStokes::getViscTerms()	49	2.02	2.507	3.251	5.06%
NavierStokes::velocity_diffusion_update()	11	1.384	2.041	2.967	4.62%
NavierStokes::scalar_update()	24	0.3784	1.9	2.504	3.90%
Amr::defBaseLevel()	1	0.851	1.123	2.371	3.69%
MLLinOp::makeSubCommunicator()	58	0.1502	1.052	1.775	2.76%
main()	1	0.4701	1.06	1.431	2.23%
MLABecLaplacian::averageDownCoeffs()	45	0.6982	0.7253	1.089	1.69%
DistributionMapping::SFCProcessorMapDoIt()	1	0.03394	0.08083	0.9515	1.48%
Projection::level_project()	10	0.1877	0.3442	0.867	1.35%
NavierStokesBase::estTimeStep()	12	0.2696	0.5882	0.8515	1.32%
BndryData::define()	189	0.4122	0.4499	0.8151	1.27%
MLABecLaplacian::define()	45	0.4065	0.4201	0.7665	1.19%
MLMG::prepareForSolve()	58	0.3863	0.6267	0.7595	1.18%
MLMG::solve()	58	0.1884	0.5171	0.7548	1.17%
NavierStokesBase::velocity_update()	12	0.3346	0.5886	0.7093	1.10%
NavierStokesBase::level_projector()	10	0.02769	0.1071	0.6738	1.05%
Amr::InitAmr()	1	0.1495	0.3566	0.5965	0.93%
ABecLaplacian::Fapply()	144	0.3632	0.4323	0.5459	0.85%
MLABecLaplacian::prepareForSolve()	45	0.01878	0.4609	0.5042	0.78%
MLLinOp::defineGrids()	58	0.2278	0.2351	0.3734	0.58%
MLNodeLaplacian::buildMasks()	13	0.2679	0.2723	0.3729	0.58%
NavierStokesBase::velocity_advection_update()	12	0.1563	0.1917	0.3401	0.53%
MacProj::mac_project()	12	0.1571	0.1871	0.3298	0.51%
Projection::getGradP()	3776	0.1923	0.2268	0.3075	0.48%
MLABecLaplacian::FFlux()	2880	0.1581	0.1711	0.2545	0.40%
StateData::define()	2	0.0923	0.1873	0.2533	0.39%
NavierStokesBase::scalar_advection_update()	24	0.06432	0.086	0.2437	0.38%
NavierStokesBase::create_umac_grown()	12	0.07334	0.1586	0.2429	0.38%

Implementing TINYPROFILER

Make flags:

PROFILE=FALSE (If TRUE, this will override the TINY_PROFILER)

TINY_PROFILE=TRUE

CMake flags:

AMReX_BASE_PROFILE=OFF (Again, will override TINY)

AMReX_TINY_PROFILE=ON

(This inserts the compiler flag: -DBL_TINY_PROFILING)

Implementing TINYPROFILER

int main(...) {

amrex::Initialize(argc, argv);

BL_PROFILE_VAR("main()", pmain);

……

BL_PROFILE_VAR_STOP(pmain);

amrex::Finalize();

Add these lines directly inside AMReX's initialize and finalize steps.

Now, the TINYPROFILER is on and working!!

Will output AMReX built-in instrumented variables.

TINYPROFILER Output

Without any additional instrumentation, the TINYPROFILER output will print to stdout at the end. Specifically, it's the first thing done in amrex::Finalize().

However, you may need to print before amrex::Finalize, e.g:

- you think there might be an error, or
- you want to use your entire batch submission allotment,

Print early by inserting the macro BL_PROFILE_TINY_FLUSH().

+ Will write **finished** timers up to the point of the call to stdout.

+ Be sure to place outside as many timers as possible and document well in stdout! (e.g. add additional output marking which result this is.)

BL_PROFILE_VAR: C++

- Implemented in both the tiny profiler and the full profiler.

C++:

Profiling variables are scoped and will automatically run a stop timer and properly close when it's destructor is implemented:

```
void YourClass:Your Function () {
BL_PROFILE_VAR("ref_name",  object_name);
   … your function ...
}
```

For profiling within a scope, use `stop` to end the timer:

```
BL_PROFILE_VAR("ref_name", object_name);
                    ….your code….
BL_PROFILE_VAR_STOP(object_name);
```

To restart an already defined timer elsewhere (to capture two separate code blocks in the same timer), use `start`:

```
BL_PROFILE_VAR("ref", refTimer);
                    ….your code block A….
BL_PROFILE_VAR_STOP(refTimer);

                    ….untimed code….

BL_PROFILE_VAR_START(refTimer);
                    ….your code block B….
BL_PROFILE_VAR_STOP(refTimer);
```

BL_PROFILE_VAR: Fortran

- Fortran timers are currently only implemented in the full profiler.
- Unavailable in the Tiny Profiler.

Profiling variables cannot be scoped and so explicit starts and stops are needed:

```
call bl_proffortfuncstart("func_name")
call bl_proffortfuncstop("func_name")
```

For a little extra speed, you can assign a numerical value to avoid the string lookup:

```
call bl_proffortfuncstart_int(int n)
call bl_proffortfuncstop_int(int n)
```

You can assign a name to a given number in top of your main():

```
BL_PROFILE_CHANGE_FORT_INT_NAME("fname", n)
```

As fortran variables cannot be scoped, there is no special implementation to capture two code blocks in one timer:

```
call bl_proffortfuncstart("func_name")
      CODE BLOCK 1
call bl_proffortfuncstop("func_name")

      UNTIMED CODE

call bl_proffortfuncstart("func_name")
      CODE BLOCK 2
call bl_proffortfuncstop("func_name")
```

BL_PROFILE_VAR: Details

Src/Base/AMReX_BLProfiler.{H, cpp}
Src/Base/AMReX_TinyProfiler.{H,cpp}

Creates a profiling variable that stores the timer for this instance of the call and the stack to calculate exclusive times.

If profiling is not turned on, it does NOTHING.

```
#ifdef BL_PROFILING

#define BL_PROFILE_VAR(fname, vname)   amrex::BLProfiler  bl_profiler__##vname((fname));

#elif defined(BL_TINY_PROFILING)

#define BL_PROFILE_VAR(fname, vname)   amrex::TinyProfiler
tiny_profiler__##vname((fname));

#else

#define BL_PROFILE_VAR(fname, vname)
```

```
void
TinyProfiler::start ()
{
#ifdef _OPENMP
#pragma omp master
#endif
if (stats.empty())
{
    Real t = amrex::second();
    ttstack.push(std::make_pair(t, 0.0));

    global_depth = ttstack.size();
    for (auto const& region : regionstack)
    {
        Stats& st = statsmap[region][fname];
        ++st.depth;
        stats.push_back(&st);
    }
}
}
```

Output

NCalls: # of times BL_PROFILE_VAR was called on I/O Processor.

Exclusive Times: Time spent ONLY in that part of the code.

Inclusive Timers: All time spent within that variable, including nested variables.

Max %: Maximum % of time spent in that variable, across all MPI ranks.

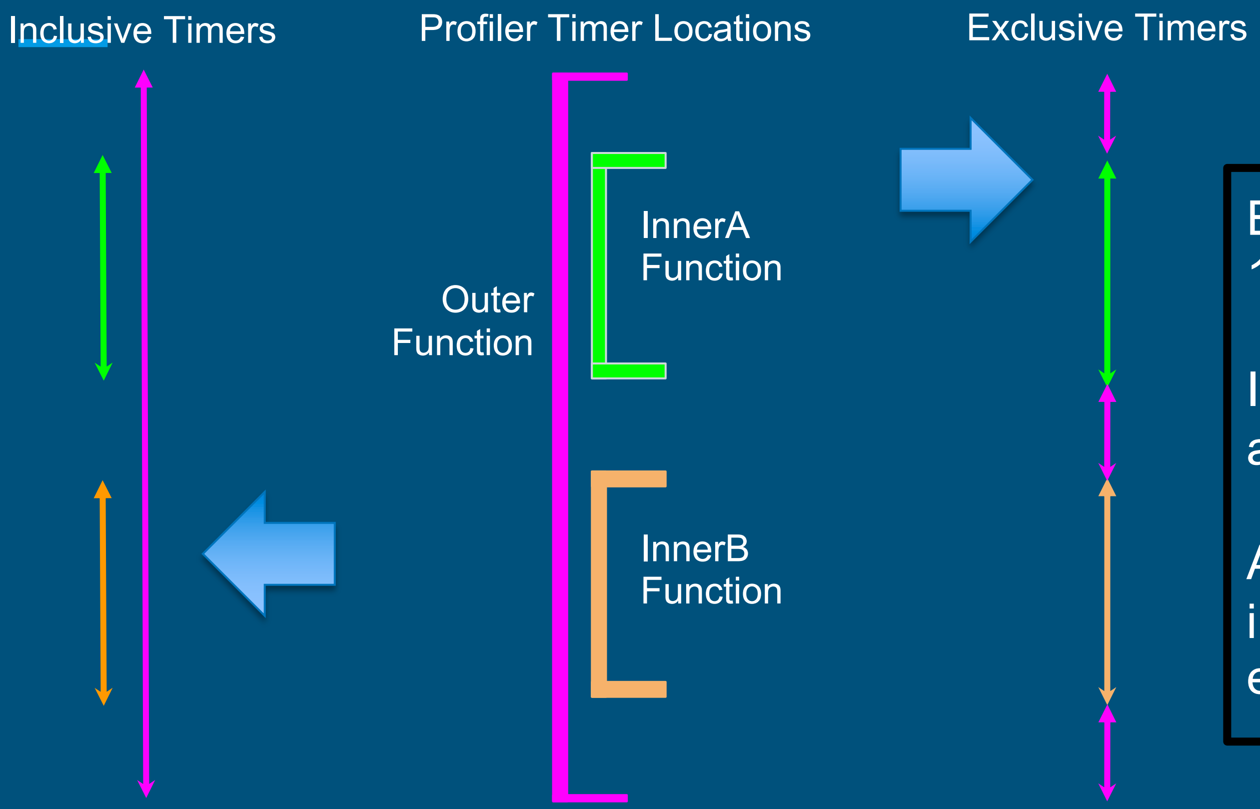
TinyProfiler total time across processes [min...avg...max]: 2.263 ... 2.263 ... 2.263

Name	NCalls	Excl. Min	Excl. Avg	Excl. Max	Max %
LevelAdvance::LevelAdvance_RRM()	14	0.004599	0.2059	0.3206	14.17%
FabArray::ParallelCopy()	109	0.1461	0.2234	0.3093	13.67%
FabArrayCopyDescriptor::CollectData()	9	0.1424	0.2202	0.2635	11.65%
Sweep::FORT_RIEMANN()	207	0.177	0.196	0.2214	9.79%
HyperCLaw::initData()	7	0.03208	0.1125	0.1692	7.48%
Sweep::FORT_PRIMITIVES()	207	0.1464	0.1574	0.1691	7.48%
CellConservativeLinear::interp()	520	0.1038	0.1092	0.1197	5.29%
LevelAdvance::FORT_DIVUNODE()	69	0.07881	0.08272	0.08674	3.83%
Trace::FORT_XTRACE()	69	0.05271	0.06297	0.07789	3.44%
Amr::InitializeInit()	1	6.682e-06	8.282e-06	1.294e-05	0.00%
AmrLevel::checkPointPost()	6	8.031e-06	9.385e-06	1.112e-05	0.00%
HyperCLaw::computeNewDt()	1	7.567e-06	8.762e-06	9.922e-06	0.00%
HyperCLaw::computeInitialDt()	2	6.996e-06	7.915e-06	9.569e-06	0.00%
Amr::init()	1	6.92e-06	7.946e-06	9.104e-06	0.00%
Amr::initialInit()	1	2.44e-06	4.516e-06	8.569e-06	0.00%
HyperCLaw::post_init()	3	5.801e-06	7.29e-06	8.531e-06	0.00%

Name	NCalls	Incl. Min	Incl. Avg	Incl. Max	Max %
main()	1	2.198	2.198	2.198	97.14%
Amr::coarseTimeStep()	2	1.805	1.806	1.806	79.80%
Amr::timeStep()	14	1.796	1.796	1.797	79.41%
HyperCLaw::advance()	14	1.743	1.747	1.75	77.34%
LevelAdvance::LevelAdvance()	14	1.743	1.746	1.749	77.32%
LevelAdvance::Sweep()	207	0.7668	0.8506	0.955	42.21%
FillPatchIterator::Initialize	22	0.4517	0.5644	0.6481	28.64%
Amr::init()	1	0.3483	0.3484	0.3484	15.40%
LevelAdvance::LevelAdvance_RRM()	14	0.004599	0.2059	0.3206	14.17%
FabArray::ParallelCopy()	109	0.1492	0.2261	0.3119	13.79%
Amr::initialInit()	1	0.2984	0.2985	0.2989	13.21%
Amr::FinalizeInit()	1	0.296	0.2967	0.2974	13.15%

Inclusive vs. Exclusive Timers

Note: In AMReX output, % is based on maximum % across ranks, so exclusive timers can sum to >100% due to noise, variation and/or load imbalance.



Exclusive timers sum to 100%.

Inclusive is fixed if you add additional timers.

At the inner most level, inclusive timers = exclusive timers.

Regions

Might also see "Regions"

Regions delineate sections of code to be analyzed separately.

If you see one, the output for a function matches that of the total code. Data inside the region has just been isolated for convenience.

```
BEGIN REGION MLMG::solve()

------------------------------------------------------------------------------------------
Name                                       NCalls  Excl. Min  Excl. Avg  Excl. Max   Max %
------------------------------------------------------------------------------------------
MLABecLaplacian::averageDownCoeffs()           45      0.691     0.7187     0.8786   2.29%
MLMG::prepareForSolve()                        58     0.3871     0.6113      0.747   1.95%
FabArray::FillBoundary()                      188     0.2563     0.4188     0.7025   1.83%
MLMG::solve()                                  58     0.2066     0.4915     0.6514   1.70%
FabArray::ParallelCopy()                     1086    0.01025     0.0529     0.4662   1.22%
MLABecLaplacian::prepareForSolve()             45    0.01089     0.2209     0.2533   0.66%
MLABecLaplacian::Fapply()                      45     0.1039     0.1338     0.1682   0.44%
MLMG::ResNormInf()                             58    0.02627    0.03675    0.08615   0.22%
```

```
MLABecLaplacian::Fapply()                      45     0.1039     0.1338     0.1682   0.44%
MLNodeLaplacian::applyBC()                     13     0.0339    0.07904      0.162   0.42%
MLMG::MLResNormInf()                           58    0.02709    0.03763    0.08708   0.23%
MLMG::ResNormInf()                             58    0.02627    0.03675    0.08615   0.22%
MLNodeLaplacian::Fapply()                      13    0.04855    0.05462    0.06648   0.17%
MLCellLinOp::prepareForSolve()                 45    0.03029    0.03712     0.0548   0.14%
FabArrayBase::getFB()                         376    0.02319    0.02573    0.04305   0.11%
MLMG::MLRhsNormInf()                           58    0.02407    0.03194    0.04204   0.11%
FabArrayBase::FB::FB()                        142    0.02024    0.02277    0.03887   0.10%
FabArrayBase::getCPC()                       1086    0.01877    0.02623    0.03622   0.09%
FabArrayBase::CPC::define()                  1086    0.01385    0.02144    0.03078   0.08%
MLMG::buildFineMask()                          58  0.0005022  0.0006132  0.0007942   0.00%
------------------------------------------------------------------------------------------

END REGION MLMG::solve()
```

For Profiling Help, Contact:

Kevin Gott:

kngott@lbl.gov