Advances of the profiling & optimization of the LSST Pipeline

Antoine BERNARD (LPSC, Grenoble, France),
Q. Le Boulc'h, J. Bregeon, D. Boutigny, F. Hernandez, D. Parello, G. Mainetti

Science Pipelines Team Meeting, Feb. 12th 2025

(Full previous presentation available here about profling and optimization works with notebooks & more results.)

(Full previous presentation available here about profling and optimization works with notebooks & more results.)

- Context:
  - **Key Motivations**: Reducing resource use (CPU time, memory, storage) for finance, ecology, performance.
  - **Code Context**: Open-source code developed primarily by the Rubin project (US), many different computing nodes (USA / UK / France (CC-IN2P3))
  - **Profiler**: required for reliable **metrics**. Some requirements (Low overhead, native code compatibility, interactive visualizations, ...)

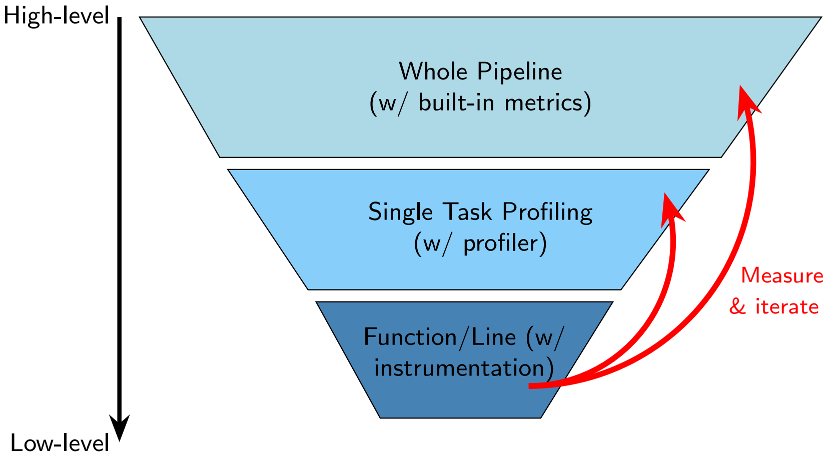
(Full previous presentation available here about profling and optimization works with notebooks & more results.)

- Context:
  - **Key Motivations**: Reducing resource use (CPU time, memory, storage) for finance, ecology, performance.
  - **Code Context**: Open-source code developed primarily by the Rubin project (US), many different computing nodes (USA / UK / France (CC-IN2P3))
  - **Profiler**: required for reliable **metrics**. Some requirements (Low overhead, native code compatibility, interactive visualizations, ...)
- Difficulties:
  - **Operational Context**: Many implementations decisions already made, cannot be disrupted / Different sites may have differents architectures (e.g. AMD vs Intel CPU)
  - **Mixed Language**: Difficulty profiling Python/C++ together (Python's memory management complicates analysis).
  - **Overhead**: Profiling tools significantly slow runtime.

(Full previous presentation available here about profling and optimization works with notebooks & more results.)

- Context:
  - **Key Motivations**: Reducing resource use (CPU time, memory, storage) for finance, ecology, performance.
  - **Code Context**: Open-source code developed primarily by the Rubin project (US), many different computing nodes (USA / UK / France (CC-IN2P3))
  - **Profiler**: required for reliable **metrics**. Some requirements (Low overhead, native code compatibility, interactive visualizations, ...)
- Difficulties:
  - **Operational Context**: Many implementations decisions already made, cannot be disrupted / Different sites may have differents architectures (e.g. AMD vs Intel CPU)
  - **Mixed Language**: Difficulty profiling Python/C++ together (Python's memory management complicates analysis).
  - **Overhead**: Profiling tools significantly slow runtime.
- Next steps:
  - **Set-up test case**: Profile with realistic datasets.
  - **Start optimization**: Piff package or **forcedPhotCoadd** seems promising targets.

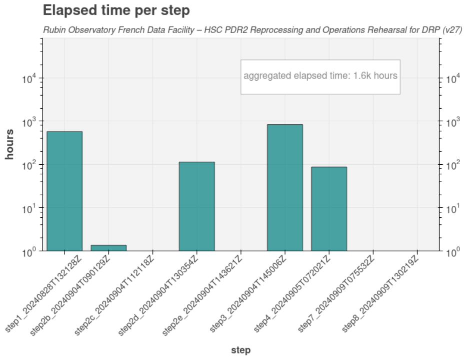
- Broad-to-specific profiling to isolate bottlenecks.
- Improvement shall be measured
- Process is iterative and empirical



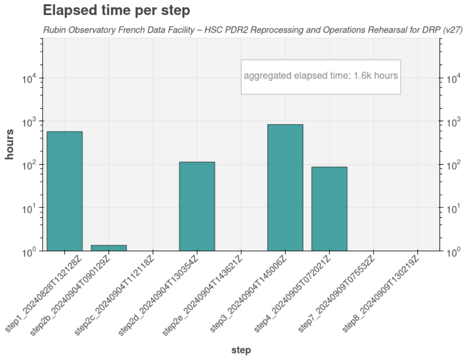
High-level profiling - Whole pipeline

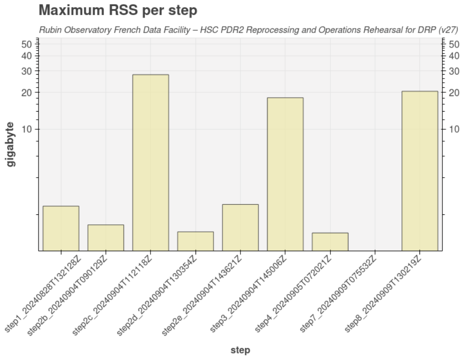
- LSST pipelines has inbuilt metrics about every task, providing memory used and time elapsed (as well as input identifiers)
- already used to analyze ressource usage during DP0.2
- re-used on a partial HSC PDR2 run, for test purpose

- LSST pipelines has inbuilt metrics about every task, providing memory used and time elapsed (as well as input identifiers)
- already used to analyze ressource usage during DP0.2
- re-used on a partial HSC PDR2 run, for test purpose

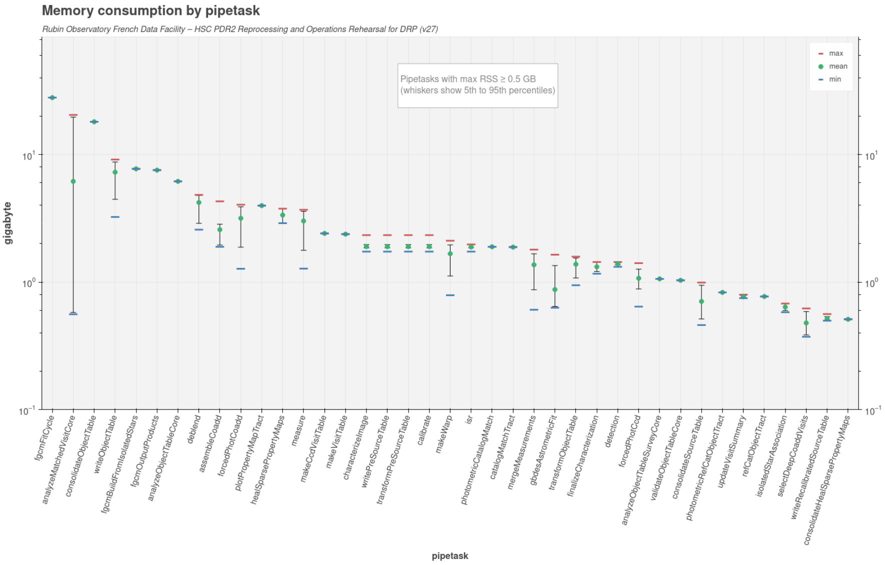


- LSST pipelines has inbuilt metrics about every task, providing memory used and time elapsed (as well as input identifiers)
- already used to analyze ressource usage during DP0.2
- re-used on a partial HSC PDR2 run, for test purpose

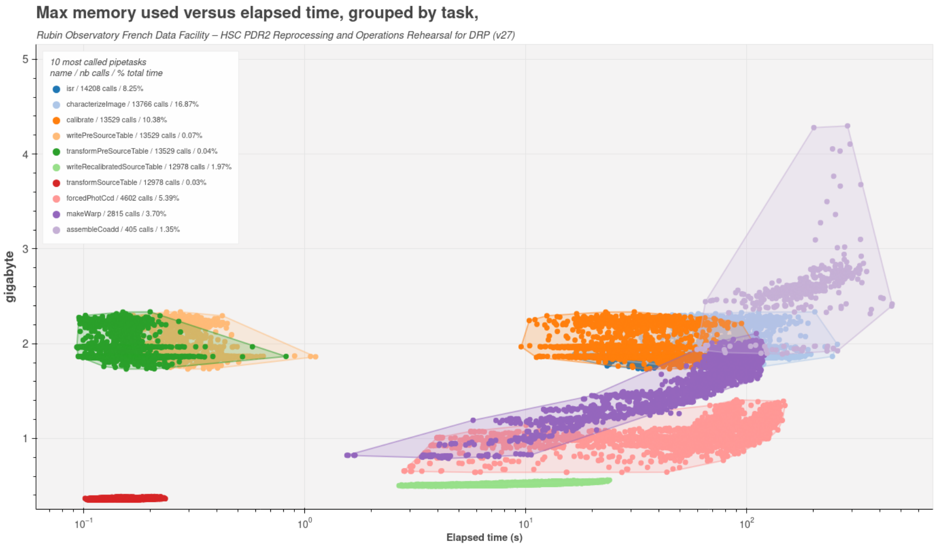




High-level profiling - Whole pipeline



High-level profiling - Whole pipeline



Profiling and optimizing forcedPhotCoadd

- Using IntelVTune profiler,

Profiling and optimizing forcedPhotCoadd

- Using IntelVTune profiler,
- Using stack version w_2024_47 with cvmfs,

Profiling and optimizing forcedPhotCoadd

- Using IntelVTune profiler,
- Using stack version w_2024_47 with cvmfs,
- Executing on ccahm002 interactive server,

Profiling and optimizing forcedPhotCoadd

- Using IntelVTune profiler,
- Using stack version w_2024_47 with cvmfs,
- Executing on ccahm002 interactive server,
- Input: HSC-PDR2, exposure=6414, patch=0, band='g' (smallest and fastest input) approx. 500sec

- Using IntelVTune profiler,
- Using stack version w_2024_47 with cvmfs,
- Executing on ccahm002 interactive server,
- Input: HSC-PDR2, exposure=6414, patch=0, band='g' (smallest and fastest input) approx. 500sec

Elapsed Time: 719.315s

CPU Time:	**698.510s**
Effective Time:	696.870s
Spin Time:	1.640s
Overhead Time:	0s
Total Thread Count:	62
Paused Time:	0s

- Using IntelVTune profiler,
- Using stack version w_2024_47 with cvmfs,
- Executing on ccahm002 interactive server,
- Input: HSC-PDR2, exposure=6414, patch=0, band='g' (smallest and fastest input) approx. 500sec

Elapsed Time: 719.315s

CPU Time:	698.510s
Effective Time:	696.870s
Spin Time:	1.640s
Overhead Time:	0s
Total Thread Count:	62
Paused Time:	0s

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time	% of CPU Time
func@0x63ea0	libm.so.6	41.466s	5.9%
Transform	libast.so.9	38.413s	5.5%
boost::math::tools::detail::evaluate_rational_c_imp<long double, long double, long double>	libmeas_base.so	29.715s	4.3%
DOUBLE_onemultadd	_sigtools.cpython-311-x86_64-linux-gnu.so	21.842s	3.1%
func@0x6b950	libm.so.6	20.688s	3.0%
[Others]	N/A*	546.386s	78.2%

*N/A is applied to non-summable metrics.

Profiling and optimizing forcedPhotCoadd

- Using IntelVTune profiler,
- Using stack version w_2024_47 with cvmfs,
- Executing on ccahm002 interactive server,
- Input: HSC-PDR2, exposure=6414, patch=0, band='g' (smallest and fastest input) approx. 500sec

Elapsed Time: 719.315s

CPU Time:	698.510s
Effective Time:	696.870s
Spin Time:	1.640s
Overhead Time:	0s
Total Thread Count:	62
Paused Time:	0s

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time	% of CPU Time
func@0x63ea0	libm.so.6	41.466s	5.9%
Transform	libast.so.9	38.413s	5.5%
boost::math::tools::detail::evaluate_rational_c_imp<long double, long double, long double>	libmeas_base.so	29.715s	4.3%
DOUBLE_onemultadd	_sigtools.cpython-311-x86_64-linux-gnu.so	21.842s	3.1%
func@0x6b950	libm.so.6	20.688s	3.0%
[Others]	N/A*	546.386s	78.2%

*N/A is applied to non-summable metrics.

Vectorization: 9.5%

Code modification

- Optimize:
  - tracked one consuming call stacks: **sincosf64x** and **sqrt** function used in **boost::math::detail::bessel_j1**,
  - in **meas_base** package, in **src/SincCoeffs.cc**, modified **calcImageKSpaceCplx** (see on Github),

```cpp
// compute the k-space values and put them in the cimg array
double const twoPiRad1 = geom::TWOPI * rad1;
double const twoPiRad2 = geom::TWOPI * rad2;
double const scale = (1.0 - ellipticity);
for (int iY = 0; iY < wid; ++iY) {
    int const fY = fftshift.shift(iY);
    double const ky = (static_cast<double>(iY) - ycen) / wid;

    for (int iX = 0; iX < wid; ++iX) {
        int const fX = fftshift.shift(iX);
        double const kx = static_cast<double>(iX - xcen) / wid;

        // rotate
        std::pair<double, double> coo = rotate(kx, ky, posAng);
        double kxr = coo.first;
        double kyr = coo.second;
        // rescale
        double const k = ::sqrt(kxr * kxr + scale * scale * kyr * kyr);
```

```cpp
// compute the k-space values and put them in the cimg array
double const twoPiRad1 = geom::TWOPI * rad1;
double const twoPiRad2 = geom::TWOPI * rad2;
double const scale = (1.0 - ellipticity);
// pre-compute rotation matrix
double const cosAng = ::cos(posAng);
double const sinAng = ::sin(posAng);
// pre-compute fftshift and kpos
double const inv_wid = 1.0 / wid;
std::vector<int> fftShifted(wid);
std::vector<double> kpos(wid);
for (int i = 0; i < wid; ++i) {
    fftShifted[i] = fftshift.shift(i);
    kpos[i] = (static_cast<double>(i) - cen) * inv_wid;
}
// pre-compute k
std::vector<double> kValuesFlat(wid * wid);
for (int iY = 0; iY < wid; ++iY) {
    int const fY = fftShifted[iY];
    double const ky = kpos[iY];

    for (int iX = 0; iX < wid; ++iX) {
        int const fX = fftShifted[iX];
        double const kx = kpos[iX];

        // rotate
        double const kxr =   cosAng * kx + sinAng * ky;
        double const kyr = - sinAng * kx + cosAng * ky;

        // rescale
        kValuesFlat[fY * wid + fX] = std::sqrt(kxr * kxr + scale * scale * kyr * kyr);
    }
}
// compute c
for (int el = 0; el < wid*wid; ++el) {
    double const k = kValuesFlat[el];
```

- Optimize:
  - tracked one consuming call stacks: **sincosf64x** and **sqrt** function used in **boost::math::detail::bessel_j1**,
  - in **meas_base** package, in **src/SincCoeffs.cc**, modified **calcImageKSpaceCplx** (see on Github),

```cpp
// compute the k-space values and put them in the cimg array
double const twoPiRad1 = geom::TWOPI * rad1;
double const twoPiRad2 = geom::TWOPI * rad2;
double const scale = (1.0 - ellipticity);
for (int iY = 0; iY < wid; ++iY) {
    int const fY = fftshift.shift(iY);
    double const ky = (static_cast<double>(iY) - ycen) / wid;

    for (int iX = 0; iX < wid; ++iX) {
        int const fX = fftshift.shift(iX);
        double const kx = static_cast<double>(iX - xcen) / wid;

        // rotate
        std::pair<double, double> coo = rotate(kx, ky, posAng);
        double kxr = coo.first;
        double kyr = coo.second;
        // rescale
        double const k = ::sqrt(kxr * kxr + scale * scale * kyr * kyr);
```

```cpp
// compute the k-space values and put them in the cimg array
double const twoPiRad1 = geom::TWOPI * rad1;
double const twoPiRad2 = geom::TWOPI * rad2;
double const scale = (1.0 - ellipticity);
// pre-compute rotation matrix
double const cosAng = ::cos(posAng);
double const sinAng = ::sin(posAng);
// pre-compute fftshift and kpos
double const inv_wid = 1.0 / wid;
std::vector<int> fftShifted(wid);
std::vector<double> kpos(wid);
for (int i = 0; i < wid; ++i) {
    fftShifted[i] = fftshift.shift(i);
    kpos[i] = (static_cast<double>(i) - cen) * inv_wid;
}
// pre-compute k
std::vector<double> kValuesFlat(wid * wid);
for (int iY = 0; iY < wid; ++iY) {
    int const fY = fftShifted[iY];
    double const ky = kpos[iY];

    for (int iX = 0; iX < wid; ++iX) {
        int const fX = fftShifted[iX];
        double const kx = kpos[iX];

        // rotate
        double const kxr =   cosAng * kx + sinAng * ky;
        double const kyr = - sinAng * kx + cosAng * ky;

        // rescale
        kValuesFlat[fY * wid + fX] = std::sqrt(kxr * kxr + scale * scale * kyr * kyr);
    }
}
// compute c
for (int el = 0; el < wid*wid; ++el) {
    double const k = kValuesFlat[el];
```

Code modification

- Optimize:
  - tracked one consuming call stacks: **sincosf64x** and **sqrt** function used in **boost::math::detail::bessel_j1**,
  - in **meas_base** package, in **src/SincCoeffs.cc**, modified **calcImageKSpaceCplx** (see on Github),

```cpp
// compute the k-space values and put them in the cimg array
double const twoPiRad1 = geom::TWOPI * rad1;
double const twoPiRad2 = geom::TWOPI * rad2;
double const scale = (1.0 - ellipticity);
for (int iY = 0; iY < wid; ++iY) {
    int const fY = fftshift.shift(iY);
    double const ky = (static_cast<double>(iY) - ycen) / wid;

    for (int iX = 0; iX < wid; ++iX) {
        int const fX = fftshift.shift(iX);
        double const kx = static_cast<double>(iX - xcen) / wid;

        // rotate
        std::pair<double, double> coo = rotate(kx, ky, posAng);
        double kxr = coo.first;
        double kyr = coo.second;
        // rescale
        double const k = ::sqrt(kxr * kxr + scale * scale * kyr * kyr);
```

```cpp
// compute the k-space values and put them in the cimg array
double const twoPiRad1 = geom::TWOPI * rad1;
double const twoPiRad2 = geom::TWOPI * rad2;
double const scale = (1.0 - ellipticity);
// pre-compute rotation matrix
double const cosAng = ::cos(posAng);
double const sinAng = ::sin(posAng);
// pre-compute fftshift and kpos
double const inv_wid = 1.0 / wid;
std::vector<int> fftShifted(wid);
std::vector<double> kpos(wid);
for (int i = 0; i < wid; ++i) {
    fftShifted[i] = fftshift.shift(i);
    kpos[i] = (static_cast<double>(i) - cen) * inv_wid;
}
// pre-compute k
std::vector<double> kValuesFlat(wid * wid);
for (int iY = 0; iY < wid; ++iY) {
    int const fY = fftShifted[iY];
    double const ky = kpos[iY];

    for (int iX = 0; iX < wid; ++iX) {
        int const fX = fftShifted[iX];
        double const kx = kpos[iX];

        // rotate
        double const kxr =   cosAng * kx + sinAng * ky;
        double const kyr = - sinAng * kx + cosAng * ky;

        // rescale
        kValuesFlat[fY * wid + fX] = std::sqrt(kxr * kxr + scale * scale * kyr * kyr);
    }
}
// compute c
for (int el = 0; el < wid*wid; ++el) {
    double const k = kValuesFlat[el];
```

- Idea in this:
  - Refactor code for simplification and enable auto-vectorization
  - Remove unecessary nested-loops

Code modification

- Optimize:
  - tracked one consuming call stacks: **sincosf64x** and **sqrt** function used in **boost::math::detail::bessel_j1**,
  - in **meas_base** package, in **src/SincCoeffs.cc**, modified **calcImageKSpaceCplx** (see on Github),

```cpp
// compute the k-space values and put them in the cimg array
double const twoPiRad1 = geom::TWOPI * rad1;
double const twoPiRad2 = geom::TWOPI * rad2;
double const scale = (1.0 - ellipticity);
for (int iY = 0; iY < wid; ++iY) {
    int const fY = fftshift.shift(iY);
    double const ky = (static_cast<double>(iY) - ycen) / wid;

    for (int iX = 0; iX < wid; ++iX) {
        int const fX = fftshift.shift(iX);
        double const kx = static_cast<double>(iX - xcen) / wid;

        // rotate
        std::pair<double, double> coo = rotate(kx, ky, posAng);
        double kxr = coo.first;
        double kyr = coo.second;
        // rescale
        double const k = ::sqrt(kxr * kxr + scale * scale * kyr * kyr);
```

```cpp
// compute the k-space values and put them in the cimg array
double const twoPiRad1 = geom::TWOPI * rad1;
double const twoPiRad2 = geom::TWOPI * rad2;
double const scale = (1.0 - ellipticity);
// pre-compute rotation matrix
double const cosAng = ::cos(posAng);
double const sinAng = ::sin(posAng);
// pre-compute fftshift and kpos
double const inv_wid = 1.0 / wid;
std::vector<int> fftShifted(wid);
std::vector<double> kpos(wid);
for (int i = 0; i < wid; ++i) {
    fftShifted[i] = fftshift.shift(i);
    kpos[i] = (static_cast<double>(i) - cen) * inv_wid;
}
// pre-compute k
std::vector<double> kValuesFlat(wid * wid);
for (int iY = 0; iY < wid; ++iY) {
    int const fY = fftShifted[iY];
    double const ky = kpos[iY];

    for (int iX = 0; iX < wid; ++iX) {
        int const fX = fftShifted[iX];
        double const kx = kpos[iX];

        // rotate
        double const kxr =   cosAng * kx + sinAng * ky;
        double const kyr = - sinAng * kx + cosAng * ky;

        // rescale
        kValuesFlat[fY * wid + fX] = std::sqrt(kxr * kxr + scale * scale * kyr * kyr);
    }
}
// compute c
for (int el = 0; el < wid*wid; ++el) {
    double const k = kValuesFlat[el];
```

- Idea in this:
  - Refactor code for simplification and enable auto-vectorization
  - Remove unecessary nested-loops
- Expected results:
  - More memory usage because pre-computation
  - Less runtime thanks to vectorization
  - Biggest chunk of time unaffected yet, cause Bessel function not vectorizable.

Elapsed Time: 719.315s

CPU Time:	698.510s
Effective Time:	696.870s
Spin Time:	1.640s
Overhead Time:	0s
Total Thread Count:	62
Paused Time:	0s

Vectorization : 9.5%

Elapsed Time: 719.315s

CPU Time:	698.510s
Effective Time:	696.870s
Spin Time:	1.640s
Overhead Time:	0s
Total Thread Count:	62
Paused Time:	0s

Vectorization : 9.5%

Elapsed Time: 603.749s

CPU Time:	591.960s
Effective Time:	590.570s
Spin Time:	1.390s
Overhead Time:	0s
Total Thread Count:	46
Paused Time:	0s

Vectorization : 10.4%

Elapsed Time: 719.315s

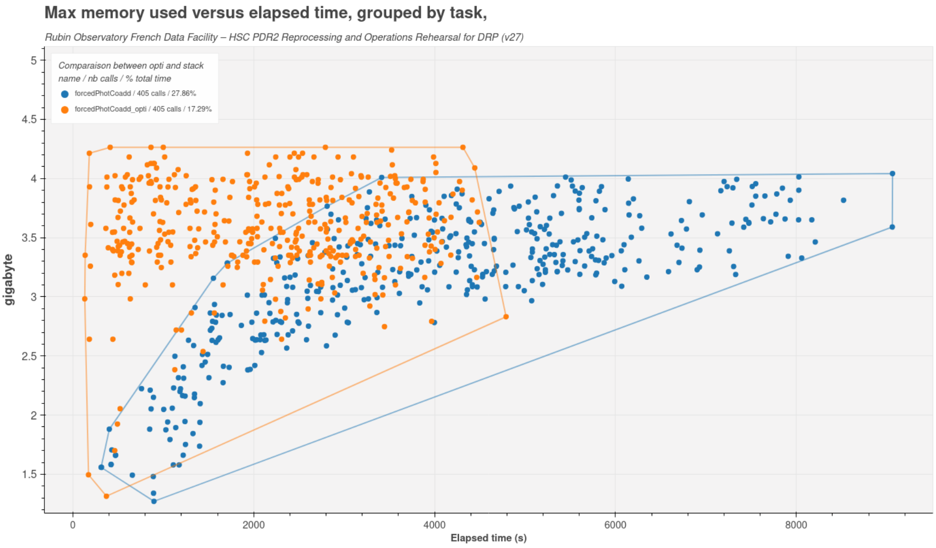
CPU Time:	698.510s
Effective Time:	696.870s
Spin Time:	1.640s
Overhead Time:	0s
Total Thread Count:	62
Paused Time:	0s

Vectorization : 9.5%

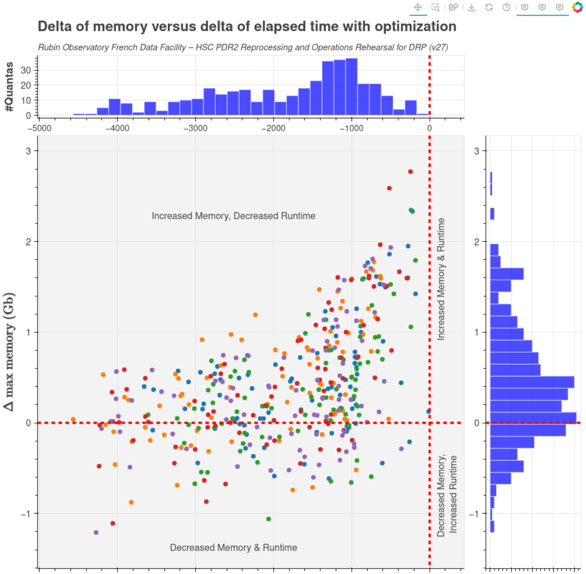
Elapsed Time: 603.749s

CPU Time:	591.960s
Effective Time:	590.570s
Spin Time:	1.390s
Overhead Time:	0s
Total Thread Count:	46
Paused Time:	0s

Vectorization : 10.4%



Max memory used versus elapsed time, grouped by task,
Rubin Observatory French Data Facility – HSC PDR2 Reprocessing and Operations Rehearsal for DRP (v27)
Comparaison between opti and stack
name / nb calls / % total time
forcedPhotCoadd / 405 calls / 27.86%
forcedPhotCoadd_opti / 405 calls / 17.29%
gigabyte
5
4.5
4
3.5
3
2.5
2
1.5
0
2000
4000
6000
8000
Elapsed time (s)



- Colors correponds to band,
- Every job runtime decreased,
- Approx 3/4 of the job memory increased,

	$\sum t_i$ (h)	$\sum m_i$ (Gb)	$\sum(t_i \cdot m_i)$ (Gb.h)
Whole Pipeline			
Base	1617.1	157045.7	4013.7
Opti	1410.5	157213.1	3367.1
Diff	206.6	-167.4	646.6
Relative (%)	-12.8%	0.1%	-16.1%
forcedPhotCoadd Task			
Base	450.5	1282.5	1525.5
Opti	243.9	1449.9	878.9
Diff	206.6	-167.4	646.6
Relative (%)	-45.9%	13.0%	-42.4%

- **Primary results:** 100s gain on a 700s run, seems great & code is more vectorized !
- **Global results:** -45.9% runtime for **forcedPhotCoadd** & -12.8% for the whole pipeline... for approx. 20 lines of codes

- **Primary results:** 100s gain on a 700s run, seems great & code is more vectorized !
- **Global results:** -45.9% runtime for **forcedPhotCoadd** & -12.8% for the whole pipeline... for approx. 20 lines of codes

Something is wrong !

Way too much performance gain ! That's suspicious !

- **Primary results:** 100s gain on a 700s run, seems great & code is more vectorized !
- **Global results:** -45.9% runtime for **forcedPhotCoadd** & -12.8% for the whole pipeline... for approx. 20 lines of codes

Something is wrong !

Way too much performance gain ! That's suspicious !

What could have gone wrong ?

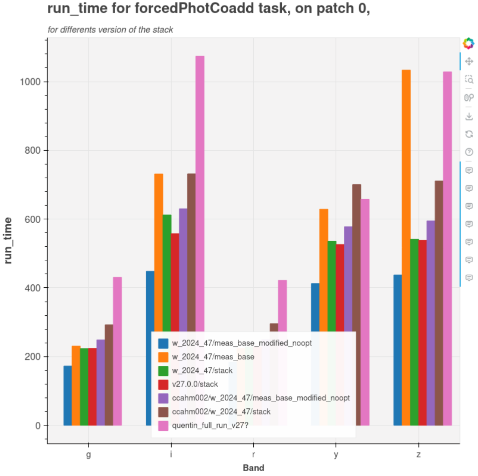
- Code improved between v27 and w_2024_47 ?
- Difference between interactive server and bacth-server ?
- Huge variability in runtime inherent to the processing ?

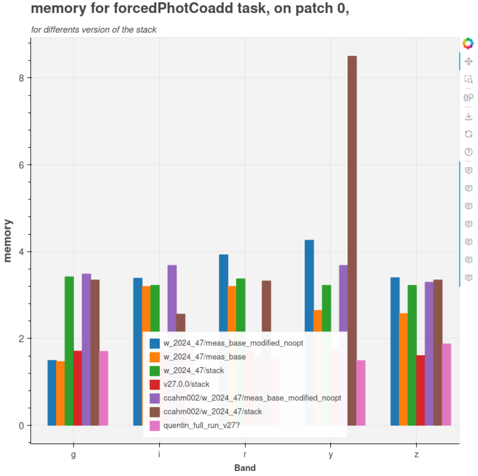
Machine impact

Interactive vs batch, & multiple stacks version

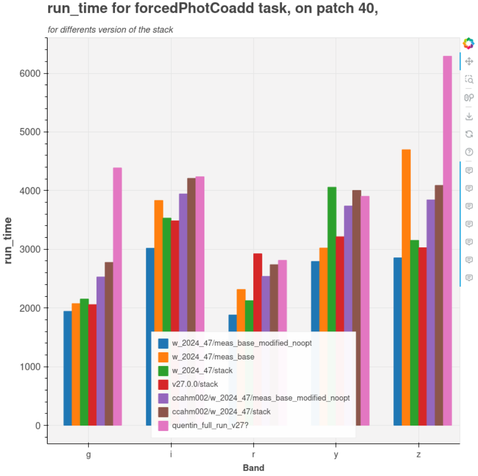
- **w_2024_47/meas_base_modified_noopt**: Using w_2024_47 of the stack, with package `meas_base` modified with base compilation option, run with `sbatch`.
- **w_2024_47/meas_base**: Using w_2024_47 of the stack, with package `meas_base` with base compilation option, run with `sbatch`.
- **w_2024_47/stack**: Using w_2024_47 version of the stack, run with `sbatch`.
- **v27.0.0/stack**: Using v27.0.0 version of the stack (dated approx. June 2024), run with `sbatch`.
- **ccahm002/w_2024_47/meas_base_modified_noopt**: Using w_2024_47 of the stack, with package `meas_base` modified with base compilation option, run on `ccahm002` interactive server.
- **ccahm002/w_2024_47/stack**: Using the w_2024_47 version of the stack, run on `ccahm002` interactive server.
- **quentin_full_run_v27?**: Results obtained from HSC-PDR2 run made by Quentin in early September 2024, using version v27.0.0. No info on the run's specifics.

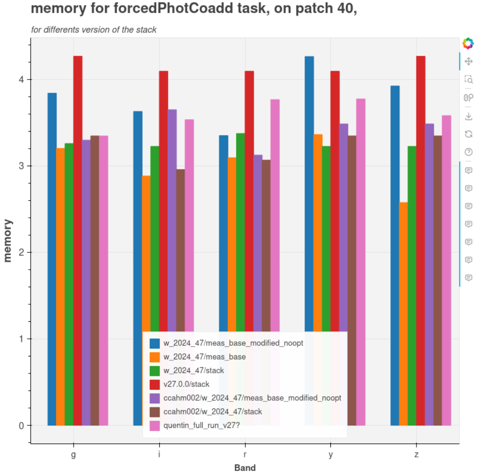
Patch 0


run_time for forcedPhotCoadd task, on patch 0,
for differents version of the stack
run_time
0
200
400
600
800
1000
g
i
r
y
z
Band
w_2024_47/meas_base_modified_noopt
w_2024_47/meas_base
w_2024_47/stack
v27.0.0/stack
ccahm002/w_2024_47/meas_base_modified_noopt
ccahm002/w_2024_47/stack
quentin_full_run_v27?

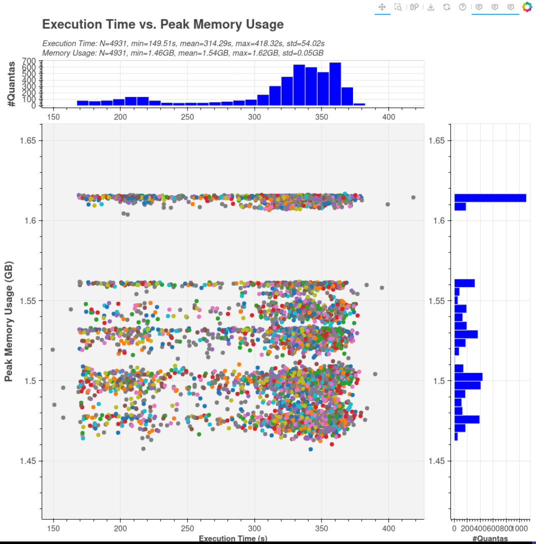

memory for forcedPhotCoadd task, on patch 0,
for differents version of the stack
memory
0
2
4
6
8
g
i
r
y
z
Band
w_2024_47/meas_base_modified_noopt
w_2024_47/meas_base
w_2024_47/stack
v27.0.0/stack
ccahm002/w_2024_47/meas_base_modified_noopt
ccahm002/w_2024_47/stack
quentin_full_run_v27?

Patch 40


run_time for forcedPhotCoadd task, on patch 40,
for differents version of the stack
run_time
6000
5000
4000
3000
2000
1000
0
g
i
r
y
z
Band
w_2024_47/meas_base_modified_noopt
w_2024_47/meas_base
w_2024_47/stack
v27.0.0/stack
ccahm002/w_2024_47/meas_base_modified_noopt
ccahm002/w_2024_47/stack
quentin_full_run_v27?

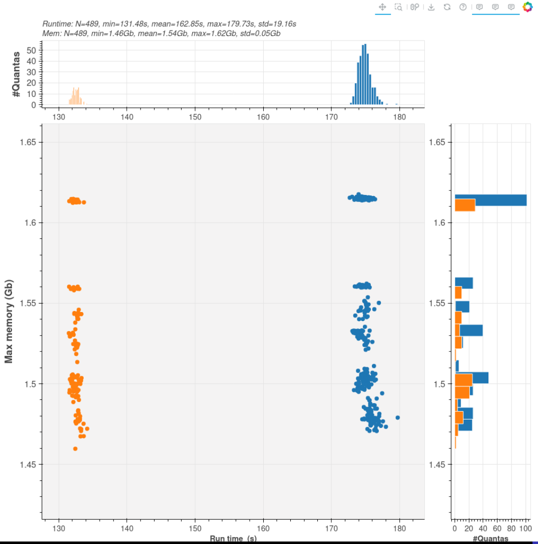

memory for forcedPhotCoadd task, on patch 40,
for differents version of the stack
memory
4
3
2
1
0
g
i
r
y
z
Band
w_2024_47/meas_base_modified_noopt
w_2024_47/meas_base
w_2024_47/stack
v27.0.0/stack
ccahm002/w_2024_47/meas_base_modified_noopt
ccahm002/w_2024_47/stack
quentin_full_run_v27?

Using all CC-IN2P3 node:

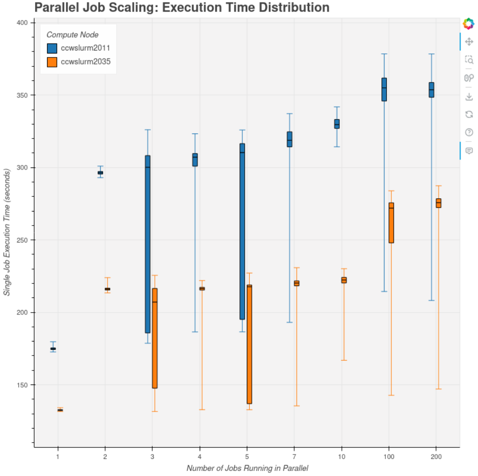


- Some memory threshold
- Runtime variability does not seems correlated to a particular server

Using 2 reserved nodes, 1 job at a time:



Using 2 reserved nodes:



- Difference between two nodes match nodes specs !
- Runtime bump even at 2 jobs in parallel seems to be slurm putting multiple thread on the same physical core,
- Hyperthreading ?
- Time per job in increase,

Notebooks here & more results here.

- Need to control the environnement to properly measure performance gain ...
- ... but we need to ensure our gains translate into a production environnement !

or

- We could just run $\tilde{1}00$ of runs to get a distribution and statistics each improvement,
- Coud work for short runtime tasks ...

Summary

- Need to control the environnement to properly measure performance gain ...
- ... but we need to ensure our gains translate into a production environnement !

or

- We could just run $\tilde{1}00$ of runs to get a distribution and statistics each improvement,
- Coud work for short runtime tasks ...

Strategy:

- Dedicate a machine to pursue testing on, simple fast case
- Test again with inputs thats take longer to process,
- Once improvements observed, tests how it scale on a loaded machine,