Using ParSL to Paralellize WaZP

Members: Rocío Zorrilla D.Sc.[1]
Eng. Carlos Cardoso [1]
Adolfo Simões [1]
Coordinator: Prof. Fábio Porto, D.Sc.[1]
[1]Laboratório Nacional de Compuação Científica (LNCC)

June 16, 2023.

Outline

1. Introduction
2. Methodology
3. Implementation and Results
4. Discussion and Analysis
5. Conclusions and Future Works

Outline

1. Introduction
2. Methodology
3. Implementation and Results
4. Discussion and Analysis
5. Conclusions and Future Works

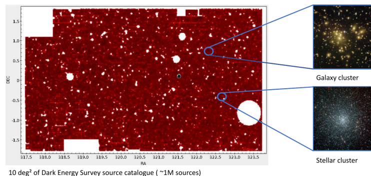
The Science Case

What is WaZP?

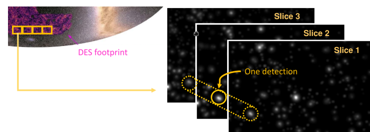
- WaZP: Galaxy cluster finder based on photometric redshifts called "Wavelet Z Photometric" in large optical surveys.
- Galaxy Cluster: a structure that consists of anywhere from hundreds to thousands of galaxies.
- Optical Survey: is a general map or image of a region of the sky (or of the whole sky) that lacks a specific observational target.
- WaZP must find Galaxy clusters on optical surveys:
  - DES: 5000 square degrees of the southern sky producing as much as 2.5TB/night.
  - LSST: a project to image the entire southern sky every three nights, roughly producing 20TB/night (Final database size (DR11): 15 PB)
- Implemented in Python 3.8.: Using Numpy for aggregation and merge processes. For astronomic processes mainly use HealPy, Sparse2D and AstroPy. Integrates a third-party specialized program to process data (MR-Filter in C).

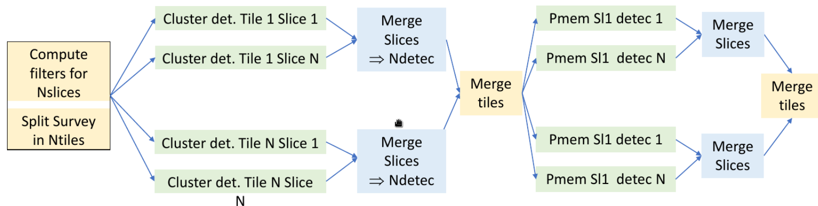
How it works.

Quick Overview for the WaZP Workflow



10 deg² of Dark Energy Survey source catalogue (~1M sources)





Project Objectives:

Main Objective

Implement a parallel version of WaZP to efficiently process large volumes of data in a distributed environment.

Parallelization Analysis:

- Analyze the original code and define the baseline environment.
- Performance analysis to detect data dependency, bottlenecks and critical code sections.
- Select a parallelism paradigm suitable for the problem.
- Implement the strategy adopted and carry out performance experiments.

Outline

1. Introduction

2. Methodology

3. Implementation and Results

4. Discussion and Analysis

5. Conclusions and Future Works

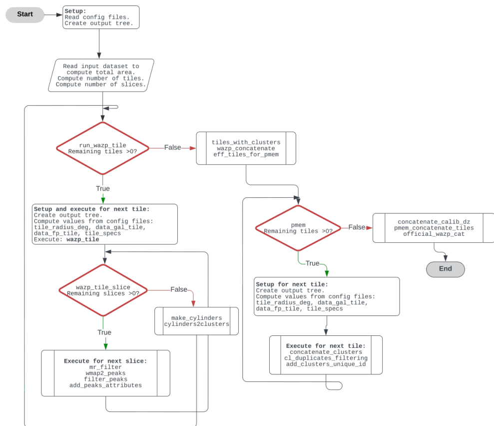
Using WaZP - The Sequential Version

Dataset `DC2_Test`

- The original WaZP Code is written sequentially.
- For the `DC2_Test` dataset which covers an effective area of approximately 10 square degrees.
- As a result are generated 3 tiles with 29 slices each.
- The total execution time is 58 minutes.
- Using `perf` and `palanteer`, performance tools, we extracted and identify:
  - Execution flowchart.
  - Processing bottlenecks.
  - Data dependency.

Code Analysis and Parallelization Opportunities

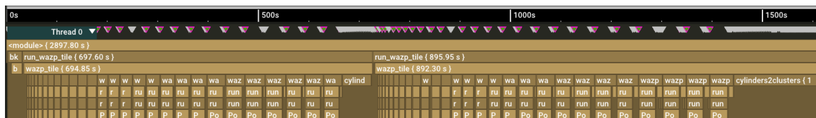
Original Code Flowchart

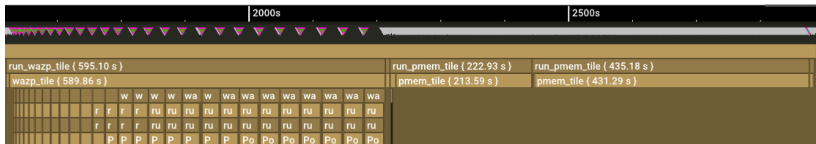


- `run_wazp_tile`: sequential tile processing, calls `wazp_tile_slice` to sequentially process slices.
- `run_pmem_tile`: sequential tile processing aggregating results from previous steps.

Profiling

- Using `perf`: Main : 2898s (100%)
  3x run_wazp_tile : 2188s (75.5%)
  3x run_pmem_tile : 666s (23%)
  bkg_global_survey : 28s (1%)
  wazp_concatenate : 7.28s (0.3%)
  pmem_concatenate_tiles : 2.68s (0.1%)
- Using Palanteer, a profiling tool, we acquire performance data and function dependency:





Performance Analysis Insights

Data Dependency

- The tile processing within the `run_wazp_tile` and `run_pmem_tile` functions present data independence, making them suitable to data parallelism (Bag of Tasks).
- The slice processing within the `wazp_tile_slice` function also have some data independence, but parallelization is not trivial because each slice lacks information about its own tile.
- Right after the `run_wazp_tile` execution, there is a procedure that waits for all the tiles be completed before the execution of the `run_pmem_tile` function.

Parallel Scripting Library – ParSL

- ParSL is a tool for enabling parallel and distributed computing in Python.
- Supports a wide range of execution environments, including multicore CPUs, GPUs, clusters, and cloud computing platforms.
- Automatically parallelizes and distributes tasks across available resources.
- Supports dynamic task scheduling and load balancing, optimizing resource utilization and performance.
- Offers fault tolerance mechanisms, allowing tasks to be retried or rescheduled in case of failures.
- Used in DES pipelines .

ParSL and SLURM

- PARSL configuration allow us to specify:
  - Number of compute nodes that will process slices. (num_nodes)
  - Number of slices that each node can process at the same time. (task_per_node).
- With PARSL it is possible to execute up to (num_nodes $\times$ task_per_node), but only if there are enough slices available.

```
executors=[
 HighThroughputExecutor(
     label='WaZP_SD',
     # one worker per manager/node
     max_workers=3,
     provider=LocalProvider(
         channel=LocalChannel(script_dir='.'),
         nodes_per_block=10,
         launcher=SrunLauncher(),
         cmd_timeout=120,
         init_blocks=1,
         max_blocks=1,
     ),
 )
```

Outline

1. Introduction

2. Methodology

3. Implementation and Results

4. Discussion and Analysis

5. Conclusions and Future Works

Parallelize the Tile Processing

We use ParSL to implement the data parallelism to process the tiles, for example for `run_wazp_tile`:

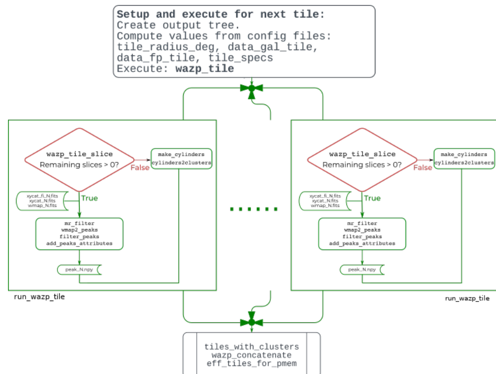


Figure: Process of Tiles in Parallel.

Implementation with ParSL

For the `run_wazp_tile` function

- Declare the decorator for the funtion to be parallelized:

```
@python_app
def run_wazp_tile(config, dconfig, thread_id):
    wazp code
```

- Implement an accumulator to wait all the tiles are processed:

```
# new list to accumulate partial tile outputs
tile_results = []

# iterate all tiles
for ith in np.unique(all_tiles['thread_id']):

    # call run_wazp_tile and save partial result
    tile_results.append(run_wazp_tile(config, dconfig, ith))

# retrieve the partial results
outputs_tiles = [r.result() for r in tile_results]
```

Experiment Setup

Computational Environment

- SDumont is composed by 36.472 CPU cores, distributed in 1.134 nodes (hyperthreading activated).
- The `cpu_shared` partition has available 480 CPUs, distributed across 20 nodes.
- Each user has the possibility to execute 96 jobs simultaneously with a maximum of 96 hours to execute the job.
- RedHat Linux 7.6, using the Software Stack Bull Supercomputing Cluster Suite 5 for nodes management.
- SLURM(Simple Linux Utility for Resource Management) for cluster management and job scheduling system.
- FileSystem Lustre v2.12 as parallel distributed file system.

Experiment Setup

- Configuration for the DC2_400_simulation dataset:
  - In `wazp.cfg` and possibly change the value for `NSide` (control the size of the tile).
    1. Change the name of the dataset to be processed.
    2. Possibly change the value for the `Nside` variable, to control the size of each tile.
  - In `data.cfg`:
    1. Change the input data structure, in particular for the footprint.
    2. Add the `galcat` options and values.
    3. Add the `footprint` options and `PATH` for the footprint file.
    4. Add the `magstar_file` options and values.
    5. Add the `zp_metrics` options and change some values.
- For this dataset that represents 440 square degrees, WaZP generates 49 tiles with 87 slices for each tile.

Performance Metrics

Speedup and Efficiency

- Speedup: Measures how well the application scales when adding new computing devices to the execution environment.

$$S(N) = \frac{t(1)}{t(N)} \quad (1)$$

  where:
  - $t(1)$ – execution time of the sequential application running on a system with 1 computing device,
  - $t(N)$ – execution time of the application on a system with N computing devices.

- Efficiency: Measure of how effectively a parallel algorithm or system utilizes available computational resources.

$$pe(N) = \frac{t(1)}{N \cdot t(N)} \quad (2)$$

Outline

1. Introduction

2. Methodology

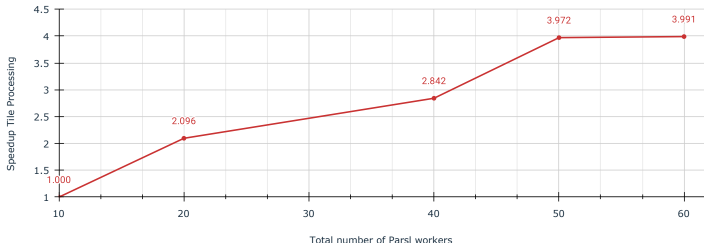
3. Implementation and Results

4. Discussion and Analysis

5. Conclusions and Future Works

Experiment 1: ParSL for Tile Processing with `run_wazp_tile`

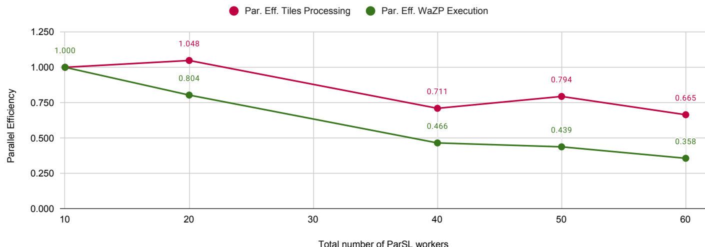
Speedup



- Non-linear speedup, multiple processes almost invariably introduces some overhead.
- For Total ParSL workers above to 50 the speedup has a parallel slowdown, diminished performance.

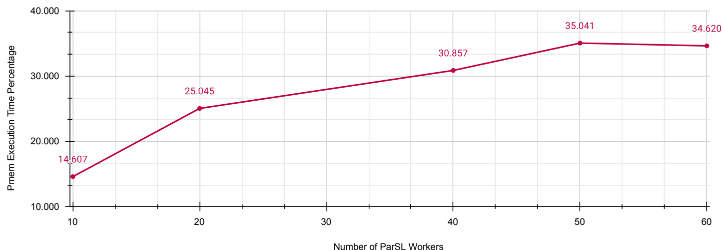
Experiment 1: ParSL for Tile Processing with `run_wazp_tile`

Efficiency



- High efficiency means the effective use of available resources and achieving good performance gains from parallelism.
- The unbalanced datasets to be processed on each computational node inside the wazp tile slice function
- Total execution time reduced to 5 hours.

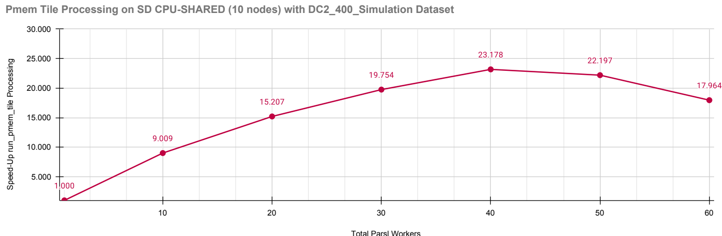
What about run_pmem_tile?



- By reducing the run_wazp_tile execution time, the sequential execution of the run_pmem_tile function start to be considered as a bottleneck.
- The Figure shows hows the run_pmem_tile percentage increases.

Experiment 2: ParSL for Tile Processing with `run_pmem_tile`

Speedup



- Non-linear speedup, multiple processes almost invariably introduces some overhead.
- For Total ParSL workers above to 50 the speedup has a parallel slowdown, diminished performance.

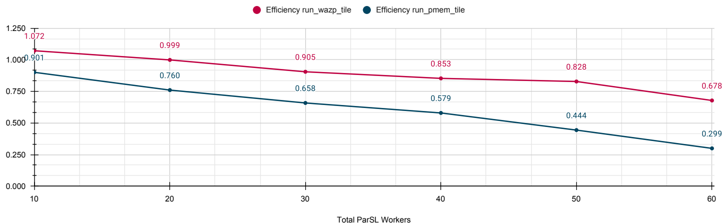
Experiment 2:ParSL for Tile Processing with `run_wazp_tile` and `run_pmem_tile`

Speedup

- Non-linear speedup, multiple processes almost invariably introduces some overhead.
- For Total ParSL workers above to 50 the speedup has a parallel slowdown, diminished performance.

Experiment 2: ParSL for Tile Processing with `run_wazp_tile` and `run_pmem_tile`

Efficiency

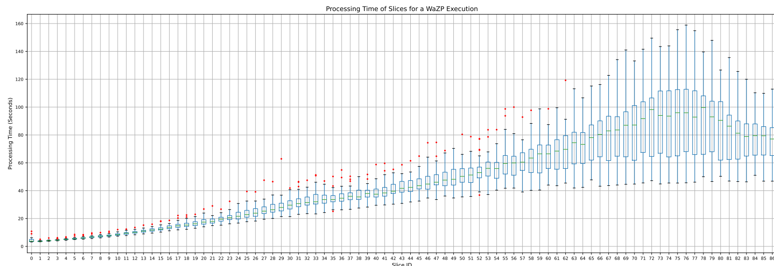


- High efficiency means the effective use of available resources and achieving good performance gains from parallelism.
- The unbalanced datasets to be processed on each computational node inside the wazp tile slice function
- Total execution time reduced to 3 hours.

DEXL
DATA EXTREME LAB

Unbalanced Tiles and..... MR-Filter

- Each slice is generated based on the red-shift measurement and in consequence the amount of information to process for slice would increase and also the file size.
- for the shallow slices, the measurements of the processing time are tightly clustered and have low variance. For the deeper slices each box-plot has a wider variation which indicates that the amount of objects in the tile affect the size of the slice.



mr_filter code

- OpenMP process that generates one thread for each CPU core available in the system.
- Experiments shows that we can increase performance when running multiple instances of `mr_filter` in the same node (`workers_per_node`.
- We execute `mr_filter` manually with `perf` to analyze the CPU utilization:

```
# nrcpus online : 16
# nrcpus avail : 16

#
# Overhead       sys       usr       Samples  CPU
# ........  ........  ........  ............  ...
#
    45.40%     0.85%    44.55%         53119  011
    17.52%     0.59%    16.93%         21235  012
     4.40%     0.55%     3.85%          5687  003
     3.66%     1.19%     2.47%          5605  002
     3.23%     0.57%     2.66%          4922  004
     2.86%     0.65%     2.22%          4460  006
     2.77%     0.26%     2.51%          3745  015
     2.71%     0.59%     2.11%          4723  005
     2.43%     0.61%     1.82%          3640  001
     2.40%     0.49%     1.91%          4045  014
     2.36%     0.62%     1.74%          4173  000
     2.19%     0.42%     1.77%          3489  007
     2.12%     0.44%     1.68%          2872  009
     2.03%     0.35%     1.68%          3133  008
     1.99%     0.33%     1.66%          2775  010
     1.94%     0.29%     1.65%          2765  013
```

- We can use the `OMP_NUM_THREADS` variable to control the amount of concurrent processes.

Outline

1. Introduction

2. Methodology

3. Implementation and Results

4. Discussion and Analysis

5. Conclusions and Future Works

Conclusions

- It is possible to parallelize the workload of WaZP with Parsl after identifying key steps in the execution flow that contain independent tasks.
- Parsl is only able to efficiently parallelize the workload as long as there are enough tasks: the speedup gets “saturated” when increasing the worker count and we achieve a plateau of performance.
- Current solution parallelizes the tiles; but improving to use slices can remedy the drawback of having too few tasks for Parsl to distribute among workers.
- The CPU-Shared partition was readily available for job submissions with low waiting time. However, resource sharing may interfere with obtaining accurate performance results.
- The CPU-Shared partition was useful to validate the proposed approach, but other more powerful partitions, with significantly larger waiting queue time, should be used for the largest datasets.

Future Works

- Tile parallellization reach to a slowdown based on the maximum number of tiles and the resources available, it is possible to overcome this limitation by having a data partitioning based on the slices instead the tiles.
- Improved load balancing and task construction enhance parallel efficiency by minimizing idle time for workers. Efficient scheduling and maintaining result integrity involve estimating task complexity, prioritizing larger tasks, and organizing partial results for proper input consistency.
- When discussing the results of `mr_filter`, we mentioned the existence of the `OMP_NUM_THREADS` variable.

Thank you!