Tutorial: Optimizing Data Processing for High-Memory, Multi-Core Systems
The Case of Deciphering CBP Output

This tutorial demonstrates how to optimize a data processing workflow on a high-memory, multi-core system. We begin by detailing the task and code logic, then analyze the system configuration and identify a suitable parallelization strategy. Finally, we show the performance improvements achieved and provide alternative approaches for different hardware setups.

1 Task Logic Overview

The task involves processing data from a large dataset with:

1) nwcbox Simulation Cells: 56,920 cells, each representing a spatial data chunk.
2) Time Span: 3,650 days of data (approximately 10 years of daily data).
3)Total Data Size: ~50 GB, which is manageable in RAM on our system.

Each day's data includes independent processing for each nwcbox simulation cell, which makes the task highly parallelizable.

2 Serial Code Logic

In the serial code, each day's data is processed sequentially with the following structure:

- Load Configuration and Initialize Data:
  - Loads configuration and metadata files.
  - Sets up arrays for temporarily storing results.
- Daily Processing Loop:
  - For each day (jday), the code:
  - Reads binary data for that day.
  - Processes each nwcbox simulation cell independently.
  - Writes each cell's result immediately to the NetCDF file.
- Data Writing to NetCDF:
  The serial code writes each simulation cell's data to the NetCDF file immediately after processing. This approach creates high I/O overhead, as every cell's data is written individually.

This structure makes the code run more slowly due to frequent disk I/O, even though each cell can be processed independently.

3 System Configuration

Our system has:

CPU: Intel Xeon Gold 6148 with 80 cores, ideal for parallel processing.

Memory: 754 GB of RAM, which allows large data to be held in memory, reducing disk I/O.

NUMA Nodes: 2, which is less critical here since each simulation cell processes independently.

This setup supports high-volume parallel processing and efficient memory-based operations, enabling us to optimize I/O.

4 Profiling Results of the Serial Code

Date	Stage	Memory Usage (MB)	Compute Time (s)	I/O Time (s)	Total Time (s)	Daily Memory Increase (MB)
1990-12-31	End of Day 1	174.03	58.30	331.55	389.85	+56.80
1991-01-01	End of Day 2	205.67	57.74	329.74	387.48	+31.64
1991-01-02	End of Day 3	237.24	58.43	326.43	384.86	+31.57

In the serial code, compute time is about 58 seconds per day, but I/O time (averaging around 330 seconds per day) is the bottleneck due to frequent writes. The parallel code will address this through a bulk I/O strategy.

5 Optimization Strategy: Leveraging System Resources

5.1 Why We Chose joblib.Parallel

We selected joblib for its ease of use in parallelizing independent tasks without complex setup. With joblib.Parallel, Python can manage tasks across multiple cores with controlled memory sharing. Key reasons for choosing joblib:

a. **Ease of Use:** joblib.Parallel provides a simple interface for distributing loop iterations across cores.

b. **Efficient Memory Management:** joblib handles shared memory by making data copies when necessary, reducing contention for large data arrays.

c. **Controlled Core Usage:** We can easily specify core count, making it easy to tune based on system resources.

5.2 Parallel Strategy for Processing and I/O

To optimize the processing workflow, we implemented a parallel processing strategy with efficient I/O accumulation. This approach leverages our system's memory and multi-core capabilities to maximize performance. Here are the two main strategies we used:

5.2.1 Parallel Processing of nwcbox Simulation Cells

Each nwcbox simulation cell can be processed independently, so we parallelize its processing using joblib.Parallel. Initially, we tested a range of core counts (from 5 to 60) and found that using 30-40 cores provided optimal performance on our system. joblib.Parallel allows us to assign tasks to multiple cores efficiently, making it well-suited for independent tasks like this.

Example code for parallel processing with joblib:

```
from joblib import Parallel, delayed

def process_nwcbox(jday, nwcbox_val):
    # Independent processing for each nwcbox simulation cell
    computed_values = compute_values(jday, nwcbox_val)
    return computed_values

# Parallel processing example with 16 cores
results = Parallel(n_jobs=16)(
    delayed(process_nwcbox)(jday, nwcbox_val) for nwcbox_val in range(1, nwcbox + 1))
```

5.2.2 Bulk Data Accumulation for Efficient I/O

To minimize I/O time, we accumulate results for each variable in a large array. By writing data to the NetCDF file once per day instead of per cell, we significantly reduce I/O overhead. This approach leverages the system's large memory to store data temporarily before writing it to disk in a single operation, which is faster than frequent writes.

Here's how we

a. *initialize the arrays for bulk accumulation*:

```
# Initialize arrays to store values for bulk writing
calc_values = {var_name: np.full((n_cells, n_layers, n_days), np.nan) for var_name in calc_var_names}
var_values = {var_name: np.full((n_cells, n_layers, n_days), np.nan) for var_name in var_names}
# Inside the parallel loop, accumulate results in large arrays
```

```
for result in results:
    cell_id, layernumber, jday, computed_values, lon, lat, Depth, nwcbox_val, c1_array_nwcbox = result
    for var_name in calc_var_names:
        calc_values[var_name][cell_id - 1, layernumber - 1, jday - start_jday]
=computed_values['selected_values'][calc_var_names.index(var_name)]
```

b. Accumulating Data in Large Arrays

During each day's processing, results are stored in large arrays for all cells and layers. This enables a single I/O operation at the end of each day, which minimizes I/O load and makes processing more efficient:

```
# Accumulating results into arrays for bulk writing
for result in results:
    cell_id, layernumber, jday, computed_values = result
    # Store computed values in calc_values and var_values
```

c. Bulk I/O Operation

Instead of writing each cell's data to disk immediately, results are accumulated in memory and written to NetCDF in a single operation at the end of the day. This strategy significantly improves efficiency by reducing the frequency of disk writes:

```
# Writing accumulated data once per day
netcdf_file.variables['nwcbox'][:, :, :] = nwcbox_vals  # Bulk write to NetCDF
netcdf_file.close()
```

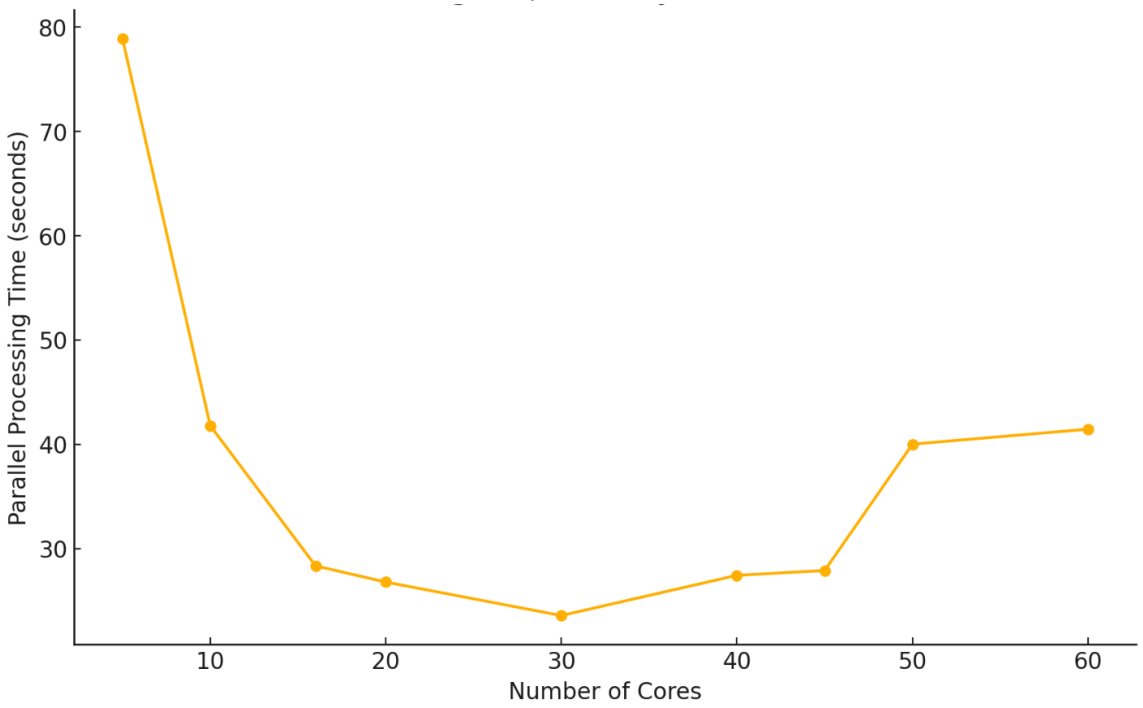
By combining these parallel and bulk I/O strategies, we achieved faster computation times and minimized I/O bottlenecks.

5.2.3 NUMA Considerations

Each nwcbox cell processes independently, so memory access across NUMA nodes is minimal, and no explicit NUMA optimizations are required.

6 Parallel Code Profiling Results and Analysis

6.1 *Attention: More Cores Don't Always Mean Less Time*



While adding cores initially decreases compute time, there's a limit to this benefit. As we observed from testing with up to 60 cores, using more cores beyond a certain point didn't yield additional speed gains due to the nature of joblib.Parallel:

6.2 How joblib.Parallel Works Beyond Computation

a. Allocation and Setup: When Parallel is called, it allocates necessary resources across the specified cores, including memory allocation for each task and setting up data partitions.
b. Parallel Processing: Each core processes its assigned task independently. While this speeds up compute time initially, it also leads to contention if too many cores are accessing the same memory resources, especially across NUMA nodes.
c. Data Gathering: After each core finishes its task, Parallel collects results from all cores, which can add overhead. This gathering phase increases as core counts grow, resulting in diminishing speed gains.

Tip: To optimize performance, always test the code with a smaller subset of the data. Plot performance metrics to identify the "sweet spot" for the number of cores, balancing computation and aggregation time. This method helps avoid the diminishing returns of over-parallelization, especially for tasks involving significant data collection.

6.3 Considerations for Advanced Users: When NUMA Matters

This specific task does not require significant cross-CPU communication because each simulation cell can be processed independently. However, in more complex applications

where frequent data exchange between CPUs is needed, NUMA considerations become crucial. In those cases, one might prefer to process all tasks within the same NUMA node (typically 20 cores per node) to reduce cross-node memory access overhead.

6.4 Parallel Code Profiling Results and Analysis

In the parallel code, compute time and memory usage were optimized as shown below, by using 16 cores:

Date	Stage	Memory Usage (MB)	Compute Time (s)	I/O Time (s)	Total Time (s)	Daily Memory Increase (MB)
1990-12-31	End of Day 1	409.79	33.39	—	33.39	+94.02
1991-01-01	End of Day 2	489.98	35.92	—	35.92	+80.19
1991-01-02	End of Day 3	489.07	32.22	1.33	33.55	-0.91

6.5 Key Improvements with Parallel Code

- Compute Time Reduction: Parallel processing reduced daily compute time to ~33 seconds by utilizing multiple cores, achieving around a 40% improvement.
- Optimized I/O: Using bulk writes reduced I/O time from 330 seconds per day in the serial code to just 1.33 seconds per day, addressing the serial code's I/O bottleneck effectively.
- Predictable Memory Growth: Holding data in memory for bulk I/O led to controlled memory usage, with a stable daily increase after the initial allocation.

7 Alternative Strategy for Systems with Fewer Cores or Memory

For systems with fewer cores or limited memory, you can adapt the parallelization strategy to better fit available resources:

- **Accumulate Data Across a Smaller Range of Days**: Instead of accumulating data for a large number of days, consider limiting the accumulation range to a smaller number of days to manage memory usage effectively. This reduces the number of I/O operations while keeping memory usage within a manageable limit.
- **CPU-Limited Systems**: If CPU resources are limited, consider running the code sequentially without parallelization. You can still accumulate results in larger arrays to avoid frequent I/O operations. Since I/O is the primary time-consuming aspect, reducing the frequency of I/O while performing sequential processing can still achieve performance gains.