Isoefficiency Analysis

Isoefficiency function: The Isoefficiency function tells you the ratio in which you have to increase the process count and problem size to maintain a constant efficiency.

Efficiency: Efficiency is represented as the ratio between the total amount of time spent by the serial algorithm (t_1) and the total amount of time spent across all processes of the parallel algorithm ($t_p \times p$). The total amount of time spent across all processes could also be represented as the time needed by the serial algorithm (the useful work done by the parallel algorithm) plus the additional overhead time (idle time, communication time, etc.) to operate the parallel algorithm ($t_1 + t_o$). Using this alternative representation of time spent across all processes, we can derive the equation for efficiency below:

$$Efficiency = \frac{t_1}{t_p \times p} = \frac{t_1}{t_1 + t_o} = \frac{1}{1 + \frac{t_o}{t_1}}$$

Looking at this equation, if t_o/t_1 stays constant, efficiency stays constant. Generally, t_1 will be a function of n, where n is the problem size (e.g. number of elements for sorting, cells in a 2D stencil, etc.), and t_o will be a function of both n and p, where p is the number of processes. Thus, given a constant K, where $Efficiency = \frac{1}{1+K}$, you can determine a function $N(p)$ (Isoefficiency Function) from $K = \frac{t_o}{t_1}$ that shows how much the problem size n needs to scale up or down for the parallel algorithm to maintain the same efficiency with a different amount of processes.

Example of Isoefficiency Analysis with Board Decomposition

The following problems will be working with square boards of size n, where the sides are $\sqrt{n}$ in length. The first problem will look at the case of 1D decomposition, while the second will consider the case of 2d decomposition.

For both problems, t_1 will have the same value as they originate from the same serial algorithm, which would be n as a calculation would have to be done for every cell in the board. To approximate t_o, we will analyze the communication calls needed for the parallel algorithms to operate, and we will count sends and receives as pairs to not double count the operations since there is a send for every receive that is done roughly under the same overhead time. For each example, p represents the number of processes being used for the algorithms.

1D Decomposition

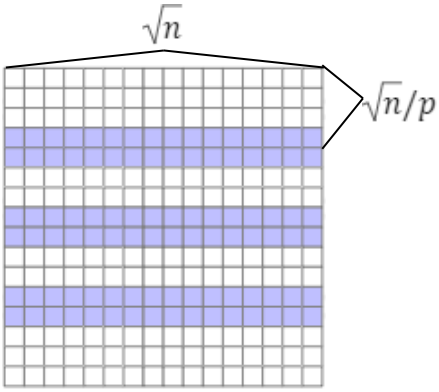


Figure 1. Board under 1D decomposition, which boundary elements in blue.

Under 1D decomposition, each process gets $\sqrt{n}/p$ rows, each with $\sqrt{n}$ elements. For the algorithm, each process needs to communicate with its two neighbors (one above and one below) to get their boundary rows as well as send them its own boundary rows for the processes to do their calculations on the elements in their boundary rows. This would have a cost of $2\sqrt{n}$ since each process has to send and receive two rows of $\sqrt{n}$ elements.

Because there are p total processes, the total number of communication calls is $2p\sqrt{n}$, which represents t_o for this problem. This means 1D decomposition has a t_o/t_1 ratio of

$$\frac{t_o}{t_1} = \frac{2p\sqrt{n}}{n} = \frac{2p}{\sqrt{n}},$$

which means, for the ratio to remain constant, n has to scale quadratically with respect to increases in p.

2D Decomposition

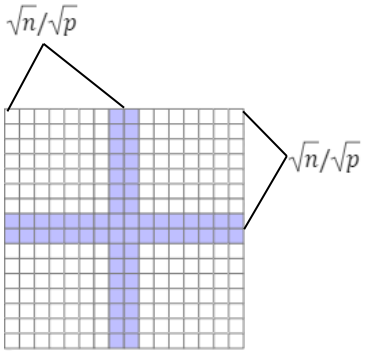


Figure 2. Board under 2D decomposition, which boundary elements in blue.

Under 2D decomposition, each process gets a square of side lengths $\sqrt{n}/\sqrt{p}$. For the algorithm, each process needs to communicate with four neighbors to send and receive their boundary elements. For each of their neighbors, they have to send/receive rows or columns of $\sqrt{n}/\sqrt{p}$

elements. Because there are p total processes, the total communication calls is $4\sqrt{np}$, which represents t_o for this problem. This means 2D decomposition has a t_o/t_1 ratio of

$$\frac{t_o}{t_1} = \frac{4\sqrt{np}}{n} = \frac{4\sqrt{p}}{\sqrt{n}},$$

which means, for the ratio to remain constant, n has to scale linearly with respect to increases in p. Because it is usually easier to scale the problem size linearly than quadratically, 2D decomposition could be stated to be more isoefficient than 1D decomposition.

Additional Note for Isoefficiency Analysis

Instead of calculating the total t_1 and t_o across the entire parallel algorithm, we could have instead only looked at the time and overhead taken by one process if that process did a representative amount of work that all processes did. This is allowed because, if the process did a representative amount of work, we could obtain t_1 and t_o by multiplying the work and overhead time of that process by p, but doing so won't change the ratio, hence using only one process would have been fine.

Empirical Performance Analysis

Empirical performance analysis involves running the algorithms versus the formula based methods of the analytical approaches.
There are two parts to empirical performance analysis:

(1) Measuring the performance of the algorithms, hopefully with representative and exhaustive test data.
(2) Analyzing and visualizing the obtained measurements from running the algorithms to determine points where performance is nonoptimal.

Simple measurements

The simplest way of measuring the runtime of algorithms is using timers and print statements. For each section of the code to be measured, surround it with time calls to get the time before executing the algorithm and the time after executing the algorithm, which can then be used to calculate the time taken by the algorithm. This value can then be printed out and put in a data table for later analysis.

Example timer usage in MPI

```
//Declaring variables
double start_time, end_time, elapsed_time;

start_time = MPI_Wtime(); //Getting time at start of code execution
{
        //Code to be executed
}
```

```
end_time = MPI_Wtime(); //Getting time at end of code execution
elapsed_time = end_time - start_time; // Getting time taken by code execution
```

To get a more fine-grained analysis of the algorithm, you can divide the algorithm into different phases and put timers on each phase of the algorithm to better determine the inefficient portions of the code.

Drawbacks

While this approach is simple, it becomes very tedious and infeasible for larger algorithms with many different sections of interest. Manually adding all the timers and print statements takes a lot of time, which brings us to performance tools.

Performance Tools

Other than manually using timers, there are other tools that can be used to measure the performance of an algorithm. There are **tracing** tools that capture the entire execution trace, **profiling tools** that provide aggregated information, and hybrids of both. These tools provide metrics like functional call counts, time spent in different portions of the code, and byte counts of different messages, which can be used to analyze the correctness of the code and where/how it may be optimized.

In addition to these metrics, there are a variety of **hardware counters**, depending on the hardware specifications of the system the code is being run on. Example hardware counters include the number of branching instructions, cache hits and misses, floating point operations, etc. Depending on the exact hardware counters you have access to or if the code is being optimized for a specific system, these details can provide more specific information relevant to the algorithm for better optimizations.

Tracing Tools

Tracing tools record all the events in the program, like function calls or MPI events, with timestamps, which they do via instrumentation.

Instrumentation: Instrumentation is the process of adding additional code to the original to measure the performance of the code. An example of manual instrumentation is the simple timer measurement method mentioned earlier in the notes.

Tracing tools do automatic instrumentation by adding code to your code itself. Doing so could cause the program to take a longer amount of time to run because of the additional overhead, but it will provide a fine-grained overview of what each process was running at any given time throughout the duration of the program. However, because of the additional overhead, the performance of the algorithm with the instrumentation may not be representative of how the algorithm would normally run due to potential differences in function execution timing (as in when functions get started in the program).

Example Tracing Tools

Examples of these tools include VampirTrace, Score-P, TAU, Projections, and HPCToolkit. There are also architecture-specific tools like Nsight, designed for Nvidia GPUs, and OmniTrace, designed for AMD GPUs.

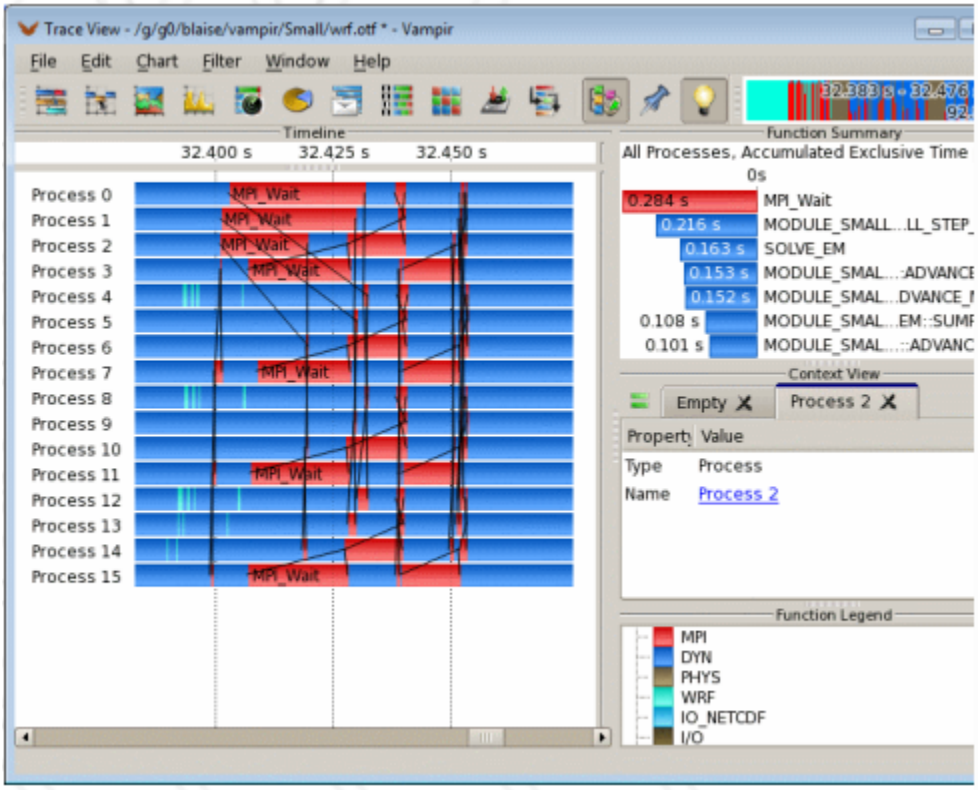


Figure 3. Example analysis GUI for the tracing tool Vampir, showing the time frames in which MPI_Wait ran on different processes in a program.

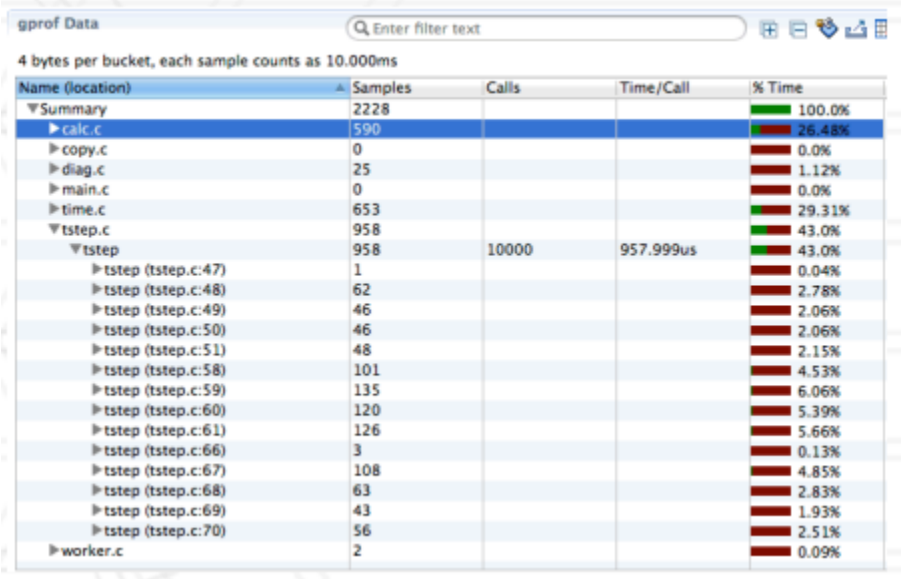
Profiling Tools

Unlike tracing tools, profiling tools ignore the specific times when events occurred and provide aggregate information about different parts of the code instead. For example, they won't keep track of each timestamp of every call of each function, but will return the total amount of time spent in each function across all their calls.

Profiling tools can use instrumentation and then aggregate the data. However, more often, it does this by **sampling**, where it looks at the program counter and the instructions being run. The profiling tools can then correlate the instructions with the actual code, and the statements in the code that the tools correlate more often are ones that they believe are being run more often. By using sampling, there is a smaller impact on the program's performance since it is less intrusive and requires less information from the program itself. By sacrificing the level of detail that tracing tools profile, profiling tools get the benefits of being easier to use and move their data output around.

Example Tracing Tools

Examples of these tools include Gprof, perf, mpiP, HPCToolkit, and caliper. Tools like Gprof are made primarily for sequential algorithms and it can be used by using the '-pg' when compiling C/C++ code, producing an output viewable in a Gprof GUI. mpiP is a profiler made for MPI and mostly keeps track of MPI function calls throughout a program. Some example Python profilers include cprofile, pyinstrument, and scalene.



Name (location)	Samples	Calls	Time/Call	% Time
Summary	2228			100.0%
calc.c	590			26.48%
copy.c	0			0.0%
diag.c	25			1.12%
main.c	0			0.0%
time.c	653			29.31%
tstep.c	958			43.0%
tstep	958	10000	957.999us	43.0%
tstep (tstep.c:47)	1			0.04%
tstep (tstep.c:48)	62			2.78%
tstep (tstep.c:49)	46			2.06%
tstep (tstep.c:50)	46			2.06%
tstep (tstep.c:51)	48			2.15%
tstep (tstep.c:58)	101			4.53%
tstep (tstep.c:59)	135			6.06%
tstep (tstep.c:60)	120			5.39%
tstep (tstep.c:61)	126			5.66%
tstep (tstep.c:66)	3			0.13%
tstep (tstep.c:67)	108			4.85%
tstep (tstep.c:68)	63			2.83%
tstep (tstep.c:69)	43			1.93%
tstep (tstep.c:70)	56			2.51%
worker.c	2			0.09%

Figure 4. Example analysis GUI for Gprof data in hpctView, showing the samples of different functions in a program and the times spent in each function.

Calling contexts, trees, and graphs

Besides only keeping track of the sum metrics of different functions (**flat profile**), profiles can also keep track of the calling context into specific instances of function calls to get a bit more information surrounding them.

The **calling context** or **call path** is the sequence of function invocations leading to a given sample function call. Merging all the call paths of a program together results in the program's **calling context tree** (**CCT**), which is a dynamic prefix tree of all call paths in an execution. The root of the CCT is the main function, as that is where all functions originate.

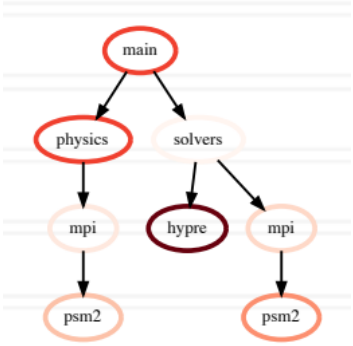


Figure 5. Example Calling Context Tree. Each node represents a function call, with arrows showing which function called which other functions (e.g. physics calls mpi).

In a CCT, most of the timings are stored in the leaf nodes, with the timings in the inner nodes only keeping track of time spent specifically in their functions and not the time spent in functions they called. The time spent specifically in these inner nodes is referred to as **exclusive time**, while the time including the time spent in their children is **inclusive time**.

Besides a calling context tree, there are also call graphs. **Call graphs** result from merging all the nodes of the same function together, which loses the specific call paths for each instance of each function, but it still maintains the caller-callee relationships between functions. This representation helps to reduce the memory required to store the profiler output compared to a calling context tree but with more information than a flat profile.

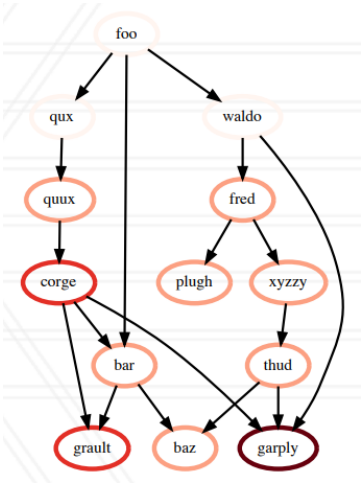


Figure 6. Example Call Graph. Like the CCT in Figure 5, arrows show caller-callee relationships, but there is only one node per function in the call graph.

Using Calling contexts, you can more finely analyze the context of each function call instead of aggregating them all together, so you can see the context in which a function call is taking the most amount of time. That way, you can better optimize the specific calls of a function that are causing the most performance hits instead of trying to modify every call of the function.

Hatchet

Hatchet is a tool developed to analyze parallel profiles more programmatically rather than through the GUI tools usually provided by the performance tools themselves. The goal of the tool is to provide much more flexibility with regard to how the measurements are analyzed. Hatchet leverages Pandas, an open-source Python library used for data analysis of tabular data structures, by creating what it calls a GraphFrame.

A **GraphFrame** is how Hatchet modifies the calling context trees and graphs into the preferred table structure of Pandas. Hatchet does this by using the nodes of the graph as the index of the

DataFrame, as well as keeping track of caller-callee relationships between the nodes in a separate column. By building GraphFrames on top of Pandas's Dataframes, Hatchet can take advantage of many functions that exist in Pandas to analyze calling context profiles, like Pandas's filter functions, column-wide modification, column addition/subtraction, and more.

Related Sources

Class Presentation (Images)

Bhatele, A., (Feb. 20, 2024) *Performance Modeling, Analysis, and Tools* [PowerPoint presentation]. https://www.cs.umd.edu/class/spring2024/cmsc416/slides/04-cmsc416-perf-analysis.pdf

Isoefficiency

Grama, A. Y., Gupta, A., & Kumar, V. (1993). Isoefficiency: Measuring the scalability of parallel algorithms and Architectures. *IEEE Parallel & Distributed Technology: Systems & Applications*, *1*(3), 12–21. https://doi.org/10.1109/88.242438