CMSC416: Introduction to Parallel Computing

Topic: Performance Modeling, Analysis, and Tools pt.2
Date: February 22, 2024

Isoefficiency

Isoefficiency is a parallel performance metric that allows us to relate problem size to the number of processes required to maintain a constant efficiency. Problem size varies from program to program, but as an example, for a Game of Life simulator, problem size would be the size of the board/grid.
In effect, isoefficiency describes a program's scalability with respect to the number of processes running, their speed, and any overhead there might be. Examples of overhead include I/O, communication between processes (such as sending and receiving messages), time to send on network, other delays, and receiving processes waiting to receive. This is essentially any time spent on non-useful/essential computation.

To derive the isoefficiency function, we can start from the efficiency function:

$$Efficiency = \frac{t_1}{p \times t_p}$$

Then, we can relate the denominator, which represents the total time spent in all processes, to be in terms of 'useful' work, and overhead:

$$p \times t_p = t_1 + t_0$$

Substituting this into the efficiency function, we get:

$$Efficiency = \frac{t_1}{p \times t_p} = \frac{t_1}{t_1 + t_0} = \frac{1}{1 + \frac{t_0}{t_1}}$$

$p \times t_p$ is the total time spent in all processes
t_1 is the time spent on useful work
t_0 is the time spent on overhead

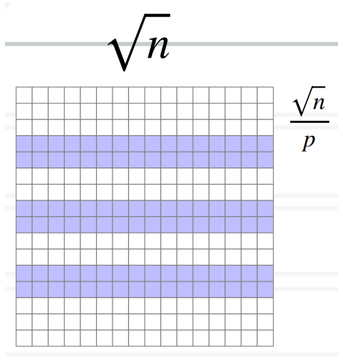
Thus, we know that efficiency is constant if $\frac{t_0}{t_1}$ is constant. We can call this K:

$$t_0 = K \times t_1$$

In general, we want to keep constant efficiency because that means that if we increase our problem size, we are growing our overhead work at the same rate as our useful work, such that we are not spending more time working on overhead work than you would have with a smaller problem size.

Example Isoefficiency Analysis

<u>1D Decomposition</u>



** image taken from lecture 4 slides, slide 15*

In a 1D decomposition we divide up the rows to different processes. So, given $\sqrt{n}$ elements per row and column, we know we will have $\frac{\sqrt{n}}{p}$ rows per process. We then need to calculate the two different computation times: useful work and overhead work. For this example, the only overhead work we will consider is the communication time, although in other examples there could be more considerations.

So, per process, the useful work consists of calculating the next value for each of the elements in the process' range. This means $\sqrt{n}$ elements per $\frac{\sqrt{n}}{p}$rows give us $\sqrt{n}\ \times \frac{\sqrt{n}}{p} = \frac{n}{p}$ of useful work.

For communication, we send 2 messages from each process to other processes, so per process, communication is 2 messages each of size $\sqrt{n}$ (the length of the row/size of the message), giving us $2 \times \sqrt{n}$ in total.

> *Note: For each process we are technically sending 2 messages and receiving 2 messages. This would imply that our communication overhead is actually 4 messages. However, we only count sending 2 messages because counting the received messages would double count the 2 messages sent by a different process.*

Going back to our isoefficiency function, we know we want to keep $\frac{t_0}{t_1}$ constant. Remember that t_0 is our overhead work and t_1 is our useful work. Thus, we get:

$$\frac{t_0}{t_1} = \frac{2 \times \sqrt{n}}{\frac{n}{p}} = \frac{2 \times p}{\sqrt{n}}$$

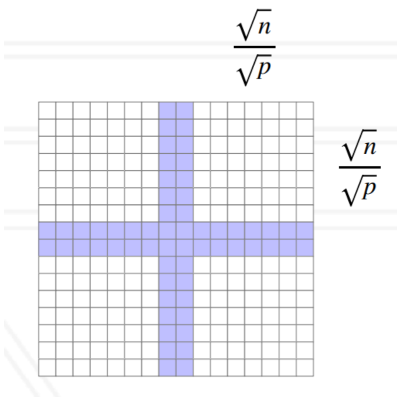
We can bring back K to be our constant and we get:

$$K = \frac{2 \times p}{\sqrt{n}}$$
$$2 \times p = K \times \sqrt{n}$$

This tells us that we have a quadratic relationship between problem size and number of processes (in order to keep K constant).

> *Note: To see the relationship we can think about if we have p=4 and n=64. What happens if we increase p to p=8? Then we need to multiply n by 4 to keep K constant.*

2D Decomposition



** image taken from lecture 4 slides, slide 15*

Calculating isoefficiency for 2D decomposition is very similar to 1D decomposition.
Useful work is simply the number of elements per process, which is one 'block' of the grid:

$$t_1 = \frac{\sqrt{n}}{\sqrt{p}} \times \frac{\sqrt{n}}{\sqrt{p}} = \frac{n}{p}$$

For communication overhead we need to send a row/column border of $\frac{\sqrt{n}}{\sqrt{p}}$ size to each of our 4 neighbors.

$$t_0 = 4 \times \frac{\sqrt{n}}{\sqrt{p}}$$

We can then look at $\frac{t_0}{t_1}$, and then K:

$$\frac{t_0}{t_1} = \frac{4 \times \frac{\sqrt{n}}{\sqrt{p}}}{\frac{n}{p}} = \frac{4 \times \sqrt{p}}{\sqrt{n}}$$
$$4 \times \sqrt{p} = K \times \sqrt{n}$$

Thus, we can see that the problem size and number of processes are linearly related (if we increase the problem size by 2, then we need to increase the number of processes by 2 to keep the efficiency constant.

Comparing 1D and 2D Decomposition Efficiency
As a general note, increasing problem size is not very straightforward for many applications, so it's much easier to scale at a smaller rate. This means that in these two examples, the 2D decomposition is much more scalable as compared to the 1D decomposition, as it only requires linear scaling rather than quadratic.

Empirical Performance Analysis

What we've done so far is essentially calculate the time complexities for parallel performance. Another way to analyze parallel performance is to simply measure the actual performance of a program and then analyze it. This is called empirical performance analysis.

In general, empirical performance analysis consists of measuring data and then analyzing/visualizing the performance. The easiest and most straightforward way to do this is with timing the code by putting manual timers in the code, such as with MPI_Wtime(). With this, we can time different parts of the code to figure out which parts run fastest/slowest, and which parts target first in increasing performance. This is also called manual instrumentation.

Performance Tools

There are 2 main categories of tools: tracing tools and profiling tools. Depending on the tool, they may only have tracing or profiling capabilities, but some of them can do both.

Tracing Tools

Tracing tools capture the entire execution trace and record every event in the program. This will generally make the overall execution time of the program a little slower, but the individual timestamps should be representative of its actual time. This is because tracking and saving the times in memory for each event adds a lot of overhead that slows down the program. Ideally, we want to try to reduce the amount of overhead so that our time tracking is as accurate to the non-tooled version of the program.

Tracing tools gather information through instrumentation: this means adding additional code to time parts of the code. This is essentially an automated way of manually adding timing code like described in the previous paragraphs. These tools can perform instrumentation through techniques like compiler pass or when compiling, intercepting every function and adding some timer code automatically.

Profiling Tools

Profiling tools capture an aggregate picture of the program, rather than a detailed execution trace. For example, if you call a function foo() 5 times in the code, then it will just sum up the times spent in foo() and give you the total time spent in foo() (it will not save the individual run timings of each call to foo(), just an overall aggregate statistic).

Some profiling tools use instrumentation to do this, and then aggregate at the end. However, many use sampling/statistical sampling. Sampling entails looking at the program counter (PC) and correlating the instruction that it points to a specific place in the code. It then figures out where in the code that you get a lot of samples. So, if it samples a certain PC many times, and that PC is correlated to a specific code place, then it knows that those code statements take longer than others or at least are generally invoked a lot more than others.

Note: Examples of both tracing and profiling tools can be found on the lecture 4's slides, starting from slide 20. Note that some tools, such as Gprof are not meant to be used on parallel programs, while others such as HPCToolkit, caliper, will work with parallel programs.

Metrics Recorded

We can measure a variety of metrics such as time recorded, function invocation counts, number of bytes sent, and hardware counters. Number of bytes sent (such as if you are sending messages via MPI) can help analyze communication, such as determining if you are sending too many messages or messages too big in size. Hardware counters can track many operations in hardware, such as special purpose registers that keep track of the # of floating-point operations, # of cache misses, # of branching instructions in the program, or # of memory (load/store) instructions. There are hundreds of different hardware counters which tools can query to get these metrics. Some tools will also output # of operations per function, rather than just for the whole programs' execution.

Calling Contexts, Trees, and Graphs

Calling contexts, or call paths, is just the path that we took to get to a part of the code, for example, a series of function invocations. If we keep track of this context, then we can also record how long a function takes to execute depending on the calling context.
Calling contexts can also be represented as calling context trees (CCT), which are dynamic prefix trees of all call paths in an execution or call graphs, which is essentially a CCT but there is only a single node per name/function (caller-callee relationships are kept).

Sophisticated tools can determine timings depending on contexts, and by using them we can determine which calling contexts/paths take the most time and which paths we might want to improve performance on. Just like the tracing and profiling tools, whether a calling context tree, call graph, or no calling context will be recorded (flat profile) depends on the tool used.

Tools can record the calling context by recording the call stack. Depending on the tool, they can return aggregate timings per function, or if they want to, they can then separate the timings of a function call per calling context. At each node we keep track of time spent in children (other function calls from there), and time spent outside of that. For example, if we do work in main(), then call solver(), then work in main() after that call, then the time spent in main will be recorded separately from the time spent in solver(), even though the function was invoked from main().

> *Note: We call the time spent in a function call, and all of its children the inclusive time, so this includes all the time in main() and solver(). On the other hand, exclusive time includes only the time spent in the function itself, excluding time spent in children.*

Timings are usually gathered at the leaf nodes, but at each node of the tree/graph times are usually recorded as well. In addition to aggregate timings, the times for each process spent at that function with that calling context can be held as well.

Q: Why use call graph vs. calling context tree knowing that the call graph has less data?
A: In general, what is generated is determined by the tools, but a call graph is likely used to save the memory space. Another reason is that it can be useful if we don't want very detailed info and we just want a general idea of which functions are taking longer time.

Hatchet
Hatchet is a tool that allows us to do programmatic analysis through writing scripts to analyze performance rather than through a GUI. It leverages pandas to help analyze calling context trees and represents the nodes of the graph as a structured table by using a structured index to index pandas DataFrames.

Pandas and DataFrames Intro
The main data structure in pandas is the DataFrame: a two-dimensional tabular data structure. You can store smaller structures inside like series. You can also have multidimensional indices, such as by using a tuple as the index, instead of just an integer. Overall, pandas supports many different operations on DataFrames that work just like common SQL operations and queries.

GraphFrame
GraphFrame is Hatchet's main data structure. It consists of a structured index graph object and a pandas DataFrame. For example, for the index, it can use the python object for each node. It can also have a multi-index of (node_name, mpi_rank) as the index to get a row per node per process.

Operations
A DataFrame operation that we can perform on Hatchet's GraphFrame is filter(). This simply entails passing in a function that takes in a row and returns a boolean value. It will then call that function on all the rows and filter out the rows that don't match the function (i.e. the function returns false).