Isoefficiency

Isoefficiency provides a good performance metric for determining scalability of a parallel program. It looks at the relationship between problem size and number of processes such that the efficiency remains constant. So, as we increase the number of processes, how much do we increase the problem size correspondingly to maintain the same efficiency.

In order to determine this relationship, let's look at the formula for efficiency. If we factor in overhead, total time in all processes ($p \times t_p$) is equal to useful computation (t_1) plus overhead (t_0), $p \times t_p = t_1 + t_0$.

The efficiency formula is $t_1 / (t_p \times p)$, substitute $t_p \times p$ with $t_1 + t_0$ as derived above and we get $t_1 / (t_1 + t_0) = 1 / (1 + (t_0 / t_1))$. By this formula, as long as t_0 / t_1 remains constant, our efficiency remains constant. So, we want t_0 / t_1 to be some constant K, since $t_0 / t_1 = K$, $t_0 = K \times t_1$.

Isoefficiency Analysis in Game of Life

Now let's apply this analysis to the Game of Life to determine which out of the 1D decomposition and 2D decomposition strategies scales better.

1D Decomposition

Suppose we have a square sqrt(n) by sqrt(n) board with a total of n cells. Starting with our 1D decomposition, our useful computation (t_1) consists of going through each cell and calculating its new state based on the neighbors. There are a total of sqrt(n) rows and we divide these rows among p processes, so the total number of rows each process has is sqrt(n) / p. We don't divide the columns at all, so each column is still of size sqrt(n). Thus, there are a total of (sqrt(n) / p) x sqrt(n) = n / p cells to perform useful computations on.

Now onto our overhead (t_0), this consists of the communication between the processes, i.e. the sending and receiving of rows. Each process performs two sends of one row each, and they are received correspondingly by the adjacent processes. Each row is of size sqrt(n), so in total, each process sends 2 x sqrt(n) cells.

With $t_1 = n / p$ and $t_0 = 2 \times \text{sqrt}(n)$, $t_1 = K \times t_0$ can be rewritten as $(n / p) = K \times 2 \times \text{sqrt}(n)$ and $\text{sqrt}(n) \times k = 2 \times p$, for $k = 1 / K$. So how should we grow our problem size to keep efficiency constant? As an example, suppose n = 64 and p = 4. If we increase our number of processes to 8, to keep k the same, we must multiply n by 4. As p increases by some factor c, n has to increase by some factor c^2. Thus we can see that our problem size varies quadratically with the number of processes.

2D Decomposition

For 2D decomposition, each process gets a block of sqrt(n) / sqrt(p) by sqrt(n) / sqrt(p). This means that there are (sqrt(n) / sqrt(p)) x (sqrt(n) / sqrt(p)) = n / p useful computations (t_1) to be done. Since each block has four boundaries, each process sends two rows and two columns to the processes holding adjacent blocks. Each row/column is of length sqrt(n) / sqrt(p), 4 x sqrt(n) / sqrt(p) cells to send (t_0).

With t_1 = n / p and t_0 = 4 x sqrt(n) / sqrt(p), t_1 = K x t_0 can be rewritten as (n / p) = K x 4 x sqrt(n) / sqrt(p) and k x sqrt(n) = 4 x sqrt(p) for some k = 1 / K. Here we can see that the relationship between p and n is linear, so increasing p by m units will also increase n by m units.

1D vs 2D Comparison

Which decomposition has better scalability? 2D decomposition does, because it has a smaller isoefficiency function n = k * p for some constant k, vs the 1D decomposition's isoefficiency function of n = k * p^2 for some constant k. (Isoefficiency: measuring the scalability of parallel algorithms and architectures (umd.edu)).

With every process we add, we need to increase the problem size by a certain amount, or else our overhead to useful computation ratio (t_0 / t_1) increases and the problem becomes less efficient. This is the relationship that the isoefficiency function tells us. So, a smaller isoefficiency function is more scalable because it tells us that smaller increases in the problem size are enough to efficiently use a larger number of processes, while a large isoefficiency function means that large increases in the problem size are required to efficiently use a larger number of processes.

Empirical Performance Analysis

Empirical performance analysis can also be utilized to examine what is happening in parallel programs. This consists of two main parts, measuring and analyzing data. Examples of the simplest tools that may be used here are timers in the code and print statements.

Performance Tools

Tools can get much more complex than just timers and print statements. Performance tools can be broken down into tracing tools and profiling tools.

Tracing tools

Tracing tools capture the entire execution trace, via instrumentation, which is additional code added in order to allow for traces. Tracing records all events with timestamps where events include functions, MPI calls, etc., providing an event timeline. Examples include VampirTrace, Score-P, Tau, HPCToolkit, etc.

Profiling tools

Profiling tools use statistical sampling to provide aggregated information about the program. Unlike tracing tools, profiling tools don't show the event timeline including individual invocations of a function, instead they only tell you the total time spent in a function. This is done by sampling the program counter which contains the address of the current instruction. This is correlated with the program code to determine how much time is being spent in what sections of the code. Examples include Gprof, perf, caliper, HPCToolkit, etc.

Tracing vs Profiling

Between tracing and profiling tools, profiling tools are easier to use and require significantly less overhead. Tracing tools have a higher performance impact and much higher overhead to keep track of all events.

Metrics Recorded

Some of the metrics that may be recorded by performance tools include: counts of function invocations, time spent in code, bytes sent, and hardware counters. Hardware counters can include tracking special registers for cache hits and cache misses, floating point operations, branching instructions, load/store instructions, etc. Once these metrics have been recorded, they can be connected to the source code to determine where performance problems may come from.

Output of Profiling Tools

Profiling tools can produce a number of different outputs. First is a flat profile which is simply just a listing of functions with their counts and execution times. They can also output a call graph profile or a calling context tree (CCT).

Calling Contexts, Trees, and Graphs

A calling context is a sequence of function invocations leading to the current sample. Essentially, the context is the stack trace of all the parent calls of a function. A calling context tree (CCT) is a dynamic prefix tree of all call paths in an execution. Finally, a call graph merges nodes in a CCT with the same name into a single node, but keeps caller-callee relationships. For example, in a CCT if a function main invokes functions foo and bar, main will have two

children, foo and bar. Suppose foo and bar each invoke a function baz, then both foo and bar will each have a distinct child baz. However, suppose the same function invocations were recorded in a call graph, then main would have two neighbors foo and bar, and foo and bar would both have links to one neighbor baz, which would be one node vs the two distinct baz nodes in the CCT.

Hatchet

Hatchet is a tool used for programmatic analysis after the profiling data from parallel programs have been collected. Programmatic analysis (in this case, scripts in Python) makes performance more flexible compared to using a GUI. Hatchet uses pandas which supports multidimensional tabular datasets. A structured index is created to enable indexing Pandas DataFrames by nodes in a graph along with a set of operators to filter, prune, and aggregate structure data.

What is a Pandas DataFrame? Pandas is an open source python library for data analysis. It uses data frames which are 2D tabular data structures. Pandas has the ability to create multi indices which enables working with high dimensional data in a 2D form.

Now we have the question of how do we convert the tree/graph in our profiling data to structured table data in Pandas? Hatchet's central data structure, a GraphFrame, addresses this question. GraphFrames consist of a structured index graph object and a Pandas DataFrame. The graph stores caller-callee relationships while the DataFrame stores all numerical and categorical data about the nodes. This preserves the graph relationships while porting the data into a tabular dataset.

An example of an operation that can now be performed on the dataset is filter. The DataFrame operation filter allows for filtering of the profiling data based on certain conditions. For example, we can filter the dataset to see which functions ran for longer than ten seconds.