**Summary of “Can Large Language Models Write Parallel Code?”**

**Background**

Large language models are rapid progressing as important tools in software development. Their ability to model and generate code has been demonstrated in a variety of applications such as code autocompletion, summarization, translation, and code search. Despite these achievements, these models often face challenges in generating code for more complex programs, such as in the generation of parallel code. Therefore, this study investigates the ability of large language models to generate parallel code. For evaluation purposes, this study proposed a new benchmark called ParEval, which consists of 420 different coding tasks across 12 computational problem types and 6 parallel programming paradigms. focusing on scientific and parallel computing challenges and it can evaluate the performance of leading open- and closed-source language models.

**What the author did in this paper**

This paper proposes a new type of benchmark for evaluating the capabilities of LLMs in high-performance computing (HPC), such as their ability to generate parallel code and perform parallel code translation.

**Why the author needed to do this and why it is important**

In modern software development, parallel code is essential but prone to errors. LLMs can help people overcome these challenges, but it's necessary to understand the current capabilities of LLMs. Therefore, the author believes that there is a need for a tool to evaluate these capabilities.

**What is the weakness of previous research**

There isn’t any focus on the generation of parallel code. Most previous research has concentrated on short tasks and array or string manipulations using Python. They primarily focus on correctness rather than the performance of the generated code, whereas performance is critical for writing parallel code.

**How the author proposed to overcome this problem**

The author proposed a new tool called the Parallel Code Generation Evaluation (ParEval) benchmark. It is a tool to evaluate the ability of LLMs to generate parallel code, containing 420 prompts that cover seven different execution models: serial, OpenMP, Kokkos, MPI, MPI+OpenMP, CUDA, and HIP. The benchmark also includes twelve problem types, such as sorting, scanning, dense linear algebra, sparse linear algebra, searching, reduction, histogram, stencil, graph, geometry, Fourier transform, and transformation. Additionally, small variations have been made to these typical problem types to prevent the prompts from being included in the training data. Since LLMs are given a code input prompt and asked to predict the next token, simply selecting the most probable next token can lead to repetitive and low-quality outputs. To mitigate this, the author utilized strategies for LLM token selection such as Nucleus Sampling (with p=0.95) and Model Temperature (with a low temperature value of 0.2). They also included the use of the #include statement in prompts, as it improves the likelihood of the LLM correctly using the required programming model. Metrics for correctness in this study included Pass@1, which aligns with how LLMs are used in practice. For performance, they used Speedupn@k, which measures the expected best performance speedup of the generated code relative to the performance of a sequential baseline given k attempts to generate the code. Additionally, for e

fficiency, they defined the Efficiencyn@k metric, which measures the expected best performance efficiency (speedup per process or thread) if the model is given k attempts to generate the code.

**The result of this research**

.

(I Still think about the weakness of this paper, or what possible for contain development of this topic)