Analytical Model for Parallel Programs

Arvind Krishnamurthy, Jeanna Neefe, and Randy Wang
{arvindk,neefe,rywang}@cs.berkeley.edu

1 Introduction

In this report we examine the relevance of the benchmark study of the Splash-2 suite to building shared memory parallel machines. A benchmark suite or its study could have several different claims to its importance. For example, the application benchmarks in the NAS suite represents a sample of the workload of the machines at NASA Ames Research Center. The NAS application benchmarks are indicative of the types of actual data movement and computation required in state-of-the-art fluid dynamics application codes. It therefore serves the purpose of benchmarking a new parallel machine based on the needs of the application programmers mainly working on aerospace problems. The stated purpose of the Splash-2 application study is not to propose a set of benchmarks that could be used as a litmus test for evaluating new machines. Instead, the Splash-2 study claims to quantitatively characterize the programs that belong to the suite in terms of certain machine-independent fundamental properties, which are key in providing insight into the architectural requirements of the programs. The study also claims to propose a methodology for studying application codes with the sole purpose of designing new machines. In this report, we examine whether the Splash-2 study is successful in its purported claims.

Our basic observations include that the Splash-2 study is successful in its working set analysis and does a very thorough analysis of the cache requirements for the Splash programs. However, it has some basic flaws in its analysis of load balance and communication properties of the programs. More specifically, it has erred in at least the following ways. Certain metrics used in the study such as the number of bytes communicated per instruction do not characterize the communication requirements of the programs. In certain ways, it is even misleading since two different programs with very different behavior/performance will share the same data point. At other places, the study mixes a number of independent parameters, which prevents us from determining the actual reason for some observed program behavior. As a result, the study falls short of providing a fail-proof methodology of examining parallel applications.

In sections 2 and 3, we discuss some of the shortcomings of the study. In section 4, we propose an alternate model/methodology for modeling the intrinsic properties of parallel programs.

2 Concurrency and Load Balance

The Splash-2 paper suggests that one of the first steps in studying an application is to analyze its available concurrency and load-balance. In particular, algorithms that do not have enough concurrency are not useful for studying the scalability characteristics of a parallel machine.

In this section of the paper, the Splash-2 developers provide two different kinds of numbers. The first characteristic they measure is the speedup of programs assuming an unit cycle memory access. The second characteristic that they measure is the amount of time spent in synchronization. Based on these numbers, they draw certain conclusions on the inherent concurrency and load-balance of the program. The approach definitely provides an upper-bound to the speedup that can be obtained for these programs, and therefore provides a truthful indicator of the available concurrency in the program. However, since load balance is more of a machine dependent parameter whose value depends on a lot of different factors, the numbers do not capture the potential load-balance of the applications. We now discuss the important deficiencies in the proposed methodology.

2.1 Inverse Amdahl's Law

The speedup curves for some of the applications deviate quite a bit from the linear speedup case. The reason given for such a behavior is that the data sets used for the simulation runs were small, and therefore the startup costs for these applications occupy a substantial portion of the running time. Also, the Splash-2 researchers claim that with bigger data sets the performance would scale better. An experiment that would support such a claim and also make the methodological process sound would be to separate the time spent in the sequential and parallel sections of the program.

The available concurrency could deviate from the ideal $PROCS$ fold parallelism either because of sequential work replicated across all processors, time spent in locks and mutually exclusive sections of the code, or due to lack of sufficient parallelism throughout the computation. By separating out the time spent in sequential section while presenting speedup characteristics of the application, we have a greater understanding of the parallel program. In particular, a program that has $0.8 * PROCS$ tasks in the parallel section could be differentiated from a program that spends 20% of the time in sequential code fragments with $PROCS$ fold parallelism in the parallel phase.

2.2 Effect of Single Cycle Memory Access Assumption

Load balance is a machine specific characteristic, and it is not necessarily a fundamental property of a program. In particular, it does not make sense to ignore the remote memory access costs while examining load balance.

Let us say that processor 1's instruction stream has an instruction mix of 50 million arithmetic operations and 50M remote memory references. Also, let processor 2's instruction stream have an instruction mix of 90M arithmetic operations and 10M remote memory accesses. The program would appear to be load balance with the single cycle memory access assumption. However, in any realistic parallel machine, there will be a considerable load imbalance.

In fact, the single cycle memory access assumption could result in other false characterizations of the program. The Splash-2 suite has an application called $Water$, which simulates the flow of water molecules in a stream. This application could be implemented in two different ways. We could assign a region of the stream to each processor and have the processor move water molecules from one processor to another when they move from one space to another. Another way of implementing the application would be not to move the molecules, but execute remote memory references to access the particles. With the single cycle memory assumption, both the implementations would have the same performance, whereas in reality one of the versions could be rather inefficient.

The bottom line is that load-balance cannot be expressed as a simple numerical value for a parallel program without referring to any of the hardware properties. A more meaningful approach is to provide a function (or formula) into which a remote access cost can be plugged into to provide the load balance of a program running on a particular machine.

2.3 Synchronization Costs

The Splash-2 report presents the time spent in synchronization in the different programs, and when programs have huge synchronization costs, their poor scalability (or efficiency) is attributed to the synchronization costs.

However, synchronization costs could be high due to two different reasons. Sometimes a program could have poor load-balance and therefore most of the processors might spend significant amount of time waiting at a $barrier$. Another scenario is that the program makes heavy use of $lock$s and $barrier$s in a fine-grained manner thus resulting in considerable overhead and poor speedups. In one of these cases, the high synchronization cost is a symptom of load imbalance, which is the actual reason for poor program performance. In the other case, the high synchronization cost results in poor program performance. While studying applications, we should separate one situation from the other.

3 Communication Behavior

In the section titled "Communication-to-Computation Ratios", the Splash-2 paper attempts to characterize the inherent communication behavior of the applications. It also claims that such characterization helps an architect of a parallel machine understand the bandwidth requirement of these applications.

First of all, we are amazed to see the Splash-2 paper attempts to characterize such an important aspect with a *single* parameter. Secondly, the authors have unfortunately chosen a particularly poor metric – traffic per instruction. As a result, the data fails to capture the intrinsic communication behavior of the programs. Programs that exhibit drastically different behavior and/or performance can share a single data point using their methodology. Furthermore, by being silent on the peak bandwidth requirements of the applications, this analysis can mislead a parallel machine designer.

The traffic per instruction is a poor metric for our purpose because it fails to capture the following important communication characteristics:

- communication granularity,
- burstiness of traffic,
- asynchrony of the communication primitives, and
- available overlap of communication and computation.

These characteristics have a profound impact on the bottom line performance but are made invisible by the umbrella metric "traffic per instruction". In the following subsections, we examine some of these issues in greater detail.

3.1 Communication Granularity

Communication granularity plays an important role in shaping the bottom line performance. They also stress different aspects of the underlying hardware. Consider the following code fragments:

```
A () {                              B () {
                                        repeat 1M times {
    read 1 MB;                              read 1 B;
    compute on this 1 MB;                   compute on this 1 B;
                                        }
}                                   }
```

Function A stresses bandwidth while function B is sensitive to latency. A high per message overhead is likely to cost B dearly while A has the potential of amortizing it. Yet these two functions have identical traffic to instruction ratio. As an example, table 1 shows the obvious performance benefit of the bulk operation.

Fundamentally, what this metric has hidden is the ability to overlap communication with communication. It also hides the peak bandwidth demand under the facade of an average. A more meaningful measure should break this average down into a histogram so we have a more precise idea about how much communication is required before each computation.

The issue of communication granularity becomes even more relevant when we start to explore different coherence models. For example, the SAM??? object layer provides coherence at the granularity of objects

Memory Op.	CM5	T3D
Word	1.0	1.0
Bulk	0.21	0.34

Table 1: Normalized running times of word and bulk transfers.

instead of cache lines. Programs can be manipulating objects at drastically different granularities with drastically different costs, and yet share the same communication per instruction metric.

3.2 Overlapping Communication and Computation

Asynchronous communication primitives also allow one to overlap the costs of communication and computation. Consider the following code fragments:

```
A () {
    repeat {
        read for this iteration;
        compute for this iteration;
    }
}
```

```
B () {
    repeat {
        read for next iteration;
        compute for this iteration;
    }
}
```

The communication in function A must be synchronous while it can be asynchronous in function B to allow the overlap of communication and computation. Although the behavior and performance of these two programs are likely to be quite different, the traffic per instruction metric, unfortunately, again fails to provide any insight. We wrote a few toy kernels that have the same amount of communication per instruction but employ different types of memory operations. We run these kernels on the CM5 and the T3D, and we noticed, not surprisingly, drastic improvements in the running time for the kernels with asynchronous memory operations. Table 2 shows one of the sample results.

Memory Op.	CM5	T3D
Read	1.0	1.0
Get	0.48	0.37

Table 2: Normalized running times of synchronous and asynchronous kernels.

Such differences in communication behavior can have important implications for an architect. For example, program IO will probably do just fine for function A but function B can potentially benefit significantly from DMA. Furthermore, depending on how sophisticated the "read" is, one might even want a second network processor to handle the traffic. Then there is the question of how powerful this processor needs to be. Needless to say, a traffic per instruction number hardly sheds any light for such questions.

To better model such communication characteristics, in addition to a message size distribution, we propose yet another histogram that shows how much computation can be performed before we demand the result of the previously initiated communication.

3.3 Traffic Burstiness

Another aspect of the communication behavior that a single traffic per instruction metric does not capture is burstiness. Consider the following code fragments:

```
A () {                                B () {
    repeat 1M times {
        write 1 B;                        repeat 1M times {
    }                                         write 1 B;
    repeat 1M times {                         compute;
        compute;                          }
    }
}                                     }
```

The burstiness issue illustrated here is related to the issues discussed above but different. It's not a granularity issue because in both programs we are doing small writes. It's not an asynchrony issue either because the writes can be asynchronous in both programs. Yet we expect the performance of these two programs to differ under certain circumstances.

Function A is attempting to inject all of its messages into the network at once. When the network is congested, A will be forced to wait. Function B, on the other hand, spreads its communication over a longer period of time and experiences less congestion.

So who wins? Interestingly enough, the answer depends on what metric we use. If we use peak bandwidth, A is likely to beat B. If we use traffic per instruction, it's a draw. And if we use average bandwidth or running time, B is likely to win!

To further illustrate the idea, we consider the "bandwidth matching" technique by Brewer. In order not to overrun a slow receiver, the sender deliberately limits its injection rate. This is an example of an application's effort of smoothing burstiness to achieve better throughput. As shown in figure 1, the performance gain can be quite dramatic.

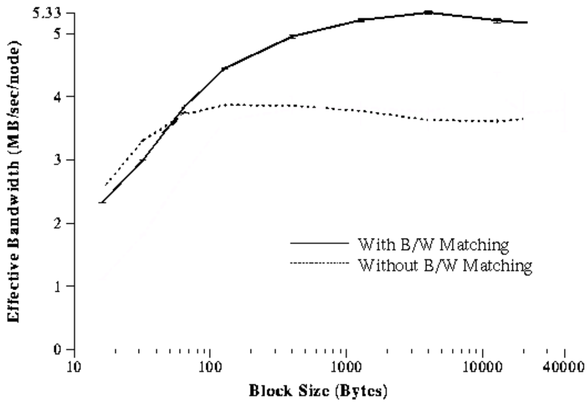

Figure 1:

Again, we have seen that the simplistic Splash-2 analysis has failed to capture an application's communication behavior. Our proposed solution to this problem is similar to the one discussed in subsection 3.1. We need to characterize the amount of communication between computations.

3.4 Summary

Communication behavior is one of the most important aspects of parallel applications. The Splash-2 paper, unfortunately, chooses to characterize such an important issue with a single umbrella parameter called "traffic per instruction". We have demonstrated in this section how inadequate this methodology is. In particular, we have shown that issues such as communication granularity, the ability of overlapping communication with computation, and traffic burstiness can result in drastically different program behavior and performance. To address the inadequacy of the Splash-2 effort, we propose mainly two sets of histograms for modeling communication behavior. One measures the amount of communication before each instance of computation; and the second measures the amount of computation that can still make progress after each instance of communication.

4 Analytical Model for Performance

The goal of developing an analytical model is to predict the performance of an application given a new machine specified by some hardware parameters. Since the advent of RISC, a very accurate model based on instruction mix, costs of instructions, and cache hit rates has been developed to predict program performance. Our goal is to develop a performance model for parallel programs that would be as comprehensive as the model proposed by Hennessey and Patterson for RISC processor performance. However, at this stage, we have not been able to develop a model that characterizes aspect of a parallel machine. In particular, our model is deficient in capturing contention issues that result in poor program performance. The two common scenarios that are not captured in our model are bandwidth limitation and lock-based serialization. In both cases, the program runs slower due to interference between the activities of different processors, which could be either non-deterministic or at least timing specific.

We would like our model to capture the load balance characteristics of a program running on a new machine as well as predict the performance penalties incurred due to communication. We, therefore, would like to measure the following aspects by running the program through a simulator.

- Measure distance (in terms of instructions) between an issue of a remote memory reference and its subsequent usage. Organize these numbers in the form of an histogram. When we actually plug in the machine parameters for remote memory latency, we would use charge different costs based on this distance. For example, if a value gets used after k cycles and if an access costs l cycles, the cost of the instruction would be $l - k$ cycles if $k < l$ or 1 cycle if $k > l$.
- Measure instruction mix of each parallel thread in every phase of the program (between every pair of *barrier* synchronizations). The hardware model for the machine would provide the scaling parameters for these instruction counts (just like the CPI calculation for RISC processors). By combining the parameters of the hardware model with the inherent behavior of the program, we can calculate the execution time of the program on a new machine. Also, since we do not make any assumptions about instruction latencies in this model, we can also calculate the load balance of a program running on a specific machine, which addresses the issues raised in section 2.