Relentless Parallelism

Nels E. Beckman
Institute for Software Research
School of Computer Science
Carnegie Mellon University
nbeckman@cs.cmu.edu

ABSTRACT

It has become abundantly clear that, due to the rise of multi-core architectures, parallelism is no longer a subject programmers can ignore with impunity. Unfortunately, programming concurrent code is hard. I mean seriously. Some algorithms can not be parallelized, and more importantly, some people cannot be bothered learning new programming constructs. Toward returning to a state of programmer ignorance, we present *Relentless Parallelism*, a programming methodology that promises full utilization of all CPUs and cores without additional programmer effort. We explain our system though an example and formal rewriting rules.

1. INTRODUCTION

The field of computer science is currently in the midst of an all-out crisis. Moore's Law, first formalized in 1965 continues to hold. The number of transistors that can be placed on a process doubles approximately every two years. However, we have reached the limit of general-purpose performance for single CPU systems. Limiting factors, for example heat, have made it increasingly difficult to utilize all those new transistors in a single processor. Instead, ICU manufacturers have begun to develop *multi-core* CPUs, processors that internally contain multiple distinct processors. Currently multi-core CPUs are shipping with two and four cores, but the near future expects to see dozens and even hundreds of cores per chip. Ladies and gentlemen, *the age of parallelism is upon us!*

Unfortunately, the eminent scholars agree: *Concurrency is Really, Really Freaking Hard* [1]. Developing applications that can actually take advantage of many cores is poised to be the next great challenge of computer science. In this paper we propose a programming methodology, christened, Relentless Parallelism, that provides a solution to this looming problem. Relentless Parallelism promises to keep each core in a machine busy, even when developing algorithms for which no natural parallel encoding exists.

This paper proceeds as follows: In Section 2 we explain relentless parallelism by way of example of a traditionally hard-to-parallelize algorithm, Huffman decoding. In Section 3 we formalize this approach using a series of rewriting rules. Finally, Section 4 concludes.

2. EXAMPLE: HUFFMAN DECODING

Some algorithms, for example, branch-and-bound search or optimization, lend themselves naturally to parallel decomposition. Other problems, unfortunately do not. These problems are particularly worrisome, since they will not be able to benefit from the coming influx of CPU codes.

```
String huffmanDecodeByte(Queue<Byte> byte_stream,
                         DecTreeNode cur_node) {
  if( cur_node.getValue() != null ) {
    // We are at a leaf node
    return cur_node.getValue();
  }
  else {
    if( byte_stream.remove().byteValue() == 0) {
      // Go to the left
      return
      huffmanDecodeByte(byte_stream,
                        cur_node.getLeftNode());
    }
    else {
      // Go to the right
      return
      huffmanDecodeByte(byte_stream,
                        cur_node.getRightNode());
    }
  }
}
```

Figure 1: Huffman decoding: Because character codes have variable lengths, a naive implementation is difficult to parallelize, for example, using divide-and-conquer.

Figure 2 is an example of one such algorithm, Huffman decoding. Huffman coding is a prefix-free coding scheme, often used in compression applications. In the scheme, characters are assigned variable length codes based upon their probability of appearance. Since probabilities are allowed to change from case to case, a tree mapping codes to characters is necessary for decoding. The natural way of decoding a series of bits is to proceed left or right down the mapping tree (depending on the current bit). When a leaf is reached, that leaf necessarily specifies exactly one character, since the scheme is prefix-free.

Unfortunately, because the length of codings is variable, parallelizing this implementation is not straightforward. The normal divide-and-conquer approach fails. If we were to divide the bit stream into multiple sections to give to multiple

cores, a seemingly natural fit, we would be unable to tell a-priori which size chunks to give to each processor, since one cannot tell which bits denote the start or end of a character until decoding has been performed.

Relentless Parallelism assures full utilization of each core even for algorithms that are not naturally parallelize. Our technique consists of a series of rewriting rules which add parallelism to otherwise sequential algorithms. Figure 2 shows the result of this transformation when applied to the Huffman decoding algorithm. Note that while Figure 2 shows the body of the `huffmanDecodeByte`, the result of the transformation can only be seen at the top level of the program.

```
String huffmanDecode(Queue<Byte> byte_stream,
                     DecTreeNode tree) {

  class Parallelizer extends Thread {
    public void run() {
      for(int i=1, acc=1;
          i<this.hashCode();i++,acc*=1 ){}
      this.run();
  }};
  int procs=
    Runtime.getRuntime().availableProcessors();
  for(int i=0;i<procs-1;i++) {
    (new Parallelizer()).start();
  }

  StringBuffer result = new StringBuffer("");
  while( !byte_stream.isEmpty() ) {
    result.append(
      huffmanDecodeByte(byte_stream, tree));
  }
  return result.toString();
}
```

Figure 2: The result of the Relentless Parallelism transform. Note how the `Parallelizer` class produces maximum CPU utilization.

The result of the transform is that previously un-utilized CPUs are now maximally utilized. The performance improvement is characterized as follows:

$$\text{Utilization}_0 = \frac{1}{|\text{CPUs}|}$$

$$\text{Utilization}_{rp} = \frac{|\text{CPUs}|}{|\text{CPUs}|}$$

3. FORMAL DESCRIPTION

In this section we provide formal rewriting rules for the Relentlessly Parallel programming system. These rules are described in Figure 3.

While the majority of the rules are relatively straight-forward, we would like to draw special attention to the ASYNCH rule. We would expect that our natural notion of parallelism would validate certain rules. One of them is that channels can not affect the computation of processes that do not use them. This rule shows that our notion of parallelism is correct.

$$\frac{x \text{ and } g \text{ do not alias}}{[x] := 1 || [g] := 2} \text{ CONCURRENT UPDATE}$$

$$\frac{}{(f, g) : W \to X} \text{ MORPHISM}$$

$$\frac{[\![(\pi)]\!] W \overset{[\![p]\!]^W}{\to} [\![Q]\!] X}{[\![(\pi)]\!] X \overset{[\![p]\!]^X}{\to} [\![Q]\!] X}$$

$$\frac{}{\pi \vdash P : Q,\ [\![P]\!] : [\![(\pi)]\!] \dot{\to} [\![Q]\!]} \text{ WORLDS}$$

$$\frac{(h!0) \setminus h = \delta \quad \text{when } h \notin \text{P's channel}}{\texttt{local } h \texttt{ in } (h!0; P) = P} \text{ ASYNCH}$$

$$\frac{P(S \times S)}{P((S \times S)^\infty)} \text{ POM}$$

Figure 3: The formal rewriting rules for the Relentless Parallelism system.

4. CONCLUSION

The future of programming is an uncertain one. The rise of multi-core architectures potentially will have vast and far-reaching consequences. A large majority of programmers are not familiar or experienced writing parallel code. Moreover, some algorithms are not easily parallelized, even by experienced coders. Yes it is a scary future. However, in this paper we have presented a programming methodology, Relentless Parallelism, that will help to remove much uncertainty from the future. Our methodology, which we have formalized with a series of rewriting rules, will allow even sequential programs to achieve maximum CPU utilization for all cores and processors.

4.1 Implementation

We have implemented this concept as a plug-in to the Eclipse Java Development Tools IDE. This plug-in and source code are available for download at the following address:

`http://www.nelsbeckman.com/software.html`

While the plug-in itself only works on Java code, rest assured that the monumental contributions we have made are applicable to any modern programming language and Fortran 77 [2].

5. REFERENCES

[1] Beckman, Nels E. *Concurrency is Really, Really Freaking Hard.* In Proceedings of SIGBOVIK: Workshop About Symposium on Robot Dance Party of Conference in Celebration of Harry Q. Bovik's 0x40th Birthday. Pittsburgh, PA, USA-A-OK. April 6, 2008.

[2] FORTRAN 77 4.0 Reference Manual. SunSoft `http://www.physics.ucdavis.edu/~vem/F77_Ref.pdf`