Evaluating the Productivity of a Multicore Architecture

Jeremy Kepner and Nadya Bliss {kepner,nt}@ll.mit.edu
MIT Lincoln Laboratory, Lexington, MA 02420

Abstract

Increased performance generally comes at the price of increased programmer effort. However, some computer architectures require less effort than others to achieve the same level of performance. Evaluating this trade off between performance and effort (i.e., productivity) for different parallel multicore processors is important for processor architects and users. This work presents one approach for evaluating the productivity of a multicore architecture based on the expected performance and programming difficulty of various programming models ranging from high level serial code (e.g., C++) to low level parallel code (e.g. assembly and DMA). This analysis indicates that small elements of a multicore architecture can have a large impact on the shape of the performance vs. effort curve for that architecture.

Introduction

The tradeoff between performance and programmer effort is a fundamental characteristic of High Performance Embedded Computing (HPEC) systems. This tradeoff encompasses three orders of magnitude in effort and five orders magnitude in performance (see Table 1).

Table 1: Approximate effort vs. performance. Applications can be implemented with a variety of interfaces with a clear tradeoff between effort and performance.

Implementation environment	Relative effort	Relative performance
Spreadsheet	1/30	1/100
Matlab, IDL, …	1/10	1/5
C++	1/3	1/1.1
VSIPL, BLAS	2/3	1/1.05
C, Fortran	1	1
Assembly	3	2
VHDL	10	10
Standard cell	30	100
Custom VLSI	100	1000

Within the context of parallel multicore architectures there are similar tradeoffs that exist between performance and effort. Examining this tradeoff is one way to evaluate the productivity of a multicore architecture. Such an assessment is valuable for both multicore processor architects and users. Architects seek to design processors that deliver high performance with the least effort. Likewise, users seek processors that will deliver the required performance within the required level of effort.

Parallel Programming Models

The principal programming challenge of a multicore architecture is effectively utilizing the parallel capabilities made available by the architecture. A baseline for assessing multicore productivity is conventional serial programming. Theoretically, the highest productivity architecture would be one that would allow a serial program to achieve high performance via simple recompilation by a parallel compiler. Unfortunately, such an ideal architecture/compiler technology is not currently available. At the other extreme is the expert "human compiler" who rewrites the serial program using low level machine specific assembly and DMA (direct memory access) calls.

Table 2: Parallel programming approaches. Estimated relative codes, fraction of programmers and difficulty of various parallel programming approaches

Parallel approach	Relative code size	Fraction of programmers	"Difficulty"
Serial	1	1.00	1
Multithread	1.1	0.95	1.15
Dist. Arrays	1.5	0.50	3
Hier. Arrays	2	0.10	20
DMA	10	0.05	200

In between these two extremes are the more conventional parallel programming techniques. The first of these techniques is the thread based approach (e.g., OpenMP, pthreads, or Cilk), that allows the programmer to quickly implement parallel concurrency, but provides less support for managing parallel data locality. Next, come distributed array based approaches (e.g., POOMA, VSIPL++, or GA) that require the programmer to address data locality and

This work is sponsored by the Department of the Air Force under Air Force contract FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government.

then derive concurrency from that locality). Lastly, hierarchical distributed array approaches have started to emerge (e.g., pMatlabXVM or PVTOL) that deal with data locality across the memory hierarchy. Estimates of the relative effort and expertise to use these different technologies are shown in Table 2. Expertise is quoted in terms of the estimated fraction of programmers who can effectively use the technology. Dividing effort by expertise results in a quantity labeled "difficulty".

Multicore Architecture Assessment

One of the biggest questions in multitcore architecture is whether or not to use homogeneous or heterogeneous cores. Consider two canonical multicore architectures. The first architecture is a replicated homogeneous RISC processing core (e.g., an x86 or MIPS core) with a cache on each core. The second architecture is a replicated RISC/SIMD heterogeneous processing core (e.g., a Cell or a x86/GPU hybrid) with a cache for the RISC part and DMA into a local storage for the SIMD unit(s).

To evaluate the architectures consider two applications. The first is a SAR (Synthetic Aperture Radar) application which relies on large 2D FFTs. The second application is a video processing application that relies on rotating large images. The applications are then "written" in each of the different parallel programming approaches and the performance is assessed on both architectures. Figure 1 shows the output of this analysis. There are a number of details that emerge from this analysis. In general, homogeneous systems would appear to give higher performance at modest programmer difficulty, while heterogeneous systems give higher performance at higher programmer difficulty.

The next step is to go beyond these baseline systems and explore architectures that potentially give better performance with less programmer difficulty.

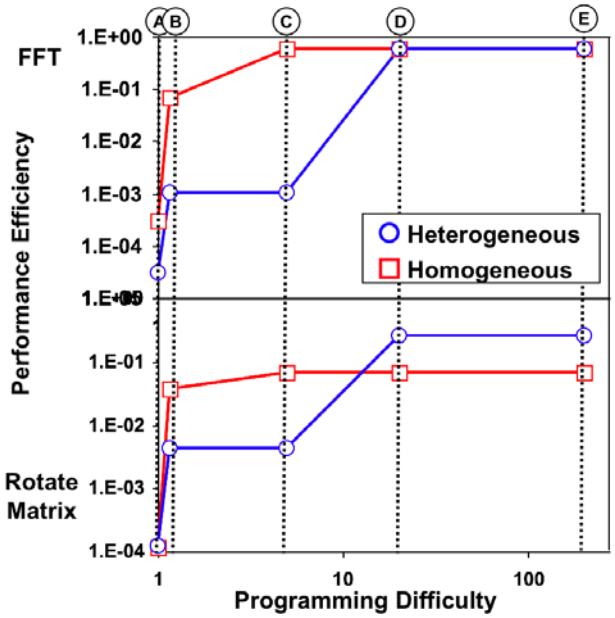

Figure 1: Estimated performance vs difficulty. (A) = Serial, (B) = Multithreaded, (C) = Distributed arrays, (D) = Hiearchical arrays, (E) = Assmebly+DMA.

References

Cilk: supertech.csail.mit.edu/cilk/

GA: www.emsl.pnl.gov/docs/global/

OpenMP: www.openmp.org/

pMatlabXVM: www.ll.mit.edu/HPEC/agendas/proc05/Day_2/Presentations/1000_Kim.pdf

POOMA: acts.nersc.gov/pooma/

PVTOL: www.mit.edu/~kepner/PVTOL/

VSIPL++: www.codesourcery.com/vsiplplusplus