Energy Saving using Parallel Computing

Gopal Menon
Computer Science Department
Utah State University
Logan, Utah 84322
Email: gopal.menon@aggiemail.usu.edu

I. Introduction

A. Data Center Energy Consumption

Data centers are the backbone of the modern economy [1], from the server rooms that power small to medium-sized organizations, to the enterprise data centers that support american corporations, to the server farms that run cloud computing services hosted by amazon, Facebook, Google, and others. However, the explosion of digital content, big data, e-commerce, and Internet traffic is also making data centers one of the fastest-growing users of electricity in developed countries, and one of the key drivers in the construction of new power plants in the united states.

While most media and public attention focuses on the largest data centers that power so-called cloud computing operations - companies that provide web-based and other Internet services to consumers and businesses?these hyper-scale cloud computing data centers represent only a small fraction of data center energy consumption in the united states.

In 2013, U.S. data centers consumed an estimated 91 billion kilowatt-hours of electricity. This is the equivalent annual output of 34 large (500-megawatt) coal-fired power plants, enough electricity to power all the households in New York City twice over. Data center electricity consumption is projected to increase to roughly 140 billion kilowatt-hours annually by 2020, the equivalent annual output of 50 power plants, costing american businesses $13 billion per year in electricity bills and causing the emission of nearly 150 million metric tons of carbon pollution annually.

The data center energy usage is illustrated in figure 1. The use of social media, email, tweets, song and movie downloads has resulted in growing demands on data centers. The internet traffic generated by this usage is shown in figure 2.

TABLE I
ESTIMATED U.S. DATA CENTER ELECTRICITY CONSUMPTION BY MARKET SEGMENT (2011) [1]

Segment	Servers (million)	Share	kWh/y
Small and Medium server rooms	4.9	49%	37.5
Enterprise/Corporate Data Centers	3.7	27%	20.5
Multi-Tenant Data Centers	2.7	19%	14.1
Hyper-scale Cloud Computing	0.9	4%	3.3
High-Performance Computing	0.1	1%	1.0
Total (rounded)	12.2	100%	76.4

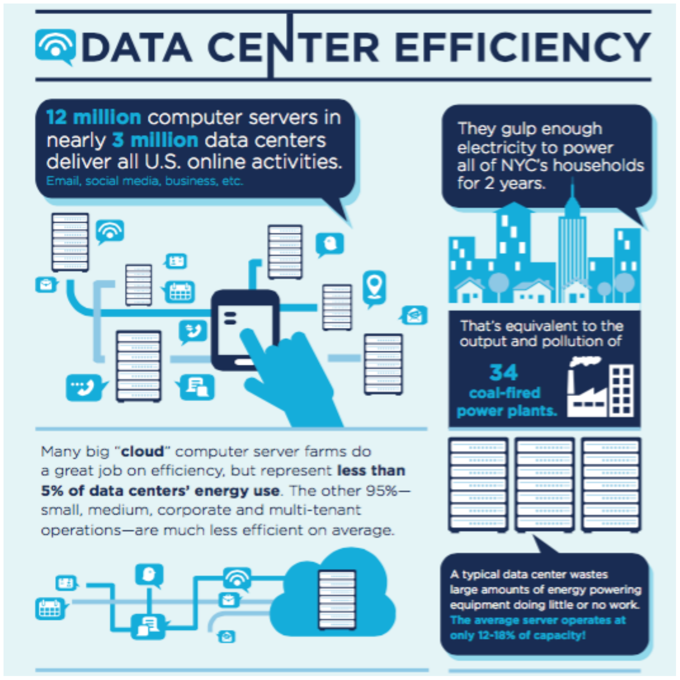
DATA CENTER EFFICIENCY
12 million computer servers in nearly 3 million data centers deliver all U.S. online activities.
Email, social media, business, etc.
They gulp enough electricity to power all of NYC's households for 2 years.
That's equivalent to the output and pollution of 34 coal-fired power plants.
Many big "cloud" computer server farms do a great job on efficiency, but represent less than 5% of data centers' energy use. The other 95%—small, medium, corporate and multi-tenant operations—are much less efficient on average.
A typical data center wastes large amounts of energy powering equipment doing little or no work. The average server operates at only 12-18% of capacity!

Fig. 1. Data Center Efficiency [1]

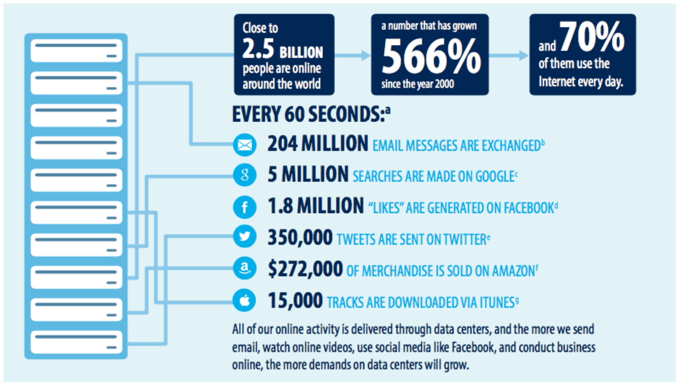
Close to 2.5 BILLION people are online around the world
a number that has grown 566% since the year 2000
and 70% of them use the Internet every day.
EVERY 60 SECONDS:a
204 MILLION EMAIL MESSAGES ARE EXCHANGEDb
5 MILLION SEARCHES ARE MADE ON GOOGLEc
1.8 MILLION "LIKES" ARE GENERATED ON FACEBOOKd
350,000 TWEETS ARE SENT ON TWITTERe
$272,000 OF MERCHANDISE IS SOLD ON AMAZONf
15,000 TRACKS ARE DOWNLOADED VIA ITUNESg
All of our online activity is delivered through data centers, and the more we send email, watch online videos, use social media like Facebook, and conduct business online, the more demands on data centers will grow.

Fig. 2. Data Center Usage [1]

The estimated energy consumption by market segment for the year 2011 is shown in table I. As can be seen, the big cloud computing companies are very energy efficient and consume less that 5% of the total energy consumed. 95% of the energy is consumed by small, medium, corporate and multi-tenant

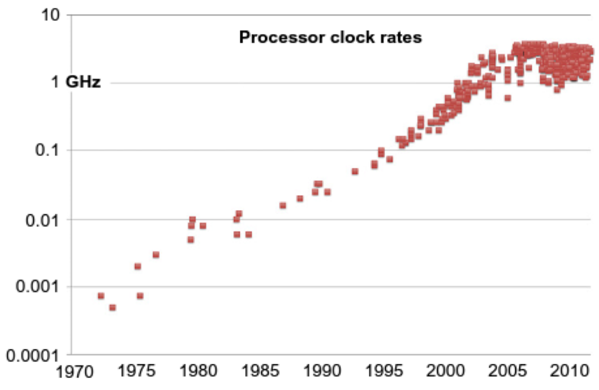

Fig. 3. Growth of processor clock rates over time (log scale). This graph shows a dramatic halt by 2005 due to the power wall, although current processors are available over a diverse range of clock frequencies. [4]

operations.

B. Problem Statement

Until 2004, along with the increase in the number of transistors on a chip as per Moore's law, there was also a rise in switching speed of the transistors [4]. This translated into improved performance of microprocessors through a steady rise in their clock speeds.

From 1973 to 2003, clock rates increased by three orders of magnitude, from about 1 MHz in 1973 to 1 GHz in 2003. However, as is clear from figure 3 clock rates have now ceased to grow and are now generally in the 3 GHz range. In 2005, three factors converged to limit the growth in performance of single cores and shift new processor designs to the use of multiple cores [4]. These are known as the “three walls”:

1) **Power wall:** Unacceptable growth in power usage with clock rate.
2) **Instruction-level parallelism (ILP) wall:** Limits to available low-level parallelism.
3) **Memory wall:** A growing discrepancy of processor speeds relative to memory speeds.

The power wall results because power consumption (and heat generation) increases non-linearly as the clock rate increases. Increasing clock rates any further will exceed the power density that can be dealt with by air cooling, and also results in power-inefficient computation.

The second wall is the instruction-level parallelism (ILP) wall. Many programmers would like parallelization to somehow be done automatically. The fact is that automatic parallelization is already being done at the instruction level, and has been done for decades, but has reached its limits. Hardware is naturally parallel, and modern processors typically include a large amount of circuitry to extract available parallelism from serial instruction streams. LP can only deliver constant factors of speedup and cannot deliver continuously scaling performance over time.

Programming has long been done primarily as if computers were serial machines. Meanwhile, computer architects (and compiler writers) worked diligently to find ways to automatically extract parallelism, via ILP, from their code. For 40 years, it was possible to maintain this illusion of a serial programming model and write reasonably efficient programs while largely ignoring the true parallel nature of hardware. However, the point of decreasing returns has been passed with ILP techniques, and most computer architects believe that these techniques have reached their limit. The ILP wall reflects the fact that the automatically extractable low-level parallelism has already been used up.

The memory wall results because off-chip memory rates have not grown as fast as on-chip computation rates. This is due to several factors, including power and the number of pins that can be easily incorporated into an integrated package. Despite recent advances, such as double-data-rate (DDR) signaling, off-chip communication is still relatively slow and power-hungry. Many of the transistors used in today's processors are for cache, a form of on-chip memory that can help with this problem. However, the performance of many applications is fundamentally bounded by memory performance, not compute performance. Many programmers have been able to ignore this due to the effectiveness of large caches for serial processors. However, for parallel processors, interprocessor communication is also bounded by the memory wall, and this can severely limit scalability. Actually, there are two problems with memory (and communication): latency and bandwidth . Bandwidth (overall data rate) can still be scaled in several ways, such as optical interconnections, but latency (the time between when a request is submitted and when it is satisfied) is subject to fundamental limits, such as the speed of light. Fortunately latency can be hidden?given sufficient additional parallelism, above and beyond that required to satisfy multiple computational units. So the memory wall has two effects: Algorithms need to be structured to avoid memory access and communication as much as possible, and fundamental limits on latency create even more requirements for parallelism.

In summary, in order to achieve increasing performance over time for each new processor generation, you cannot depend on rising clock rates, due to the power wall. You also cannot depend on automatic mechanisms to find (more) parallelism in na?ve serial code, due to the ILP wall. To achieve higher performance, you now have to write explicitly parallel programs. And finally, when you write these parallel programs, the memory wall means that you also have to seriously consider communication and memory access costs and may even have to use additional parallelism to hide latency. Instead of using the growing number of transistors predicted by Moore's Law for ways to maintain the “serial processor illusion”, architects of modern processor designs now provide multiple mechanisms for explicit parallelism. However, you must use them, and use them well, in order to

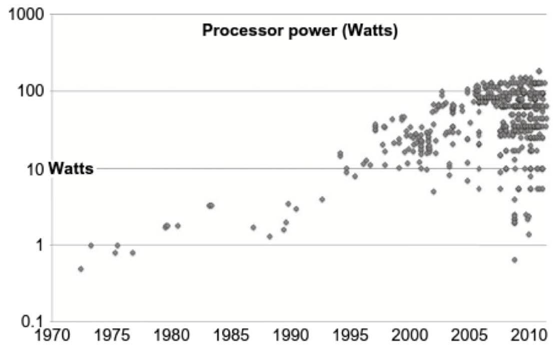

Fig. 4. Graph of processor total power consumption (log scale). The maximum power consumption of processors saw steady growth for nearly two decades before the multicore era. The inability to dissipate heat with air cooling not only brought this growth to a halt but increased interest in reduced power consumption, greater efficiencies, and mobile operation created more options at lower power as well. [4]

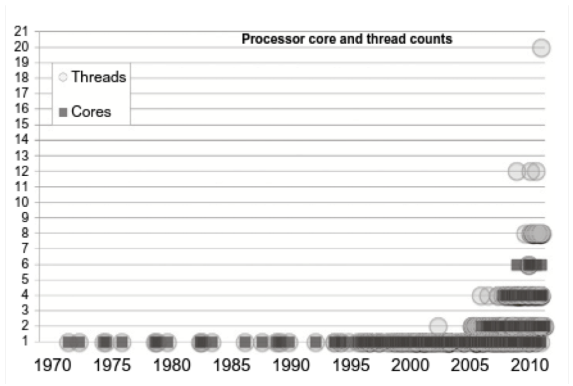

Fig. 5. The number of cores and hardware threads per processor was one until around 2004, when growth in hardware threads emerged as the trend instead of growth in clock rate. [4]

achieve performance that will continue to scale over time. The resulting trend in hardware is clear: More and more parallelism at a hardware level will become available for any application that is written to utilize it. However, unlike rising clock rates, non-parallelized application performance will not change without active changes in programming. The “free lunch” of automatically faster serial applications through faster microprocessors has ended. The new “free lunch” requires scalable parallel programming. The good news is that if you design a program for scalable parallelism, it will continue to scale as processors with more parallelism become available.

Figure 4 shows a graph of total power consumption over time. After decades of steady increase in power consumption, the so-called power wall was hit about 2004. Above around 130W, air cooling is no longer practical. Arresting power growth required that clock rates stop climbing. From this chart we can see that modern processors now span a large range of power consumption, with the availability of lower power parts driven by the growth of mobile and embedded computing.

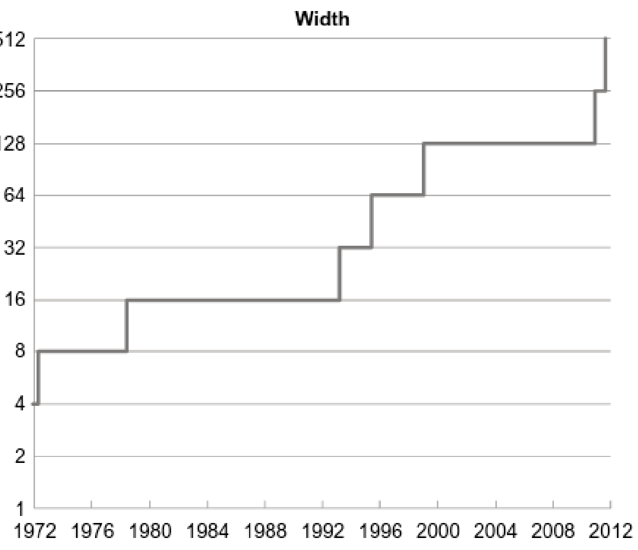

Fig. 6. Growth in data processing widths (log scale), measured as the number of bits in registers over time. At first the width of scalar elements grew, but now the number of elements in a register is growing with the addition of vector (SIMD) instructions that can specify the processing of multiple scalar elements at once. [4]

The resulting trend toward explicit parallelism mechanisms is obvious looking at figure 5, which plots the sudden rise in the number of hardware threads after 2004. It is common to refer to hardware parallelism as processor cores and to stress multicore. But it is more precise to speak of hardware threads, since some cores can execute more than one thread at a time. Both are shown in the graph.This date aligns with the halt in the growth in clock rate. The power problem was arrested by adding more cores and more threads in each core rather than increasing the clock rate. This ushered in the multicore era, but using multiple hardware threads requires more software changes than prior changes. During this time vector instructions were added as well, and these provide an additional, multiplicative form of explicit parallelism. Vector parallelism can be seen as an extension of data width parallelism, since both are related to the width of hardware registers and the amount of data that can be processed with a single instruction. A measure of the growth of data width parallelism is shown in figure 6. While data width parallelism growth predates the halt in the growth of clock rates, the forces driving multicore parallelism growth are also adding motivation to increase data width. While some automatic parallelization (including vectorization) is possible, it has not been universally successful. Explicit parallel programming is generally needed to fully exploit these two forms of hardware parallelism capabilities.

Additional hardware parallelism will continue to be motivated by Moore's Law coupled with power constraints. This will lead to processor designs that are increasingly complex and diverse. Proper abstraction of parallel programming methods is necessary to be able to deal

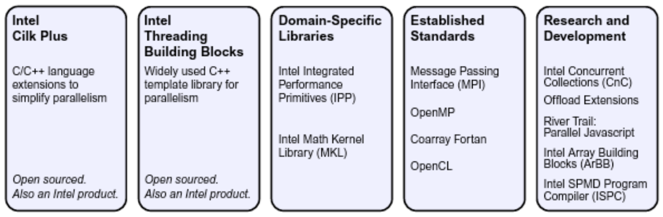
Intel Cilk Plus
C/C++ language extensions to simplify parallelism
Open sourced. Also an Intel product.
Intel Threading Building Blocks
Widely used C++ template library for parallelism
Open sourced. Also an Intel product.
Domain-Specific Libraries
Intel Integrated Performance Primitives (IPP)
Intel Math Kernel Library (MKL)
Established Standards
Message Passing Interface (MPI)
OpenMP
Coarray Fortan
OpenCL
Research and Development
Intel Concurrent Collections (CnC)
Offload Extensions
River Trail: Parallel Javascript
Intel Array Building Blocks (ArBB)
Intel SPMD Program Compiler (ISPC)

Fig. 7. Parallel programming models supported by Intel. A choice of approaches is available, including pre-optimized parallel libraries; standards such as MPI, Coarray Fortran, OpenMP, and OpenCL; dynamic data-parallel virtual machines such as ArBB; domain-specific languages targeting SPMD vector parallelism such as ISPC; coordination languages such as CnC; and Cilk Plus and TBB. [4]

with this diversity and to deal with the fact that Moore's Law continues unabated, so the maximum number of cores (and the diversity of processors) will continue to increase.

C. Parallel Programming Models

Unfortunately, none of the most popular programming languages in use today was designed for parallel programming. However, since a large amount of code has already been written in existing serial languages, practically speaking it is necessary to find an evolutionary path that extends existing programming practices and tools to support parallelism. Broadly speaking, while enabling dependable results, parallel programming models should have the following properties:

- **Performance:** Achievable, scalable, predictable, and tunable. It should be possible to predictably achieve good performance and to scale that performance to larger systems.
- **Productivity:** Expressive, composable, debuggable, and maintainable. Programming models should be complete and it should be possible to directly and clearly express efficient implementations for a suitable range of algorithms. Observability and predictability should make it possible to debug and maintain programs.
- **Portability:** Functionality and performance, across operating systems and compilers. Parallel programming models should work on a range of targets, now and into the future.

Figure 7 shows the parallel programming models supported by Intel.

D. Speedup and Amdahl's Law

Two important metrics related to performance and parallelism are **speedup** and **efficiency**. Speedup compares the latency for solving the identical computational problem on one hardware unit (“worker”) versus on P hardware units:

$$speedup = S_P = \frac{T_1}{T_P} \tag{1}$$

where T_1 is the latency of the program with one worker and T_P is the latency on P workers.

Efficiency is speedup divided by the number of workers:

$$efficiency = \frac{S_P}{P} = \frac{T_1}{PT_P} \tag{2}$$

Efficiency measures return on hardware investment. Ideal efficiency is 1 (often reported as 100%), which corresponds to a linear speedup, but many factors can reduce efficiency below this ideal.

An algorithm that runs P times faster on P processors is said to exhibit linear speedup . Linear speedup is rare in practice, since there is extra work involved in distributing work to processors and coordinating them. In addition, an optimal serial algorithm may be able to do less work overall than an optimal parallel algorithm for certain problems, so the achievable speedup may be sublinear in P , even on theoretical ideal machines. Linear speedup is usually considered optimal since we can serialize the parallel algorithm, as noted above, and run it on a serial machine with a linear slowdown as a worst-case baseline.

Amdahl said that the execution time T_1 of a program falls into two categories:

- Time spent doing non-parallelizable serialwork
- Time spent doing parallelizable work

Call these W_{ser} and W_{par}, respectively. Given P workers available to do the parallelizable work, the times for sequential execution and parallel execution are:

$$T_1 = W_{ser} + W_{par} \tag{3}$$

$$T_P \geq W_{ser} + \frac{W_{par}}{P} \tag{4}$$

Substituting these equations into the definition of speedup in equation 1 gives us:

$$S_P \leq \frac{W_{ser} + W_{par}}{W_{ser} + \frac{W_{par}}{P}} \tag{5}$$

Figure 8 shows this bound on speedup. Let f be the non-parallelizable serial fraction of the total work. Then the following equalities hold:

$$W_{ser} = fT_1 \tag{6}$$

$$W_{par} = (1 - f)T_1 \tag{7}$$

If we substitute these into equation 5, we get

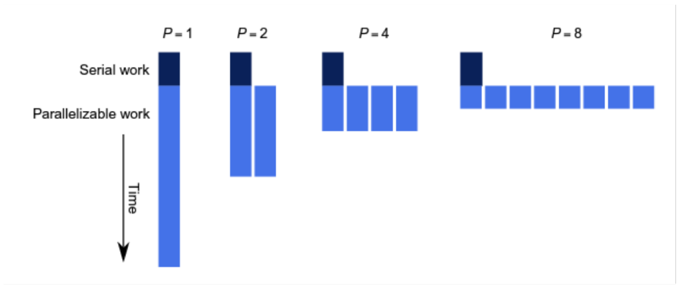

Fig. 8. Amdahl?s Law. Speedup is limited by the non-parallelizable serial portion of the work. [4]

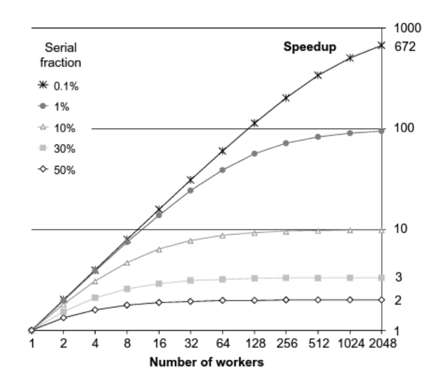

Fig. 9. Amdahl?s Law: speedup. The scalability of parallelization is limited by the non-parallelizable (serial) portion of the workload. The serial fraction is the percentage of code that is not parallelized. [4]

$$S_P \leq \frac{1}{f + \frac{(1-f)}{P}} \tag{8}$$

When the number of processors is infinite, the speedup becomes

$$S_\infty \leq \frac{1}{f} \tag{9}$$

Speedup is limited by the fraction of the work that is not parallelizable, even using an infinite number of processors. If 10% of the application cannot be parallelized, then the maximum speedup is 10. If 1% of the application cannot be parallelized, then the maximum speedup is 100. In practice, an infinite number of processors is not available. With fewer processors, the speedup may be reduced, which gives an upper bound on the speedup. Amdahl's Law is graphed in figure 9, which shows the bound for various values of f and P. For example, observe that even with $f = 0.001$ (that is, only 0.1% of the application is serial) and $P = 2048$, a program's speedup is limited to 672.

E. Opportunity

Parallelization can reduce power consumption. CMOS is the dominant circuit technology for current computer hardware. CMOS power consumption is the sum of dynamic power consumption and static power consumption [4]. For a circuit supply voltage V and operating frequency f, CMOS dynamic power dissipation is governed by the proportion

$$P_{dynamic} \propto V^2 f \tag{10}$$

The frequency dependence is actually more severe than the equation suggests, because the highest frequency at which a CMOS circuit can operate is roughly proportional to the voltage. Thus dynamic power varies as the cube of the maximum frequency. Static power consumption is nominally independent of frequency but is dependent on voltage. The relation is more complex than for dynamic power, but, for sake of argument, assume it varies cubically with voltage. Since the necessary voltage is proportional to the maximum frequency, the static power consumption varies as the cube of the maximum frequency, too. Under this assumption we can use a simple overall model where the total power consumption varies by the cube of the frequency.

Suppose that parallelization speeds up an application by 1.5 on two cores. You can use this speedup either to reduce latency or reduce power. If your latency requirement is already met, then reducing the clock rate of the cores by 1.5 will save a significant amount of power. Let P_1 be the power consumed by one core running the serial version of the application. Then the power consumed by two cores running the parallel version of the application will be given by:

$$\begin{aligned} P_2 &= 2(\frac{1}{1.5})^3 P_1 \\ &\approx 0.6 P_1 \end{aligned} \tag{11}$$

where the factor of 2 arises from having two cores. Using two cores running the parallelized version of the application at the lower clock rate has the same latency but uses (in this case) 40% less power. Unfortunately, reality is not so simple. Current chips have so many transistors that frequency and voltage are already scaled down to near the lower limit just to avoid overheating, so there is not much leeway for raising the frequency. For example, Intel Turbo Boost Technology enables cores to be put to sleep so that the power can be devoted to the remaining cores while keeping the chip within its thermal design power limits. Table II shows an example. Still, the table shows that even low parallel efficiencies offer more performance on this chip than serial execution.

Another way to save power is to “race to sleep”. In this strategy, we try to get the computation done as fast as possible (with the lowest latency) so that all the cores can be put in a sleep state that draws very little power. This approach is attractive if a significant fraction of the wakeful power is

TABLE II
RUNNING FEWER CORES FASTER. THE TABLE SHOWS HOW THE MAXIMUM CORE FREQUENCY FOR AN INTEL CORE I5-2500T CHIP DEPENDS ON THE NUMBER OF ACTIVE CORES. THE LAST COLUMN SHOWS THE PARALLEL EFFICIENCY OVER ALL FOUR CORES REQUIRED TO MATCH THE SPEED OF USING ONLY ONE ACTIVE CORE. [4]

Acive Cores	Maximum Frequency (GHz)	Breakeven Efficiency
4	2.4	34%
3	2.8	39%
2	3.2	52%
1	3.3	100%

fixed regardless of how many cores are running. Especially in mobile devices, parallelism can be used to reduce latency. This reduces the time the device, including its display and other components, is powered up. This not only improves the user experience but also reduces the overall power consumption for performing a user's task: a win-win.

F. Contributions

II. MOTIVATION AND BACKGROUND

A. Motivation

B. Technologies Used

The Threading Building Blocks (TBB) programming model has been selected for the purpose of this project. TBB supports parallelism based on a tasking model. It provides the following features:

- Template library supporting both regular and irregular parallelism
- Direct support for a variety of parallel patterns, including map, fork-join, task graphs, reduction, scan, and pipelines
- Efficient work-stealing load balancing
- A collection of thread-safe data structures
- Efficient low-level primitives for atomic operations and memory allocation

TBB is a library, not a language extension, and thus can be used with with any compiler supporting ISO C++. Because of that, TBB uses C++ features to implement its ?syntax.? TBB requires the use of function objects (also known as functors) to specify blocks of code to run in parallel. These were somewhat tedious to specify in C++98. However, the C++11 addition of lambda expressions greatly simplifies specifying these blocks of code, so that is the style used in this book. TBB relies on templates and generic programming. Generic programming means that algorithms are written with the fewest possible assumptions about data structures, which maximizes potential for reuse. The C++ Standard Template Library (STL) is a good example of generic programming in which the interfaces are specified only by requirements on template types and work across a broad range of types that meet those requirements. TBB follows a similar philosophy. Like Cilk Plus, TBB is based on programming in terms of tasks, not threads. This allows it to reduce overhead and to more efficiently manage resources. As with Cilk Plus, TBB implements a common thread pool shared by all tasks and balances load via work-stealing. Use of this model allows for nested parallelism while avoiding the problem of over-subscription. The TBB implementation generally avoids global locks in its implementation. In particular, there is no global task queue and the memory allocator is lock free. This allows for much more scalability. As discussed later, global locks effectively serialize programs that could otherwise run in parallel. Individual components of TBB may also be used with other parallel programming models. It is common to see the TBB parallel memory allocator used with Cilk Plus or OpenMP programs, for example.

III. EXPERIMENTS

A. Subjects

B. Experimental Setup

C. Experimental Results

D. Threats to Validity

IV. DISCUSSION

A. Results Analysis

V. RELATED WORK

VI. CONCLUSION/SUMMARY

REFERENCES

[1] Whitney, J., and P. Delforge. *Data Center Efficiency Assessment?Scaling Up Energy Efficiency Across the Data Center Industry: Evaluating Key Drivers and Barriers.* NRDC and Anthesis, Rep. IP (2014): 14-08.

[2] Woo, Dong Hyuk, and Hsien-Hsin S. Lee. *Extending Amdahl's law for energy-efficient computing in the many-core era.* Computer 12 (2008): 24-31.

[3] Ribic, Haris, and Yu David Liu. *Energy-efficient work-stealing language runtimes.* ACM SIGARCH Computer Architecture News. Vol. 42. No. 1. ACM, 2014.

[4] McCool, M. J. (2012). *Structured Parallel Programming: Patterns for Efficient Computation.* Amsterdam: Elsevier, Morgan Kaufman.

[5] Nanz, S. S. (2013). *Examining the expert gap in parallel programming.* Euro-Par 2013 Parallel Processing. , 434-445.

[6] Wilson, G. V. (1995). *Assessing and comparing the usability of parallel programming systems.* Computer Systems Research Institute.

[7] Cebrin, Juan M., Lasse Natvig, and Jan Christian Meyer. *Performance and energy impact of parallelization and vectorization techniques in modern microprocessors.* Computing 96.12 (2014): 1179-1193.