Extending Amdahl's Law for the Cloud Computing Era

Fernando Díaz-del-Río, Javier Salmerón-García, and José Luis Sevillano, University of Seville

By extending Amdahl's law, software developers can weigh the pros and cons of moving their applications to the cloud.

According to Gartner's hype cycle,[1] cloud computing is now in a productive phase. Thus, it can be exploited at several levels: infrastructure as a service, which, like Amazon Elastic Compute Cloud (EC2), provides customers with customizable virtual machines; platform as a service, which offers a framework, such as Google App Engine or Windows Azure, that customers can use to develop their own applications; and software as a service, which gives customers access to specific applications only, such as Microsoft Office Web or Dropbox.

Cloud computing has considerable advantages over computing on local devices: it offers automatic scaling; there is no need to purchase, upgrade, or maintain bare-metal hardware; and it saves energy resources for mobile devices.[2] Cloud computing has also opened up research opportunities in areas such as new programming paradigms, mobile agent software, security and privacy, and tool balancing and deployment. However, for apps with real-time constraints, one might question whether remote execution is faster than local execution. Hence, both energy consumption and performance are crucial for cloud computing systems.

Moving some apps to the cloud yields extraordinary results, whereas offloading others is out of the question. Consider, for instance, apps that search huge information databases stored in servers. Whether these apps involve simple Internet searches for a term or feature-based image searches (for example, Google Goggles), they must sort through a vast number of possible matches. Hence, the offloading overhead time is much less than the time to obtain search results. These apps occur on the server side for two obvious reasons: the extensive volume of information to be processed, and the high amount of computation required. If the user information to be processed is already stored in the cloud, no data transfer is required (only a pointer to the data),[5] making cloud computing preferable. But what about those apps that capture information online from the local device? Could they benefit by cloud offloading today? What about tomorrow?

To help software developers weigh the pros and cons of cloud offloading, we offer a simple extension of Gene

Amdahl's law, which is used to predict the maximum speedup of a system due to improvements in part of the system. This extension also allows us to consider what apps are most suitable for offloading, as well peer into cloud computing's future.

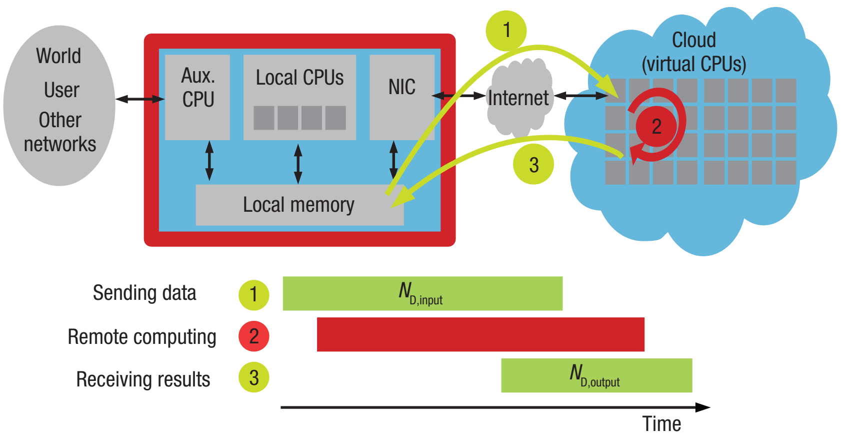

FIGURE 1. Centralized application execution (top left) versus offloaded app execution (top right and bottom). $N_{\text{D,input}}$ and $N_{\text{D,output}}$ together represent the total bits of data exchanged with the cloud. NIC: network interface controller.

AMDAHL'S LAW AND THE CLOUD

Although originally intended for uniprocessor versus multiprocessor execution time comparisons, Amdahl's famous law can be applied to any system. Let speedup S be the original execution time divided by an enhanced execution time.[3] If a fraction F of the original time is enhanced by a speedup S_{fraction}, the overall speedup is

$$S = \frac{1}{(1-F)+F/S_{\text{fraction}}}.$$

Note that Amdahl assumed the extreme case: fraction F was infinitely parallelizable (no overhead times were included), and the remaining fraction, 1 – F, was totally sequential.

To extend Amdahl's law to cloud computing, we must compare centralized versus offloaded application performance. Figure 1 shows the main system architectural components involved in both execution models.

Centralized application execution

Classical centralized architectures consist of a CPU that captures code instructions and operand data from its local hierarchic memory, executes them, and stores the results in memory. Each core executes instructions sequentially according to the von Neumann model. Consequently, uniprocessor execution time can be expressed as $N_I \times CPI \times T$, where N_I is the number of program instructions, CPI is the mean number of cycles per instruction, and T is the clock period (see the "Components of Execution Time" sidebar).

Nowadays, CPUs contain $N_{\text{c,local}}$ number of cores. According to Amdahl's law, the $N_{\text{c,local}}$ cores execute a fraction F of the program in parallel, while a single core executes the rest—namely, $F \times N_I$ instructions are executed in parallel, but $(1 - F)N_I$ instructions are not. For simplicity's sake, suppose that a CPU is a symmetric multicore chip (that is, all its cores are identical, which is the most common type) and that interaction with the outer world (the input/output subsystem) is irrelevant to execution time; only the Internet connection, or network interface controller (NIC), plays a significant role. Given these assumptions, local program execution time is

$$t_{\text{local}} = (1-F)N_I CPI_{\text{local}} T_{\text{local}} + \frac{FN_I}{N_{\text{c,local}}} CPI_{\text{local}} T_{\text{local}}$$
$$= \left[(1-F) + \frac{F}{N_{\text{c,local}}}\right] N_I CPI_{\text{local}} T_{\text{local}}.$$

Offloaded application execution

If an app is offloaded to the cloud, the local device first sends its data and code through the Internet (Figure 1: green arrow, step 1) to the cloud server. Next, the transferred app is executed on the cloud. Finally, the server returns the results to the local device (green arrow, step 3). The bottom of Figure 1 schematizes the time involved in steps 1–3, where $N_{\text{Data}} = N_{\text{D,input}} + N_{\text{D,output}}$, or the total amount in bits of data exchanged with the cloud.

We assume some additional simplifications to calculate cloud execution time:

- The program's code size can be neglected: it is either much smaller than the data size (evident when images, videos, big data, and so on are processed) or already resides, for the most part, in the cloud (for example, libraries).
- Data is transferred at a constant communication bandwidth (BW), while its startup latency is negligible (or done in parallel with transmissions).
- Internal cloud overhead times are not considered, because most occur in parallel with other times.

COMPONENTS OF EXECUTION TIME

Most computers are finite state machines with a CPU in which almost everything is synchronous with a clock of period *T*. A program's execution time t_{exec} is a multiple of N_{clocks} periods; that is, $t_{exec} = N_{clocks}T$. Likewise, the vast majority of modern computers use the von Neumann model, established 60 years ago, in which a program consists of a number N_I of instructions executed in sequential order. Sequential execution is not an efficient computation method but is how most people express solutions to problems; hence, most computing languages stick to this representation.

N_{clocks} can be split as

$$N_{clocks} = N_I \frac{N_{clocks}}{N_I} = N_I CPI$$

and, therefore, $t_{exec} = N_I \times CPI \times T$. This is the fundamental formula of computer architects, and the three factors N_I, *CPI*, and *T* each play a role in microprocessor design.

CPI (clocks per instruction) is the CPU's ability to execute many instructions per clock. In the early stages of computing, *CPI* was relatively high (several clocks per instruction). However, the advent of reduced-instruction-set machines in 1985 allowed architects to design efficient CPU pipelines that executed several instructions in just one clock cycle ($CPI < 1$). Thus, execution times could be reduced progressively as a result of *CPI* diminution and period contraction. By the end of the last century, architectural innovations had plateaued. Today, ideal *CPI* is given by the inverse of the width of the processor issue stage (around 5 instructions per cycle for the past 15 years; hence, ideally $CPI \approx 1/5$). Nevertheless, CPU stall cycles make real-world $CPI > 1$ for a representative set of benchmarks. To make matters worse, around 2005, the CPU period also came to a standstill. These constraints led to the so-called "multicore era."

Indeed, communication and computation times might overlap. The two extremes are described below.

If no overlap exists, cloud execution time is the sum of communication and computation times:

$$t_{cloud} = \frac{N_{Data}}{BW} + \left[(1-F) + \frac{F}{N_{c,cloud}}\right] N_I CPI_{cloud} T_{cloud}$$
$$= \frac{N_{Data}}{BW} + (1-F) N_I CPI_{cloud} T_{cloud}.$$

Because cloud resources can be scaled up dynamically, the number of virtual CPUs should be large enough that $F/N_{c,cloud} \to 0$. Conversely, if the overlap is complete, communication times are completely hidden; hence,

$$t_{cloud,overlapping} = (1-F) N_I CPI_{cloud} T_{cloud}.$$

Comparing cloud and local performance

For simplicity, assume that the local and cloud CPU technologies are similar; that is, $CPI_{cloud} \approx CPI_{local}$, and $T_{cloud} \approx T_{local}$. In fact, because the cloud contains cutting-edge technology, its cores might well be faster than local cores. Hence, this simplification favors local machines.

With complete communication–computation overlap, local execution time will exceed cloud execution time because communication penalties are negligible when cloud resources are far bigger than local ones. In a worst-case cloud scenario, where there is no overlap, computing S_t as t_{local}/t_{cloud}, dividing the numerator and denominator by $N_I \times CPI \times T$, and reordering yields the following speedup:

$$S_t(F, N_{c,local}, \mu, D_I) = \frac{t_{local}}{t_{cloud}} = \frac{\frac{F}{N_{c,local}} + (1-F)}{\frac{\mu}{D_I} + (1-F)}.$$

Thus, we define two parameters. The first, $\mu = (CPI \times T \times BW)^{-1}$, establishes the ratio between the local machine's capacity to execute instructions per second and core $(CPI \times T)^{-1}$ and its capacity to send data bits per second (*BW*). That is, μ is reciprocal to the machine's offloading capabilities. The second parameter, $D_I = N_I/N_{Data}$, represents the app's computing density—the mean number of instructions that must be executed for any data bit to be exchanged with the cloud.

Figure 2a shows the speedup S_t as a function of D_I for different *F*, supposing a modest local device ($N_{c,local} = 1$). Note

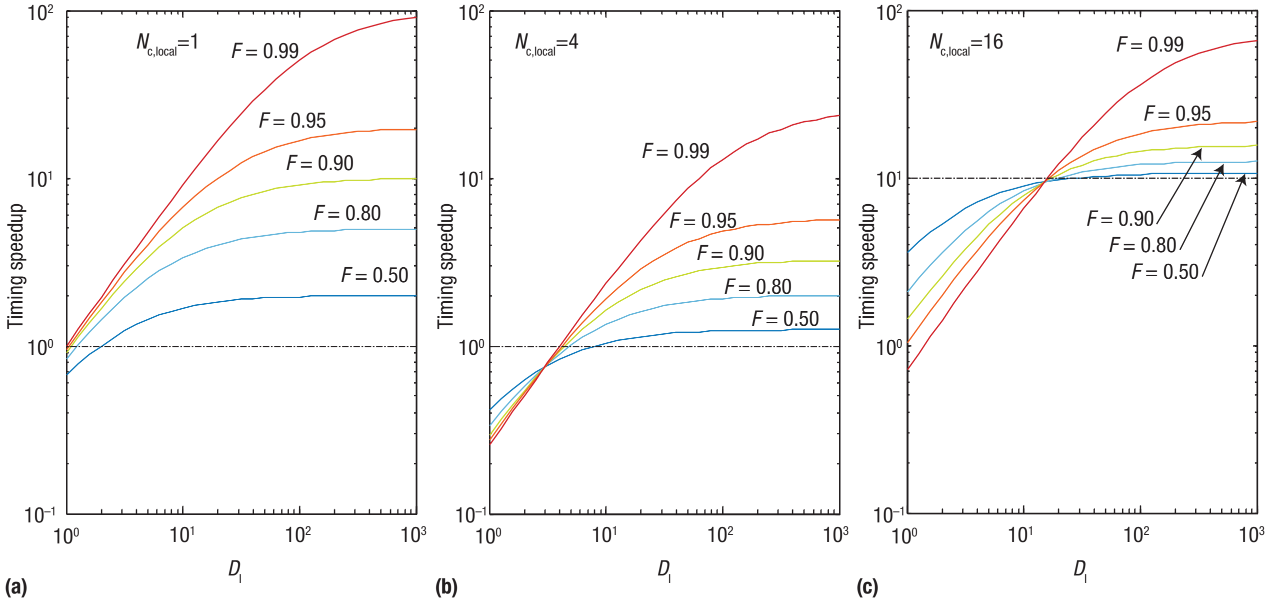

FIGURE 2. Speedup for remote versus local execution ($N_{c,local}$) as a function of $D_I = N_I/N_{Data}$ (app computing density) for different fraction F, with $\mu = (CPI \times T \times BW)^{-1} = 1$: (a) $N_{c,local} = 1$, (b) $N_{c,local} = 4$, and (c) $N_{c,local} = 16$.

that the scales are logarithmic, because S_t rises to high values for a large D_I. For high F values, remote execution is clearly advantageous when D_I is moderate. But for $F < 0.5$, offloading is faster even for low D_I values (>2). Thus, for simple devices, cloud computing can benefit a huge range of apps. The situation is less favorable when the local device is more powerful ($N_{c,local} = 4$; see Figure 2b). In this case, a new effect appears: for moderate values of $D_I > 8$, offloading is advantageous for any $F \geq 0.5$. A similar effect occurs with an extremely powerful device ($N_{c,local} = 16$; see Figure 2c): if $D_I > 20$, the high-energy-consuming local device would not be beneficial.

Our comparison of local and cloud performance yields three important results.

First, total cloud execution time depends strongly on the amount of overlap between communication and computation times. This overlap should be thoroughly analyzed by the middleware that manages the task offloading. (The "Stereo Vision Offloading for Mobile Robots" sidebar describes an application case.)

Second, μ/D_I plays a critical role in speedup. Currently, μ is estimated to be a few units (*CPI* rounds up to 1 for most programs, and *T* and *BW* are on the order of 1 ns and 1 Gbps, respectively). However, μ is expected to progressively decrease because *BW* will presumably continue increasing at a geometric rate, and *CPI* × *T* has reached a fixed value that will be difficult to surpass with present technology (see the "Components of Execution Time" sidebar). This implies that because μ/D_I is expected to decrease, the benefits of using the cloud will increase in the near future.

Finally, when using uniprocessors in the local device ($N_{c,local} = 1$),

$$S_t = \frac{1}{\frac{\mu}{D_I} + (1 - F)}.$$

Thus, offloading execution would be convenient for apps with low D_I (even if $F < 0.5$). Therefore, if μ/D_I decreases in the future, simplified device hardware might become the new trend. This would have the additional benefits of saving energy and reducing software complexity. Note that in Internet of Things or bare-metal thin client devices, local CPUs would not even exist (see Figure 1).

STEREO VISION OFFLOADING FOR MOBILE ROBOTS

Mobile robot technology has reached an elevated degree of maturity in the past decade. For example, fully autonomous cars could soon become commercially available. This has been motivated in part by advances in sensory information processing such as stereo vision. Such advancements were due mainly to the advent of more powerful parallel architectures and the availability of distributed OSs, such as robotics software frameworks (RSFs). RSFs have allowed improved scalability, reusability, deployment, and debugging[1] and the deployment of tasks either on board or in cloud computing systems, depending on their constraints.[2]

Stereo vision is an active research field. A higher degree of time complexity is involved in analyzing the difference between stereo frame pairs, that is, the difference between what the left and right eyes see. Disparity maps permit calculation of the objects' distance to the cameras, more or less accurately depending on the algorithm used and the image features. Most representative algorithms have time complexities ranging from $O(N^2)$ to $O(N^{11})$, where N is the horizontal image size.[3] OpenCV (http://opencv.org) has one of the most extended algorithms with complexity on the order of $O(N^3)$—the usual level for most algorithms. The problem is that real-time requirements can be difficult to meet when more accurate reconstruction of the environment is demanded. The frequency of stereo frame processing must be sufficiently high but its latency sufficiently small. For example, when the robot is moving, the distance to the nearest obstacle must be computed early enough to avoid a crash or an emergency stop. Such a task must be processed by a powerful computing system or—as is more common nowadays—performed with low-resolution images.

Thus, the possibility of cloud-based stereo vision is gaining momentum. The offloading process must not only be parallel, but must also exploit the cloud's dynamic scalability. Concretely, if multiple CPU cores were available, processing times could run in parallel to transfer times, which are often the bottleneck during cloud offloading.[2] This could yield mean processing frequencies nearly proportional to network bandwidth.

References

1. P. Iñigo-Blasco et al., "Robotics Software Frameworks for Multi-agent Robotic Systems Development," *Robotics and Autonomous Systems*, vol. 60, no. 6, 2012, pp. 803–821.
2. J. Salmerón-García et al., "A Trade-Off Analysis of a Cloud-Based Robot Navigation Assistant Using Stereo Image Processing," *IEEE Trans. Automation Science and Eng.*, vol. 12, no. 2, 2015, pp. 444–454.
3. M. Sizintsev and R.P. Wildes, "Coarse-to-Fine Stereo Vision with Accurate 3D Boundaries," *Image and Vision Computing*, vol. 28, no. 3, 2010, pp. 352–366.

CONSIDERING PERFORMANCE AND ENERGY

Performance speedup is useful for any system, but mobile devices' battery lifetime is a serious constraint when executing computation-demanding apps.[4] With existing technology, most of the energy consumed when offloading an app to the cloud is due to data transmission. Therefore, the question becomes, can extending Amdahl's law for the cloud predict whether the energy that offloading saves compensate for the energy that local processing consumes?[5]

Energy consumption is the sum of power multiplied by the time of the different periods:

$$\sum_i P_i \times t_i .$$

According to Dong Hyuk Woo and Hsien-Hsin Lee's model, local app execution consists of a parallel period and a sequential period.[6] During the fraction F of parallel execution time, all the cores are involved, so power is $N_{c,\text{local}}P_1$, with P_1 being the power of a single core. During the sequential period $(1 - F)$, the power is $k_{\text{idle}}P_1(N_{c,\text{local}} - 1) + P_1$, because one core is fully active while the others are idle (and consume $k_{\text{idle}}P_1$, with $k_{\text{idle}} < 1$). Considering only the app and ignoring the remaining system and processes, consumed energy would be

$$E_{\text{local}} = [(1-F)(N_{\text{c,local}}-1)k_{\text{idle}}+1]P_1(N_I \times CPI \times T).$$[6]

As mentioned earlier, for cloud computing, local energy consumption depends on the overlap between communication and computation times, with consumption obviously lowest for complete overlap. We again consider the worst-case cloud scenario of no overlap. Local energy is the sum of the communication and cloud computation periods. Data transmission adds an extra power P_t. Assuming that the NIC transmits data by directly accessing local memory, then all local cores would be in an off state during both periods. Hence, the energy wasted by the local device would be

$$E_{\text{cloud}} = \frac{N_D}{BW}P_t + \left[(1-F)(N_I \times CPI \times T) + \frac{N_D}{BW}\right]N_{\text{c,local}}k_{\text{off}}P_1.$$

The local device need not be running while waiting for the cloud response. Obviously, the device can stay awake to manage other inner tasks, but this energy consumption cannot be attributed to offloading. From an application viewpoint, the local device could be almost fully off (only waiting for the NIC), which means that k_{off} would be negligible.

Energy efficiency is determined by the performance achievable in the same battery life cycle (that is, with the same energy).[6] The resultant speedup $S_{t\times E}$ is given by

$$S_{t\times E} = \frac{E_{\text{local}}t_{\text{local}}}{E_{\text{cloud}}t_{\text{cloud}}} = S_E S_t.$$

Finally, reordering and using the parameters μ and D_I (and assuming that $k_{\text{off}} = 0$), we obtain

$$S_{t\times E} = \frac{[(1-F)(N_{\text{c,local}}-1)k_{\text{idle}}+1]P_1}{\frac{\mu}{D_I}P_t} \times \frac{\frac{F}{N_{\text{c,local}}}+(1-F)}{\frac{\mu}{D_I}+(1-F)}.$$

Again, three significant results emerge when we consider performance and energy.

First, D_I is the fundamental parameter for determining whether cloud offloading is energetically beneficial. Using technological magnitudes for a typical mobile device ($\mu = 1$, $k_{\text{idle}} = 0.3$,[6] and $P_1 = P_t = 1\text{W}$), $S_{t\times E}$ begins to favor the cloud for moderate D_I values. For simple devices ($N_{\text{c,local}} = 1$) with a $D_I > 1.3$, migration is advantageous ($S_{t\times E} > 1$) for any $F \geq 0.5$. For more powerful devices, the bounds grow a little. Moreover, for apps with low F, migration is more favorable even for lower D_I (and the more powerful the device, the more notable this effect). This result implies that middleware designers should focus on increasing D_I by using compression techniques and good data coding. Furthermore, software engineers should estimate whether future app versions will increase D_I. In general, cloud offloading would not benefit apps with low D_I, such as those that do not reuse input data like video or audio streaming.

Second, if μ's expected reduction continues, these bounds will decrease at a nearly proportional rate. That is, if μ were reduced by one-tenth, the bounds on D_I would decline by approximately 0.1, 0.2, and 0.4 for $N_{\text{c,local}} = 1, 4,$ and 16, respectively. Thus, in the future, technology will favor offloading app execution for embedded devices.

Finally, when $F \to 1$ (which is common in most scientific apps,) or $N_{\text{c,local}} = 1$, speedup is almost proportional to

$$S_{t\times E} = \left(\frac{D_I}{\mu}\right)^2.$$

So, for simple devices or very parallel apps, we can expect to achieve energy efficiency for cloud migration much earlier than for timing speedup (quadratic order).

CHARACTERIZING APPLICATION SUITABILITY FOR CLOUD OFFLOADING

To determine whether a given app might benefit from cloud offloading, we consider two examples.

Example 1

The first example is a very computationally intensive algorithm, matrix multiplication, which is the kernel of many scientific apps. Assuming $A = B \times C$ is the product of $n \times n$ ranked matrices, the calculation basically consists of these three loops:

```
for row = 1...n
    for col = 1...n
        for  k = 1...n
            A[row][col] += B[row][k] * C[k][col]
```

The two outer loops iterate over A's elements, while the inner loop computes the dot product of a row of the first matrix by a column of the second matrix. The total number of multiply operations equals the number of inner-loop iterations and is on the order of $O(n^3)$. For most scientific apps,

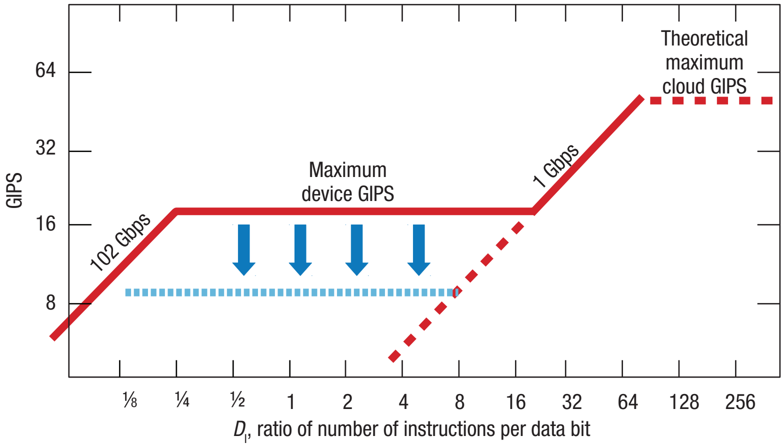

FIGURE 3. Two-roofline model for giga instructions per second (GIPS) versus D_I for local (Qualcomm Snapdragon 610 S4 Pro, with 1.5 GHz quad-core Krait 300) and cloud application execution. Decreased maximum device GIPS is marked with down arrows.

the bigger n is, the more accurate the results will be. However, the high computational order together with the elevated amount of memory makes many programmers reluctant to use big matrices.

What if application execution was offloaded? B and C must be sent to the cloud, and, after processing, A must be returned back—that is, $O(n^2)$ bits would be exchanged. Hence, $D_I = O(n^3)/O(n^2) = O(n)$. All of A's elements can be computed in parallel, and only the sum of products—$O(n)$—must be done sequentially, which gives $1 - F = O(n)/O(n^3) = O(n^{-2})$. Finally, $S_t = O(n)$, $S_{t \times E} = O(n^2)$. Hence, the more accurate the results desired, the more speedup can be extracted from cloud execution. In sum, not only μ but also F and D_I favor cloud offloading.

Example 2

The second example is crucial to many robotic apps: processing frame pairs captured by a stereo camera. The "Stereo Vision Offloading for Mobile Robots" sidebar shows that this processing can be designed not only to be parallel but also to establish an efficient pipeline with transmission times. Hence, parallel ratio F is very near to 1 for a medium-resolution image. The sidebar shows that the usual complexities of these algorithms are on the order of $O(N^3)$, N being the horizontal image size. Because transmitted data are proportional to image resolution, that is, $O(N^2)$, D_I is again $O(N)$. So, the higher the image's resolution, the more speedup can be achieved through cloud offloading.

This reasoning can be extended to most cases. For instance, scientific apps are usually parallel and have complexity whose order varies between $O(n \log n)$—for example, fast Fourier transform (FFT) calculations—and $O(n)$—for example, finite element–based calculations—for a data size n.[3] This means that whereas F approaches 1, D_I grows like $O(\log n)$ or remains constant. In the FFT case, the more data used, the more speedup will be achieved, whereas with finite element–based computation, speedup benefits will only be possible once technological progress increases by μ.

These examples lead to two main conclusions.

First, for many apps, F and D_I grow with problem complexity. This implies that as new apps require more accurate solutions, cloud offloading becomes more viable. Furthermore, software designers and researchers should strive to decrease N_{Data} by using compression techniques and good data coding[7] to increase D_I.

Second, and contrary to expectations, remote execution of apps that have high CPI and giga instructions per second (GIPS) far below the theoretical maximum[3] (marked by down arrows in Figure 3) might be the best option. For $F = 1$, the maximum performance that can be achieved through local and cloud execution of such an app is depicted in Figure 3, which is based on a real case (Snapdragon 610 S4 Pro). The abscissa represents different D_I values, while the ordinate represents GIPS. This extension of the roofline model for GIPS,[8] rather than giga floating-point operations per second (Gflops), results in a two-roofline model. Maximum device GIPS is calculated as the inverse of clocks/instruction × seconds/clock, multiplied by the number of cores $N_{c,local}$, that is, $(CPI_{minimum} \times T)^{-1} N_{c,local}$. Theoretical maximum cloud GIPS would be calculated similarly (if cloud resources were finite). When execution implies many RAM accesses, this maximum cannot be reached: the product instructions/data bit × data bit/seconds, that is, $D_I \times BW_{RAM}$, gives the first roofline. The second roofline is obtained for the network connection between the device and the cloud: $D_I \times BW$. If prolongation of the second roofline (dotted red line in Figure 3) crossed real device GIPS, remote execution would achieve the same GIPS as local execution. Other benefits, such as energy savings, might also motivate use of the remote option.

LESSONS FROM HISTORY

For apps that require storing vast amounts of data on a server, cloud computing is the only option. But what does our extension of Amdahl's law predict about those apps that elude offloading?

If Moore's law continues to hold for the next decade, cloud computing will clearly benefit from the effect of technological progress on μ. For holistic factors such as F and D_I, the matter is less simple. Nevertheless, we can learn some valuable lessons from history. Amdahl argued that $1 - F$ values were large enough to favor single processors. Today's enhanced machines allow massive computations beyond anything he could have envisioned,[9] making F values soar very close to 1. Thus, F can be stretched as more computation resources become available.

We can envision D_I's evolution. One of Myhrvold's laws states that "software is a gas: it expands to fill any size hardware container."[10] Users or developers might notice that a remotely executed app is more powerful (and maybe faster and more energy efficient) than its locally executed counterpart, which runs in a smaller hardware container. Or mobile programmers might discover that they have unlimited computation power in the cloud. We wager that these users would ditch the local option and embrace the cloud, and that the efforts needed to transform a complex algorithm to run faster on a low-power computer could be turned toward other, more productive purposes. Hence, D_I will very likely increase when all programmers easily and transparently use the cloud. In sum, S_t and $S_{t\times E}$ are being boosted by these three factors (μ, F, and D_I).

Even apps that must run close to the target system tend to be offloaded to the cloud nowadays. Platforms such as the ARM mbed IoT Device Platform (www.mbed.com) offer a remote compilation for embedded systems and the possibility of generating a development project to be loaded into a certain device. By enabling embedded-system programmers to work in their preferred environment and with their favorite tools, the cloud makes app development both easier and faster.

ABOUT THE AUTHORS

FERNANDO DÍAZ-DEL-RÍO is an associate professor at the University of Seville. His research interests include mobile robot navigation, bioinspired systems, and distributed computing systems. Díaz-del-Río received a PhD in physics (electronics) from the University of Seville. Contact him at fdiaz@us.es.

JAVIER SALMERÓN-GARCÍA is a PhD student and part-time lecturer at the University of Seville. His research focus is cloud robotics. Salmerón-García received an MSc in computer engineering and networks from the University of Seville, and an MSc in software engineering for technical computing from Cranfield University. Contact him at jsalmeron2@us.es.

JOSÉ LUIS SEVILLANO is an associate professor at the University of Seville. His research interests include real-time communications and architectures, mobile robots, and eHealth and rehabilitation systems. Sevillano is an associate editor of *Simulation* and the *International Journal of Communication Systems*. He is an IEEE Senior Member. Contact him at jlsevillano@us.es.

The real world is much more complex than a theoretical model that tries to encompass major trends and must include several assumptions. Some of our model's assumptions benefit cloud offloading, such as overlooking overheads from virtualization, middleware, and OS, or not considering the time required to divide computation between local and external resources. Other assumptions benefit local execution, such as considering no overlap between communication and computation, or supposing that resources (for example, memory, caches, and bus speeds) are identical for the cloud and the local system. The literature shows how complex these practical issues can be.[11]

Two practical issues make the cloud attractive for app developers. First, there has been substantial R&D into making cloud facilities commercially successful, which should help real cloud execution times soon approach those of our model. Second, migrating apps to the cloud would simplify local devices at minimal cost. This benefit will likely be brought about by efficient networking and distributed computing techniques, which could pave the way for new programming paradigms that contemplate automatic code migration as a new form of computation.

ACKNOWLEDGMENTS

This work was supported by the Spanish grant BIOSENSE TEC2012-37868-C04-02/01 (with support from the European Regional Development Fund).

REFERENCES

1. G. Van Huizen, "Hype Cycle for Application Architecture, 2013," Gartner, 31 July 2013; www.gartner.com/doc/2569522/hype-cycle-application-architecture-.
2. M. Armbrust et al., "Above the Clouds: A Berkeley View of Cloud Computing," tech. report, EECS Dept., Univ. California, Berkeley, Feb. 2009; www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html.
3. J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach,* 5th ed., Morgan Kaufmann, 2011.
4. R. Friedman, A. Kogan, and Y. Krivolapov, "On Power and Throughput Tradeoffs of WiFi and Bluetooth in Smartphones," *IEEE Trans. Mobile Computing*, vol. 12, no. 7, 2013, pp. 1363–1376.
5. K. Kumar and Y.-H. Lu, "Cloud Computing for Mobile Users: Can Offloading Computation Save Energy?," *Computer*, vol. 43, no. 4, 2010, pp. 51–56.
6. D.H. Woo and H.-H.S. Lee, "Extending Amdahl's Law for Energy-Efficient Computing in the Many-Core Era," *Computer*, vol. 41, no. 12, 2008, pp. 24–31.
7. E. Tilevich and Y.-W. Kwon, "Cloud-Based Execution to Improve Mobile Application Energy Efficiency," *Computer*, vol. 47, no. 1, 2014, pp. 75–77.
8. S. Williams, A. Waterman, and D. Patterson, "Roofline: An Insightful Visual Performance Model for Multicore Architectures," *Comm. ACM*, vol. 52, no. 4, 2009, pp. 65–76.
9. M.D. Hill and M.R. Marty, "Amdahl's Law in the Multicore Era," *Computer*, vol. 41, no. 7, 2008, pp. 33–38.
10. W. Wayt Gibbs, "Taking Computers To Task," *Scientific American*, vol. 277, no. 1, 1997, pp. 82–89.
11. M.S. Gordon et al., "COMET: Code Offload by Migrating Execution Transparently," *Proc. 10th USENIX Conf. Symp. Operating Systems Design and Implementation* (OSDI 12), 2012, pp. 93–106.