Outlining the strategy for high performance computing for the Health Care Company.

James M. Lamb and Pawel Manikowski

As our understanding of biological processes and pathology grows, we are invariably faced with greater, more complex stochastic biochemical systems and also the larger sets of data that are ascertained therein. This presents a problem within the health-care industry as a whole; How can we manage and analyse this expanding set of data in a timely and efficient manner, and how can we continue to model and simulate ever more complicated biochemical reactions?

Traditional computing systems are incapable of providing the computational power to do so, moreover, the rate at which computational power is increasing is very unlikely to ever catch up with the demands set by health organisations. In order to produce viable vaccines and preventative medicines for emerging viral mutations, it is an inevitable conclusion that we must turn to high performance computing (HPC) as a solution.

Typically, drug development, for example, can take anywhere between 10 and 17 years. This timescale, coupled with the rising cost of testing drug candidate molecules is generally making traditional testing methods unsustainable. However, by utilising a large number of parallel operations on a set of data, we can more easily simulate multiple simultaneous biochemical processes such as protein mutations and also analyse the culminating wide range of data in a similar manner, both increasing the scope of research and vastly reducing the time and cost it would take to do than by using traditional methods.

In a more practical form, HPC provides direct utility to health-care professionals and patients. To illustrate, for the detection of rare diseases, it is becoming a prevalent feature of diagnostics to rely on substantial amounts of data provided by sensors and mobile devices, which are analysed to create a patient profile and generate personalised disease management plans. This data, considering the increasing number of patients involved, is already far too broad for computational processing on a traditional system and as more practices move into real time analysis of this data so too do those traditional systems become superannuated. Yet the implementation of data analysis in real time only compounds the benefits to both patients and health-care providers, in that it can more easily indicate warning signs of ill health to ensure preventative measures are taken to reduce further development of time-critical ailments, for instance, to warn sufferers of cataplexy or seizures of imminent episodes. (Evropskyvyzkum.cz, 2018)

One emergent application of HPC within precision medicine lies within the sequencing of genomes, this diagnostic technique has already revolutionised modern medicine practices to providing rapid analysis of an individual's current and prospective health, providing a clear early warning to health-care professionals of potential genetic susceptibility to specific diseases. This process of next generation sequencing is entirely possible due to the application of HPC, as large data sets from whole genome sequencing or thousands of exomes have to be analyzed, and hence more storage and computational power is required than most conventional hardware can provide. (Kawalia et al., 2015)

Types of HPC Systems

Fundamentally, supercomputers are either multi-core systems or multi-nodal systems. The former being a computer that contains multiple processing units called "cores" typically on an integrated circuit (IC). The cores can read and execute CPU instructions such as add, move data and branch as expected but can implement multiprocessing, meaning it can run multiple instructions on each separate core simultaneously. There are variations of this premise, wherein the individual cores may or may not be heterogeneous and may have independent caches, utilise shared memory and implement message passing. This kind of system is ideal for perfectly parallel problems; the number of cores being a factor in the determination of processing speed. However, in less simplified parallel workloads, the process's acceleration is very much limited to the parallelism of the program running; most current software and techniques are not able to exploit multi-core architectures (Hasib, 2018).

Another factor to consider in a multicore system is the proximity of the caches and number of CPU cores on the same die. Having multiple CPUs and caches on a single die means that the signals between them have to travel shorter distances and in turn reduce their degradation compared to signals travelling off chip, this results in higher clock rates and thereby larger quantities of data transmission. This also implies that the system would have a higher ratio of performance per watt. This improvement is limited, however, to silicon real-estate; higher core density leaves less room for caches, increases heat generation and synchronously increases the physical core-to-cache distance and hence the signal latency. (Lotfi-Kamran et al., 2012)

The alternative then is a multi-node system, otherwise referred to as a computer cluster, comprising of multiple nodes (computers acting as servers) running individual instances of an operating system. These nodes are interconnected, usually via a local area network, and perform parallel operations.

A multi-node system is a much more versatile approach to HPC. There are three main forms, the first being a high availability cluster. These systems utilise redundant nodes which are employed as and when another node fails. This helps to ensure that the entire system is not subject to the functionality of any single node and improves reliability of the entire system.

Another category is a load balancing system; when a request is made to the cluster it is sent to the node that is the least busy and thereby improves the overall response time of the cluster.

Finally, there is the HPC cluster format, in that each node may perform similar calculations on some partition of data, or even that each would perform a unique operation on a common set of data or some combination of the two; much like, if not a direct example of, parallel processing. (The openMosix HOWTO, 2018)

Both of these types of systems utilise one of three types of memory allocation, the first of which is shared memory. As the name suggests, each CPU core or cluster node has access to a single large block of RAM, access to this is defined by the shared memory architecture. Uniform memory access (UMA), being one example, where access time to a location in memory is not dependent on which processor makes the request, whereas in a non uniform memory access (NUMA), memory is allocated to the cache local to the CPU executing the code. Although other approaches are possible if preordained within the program itself, this essentially means that there are two types of cache within a NUMA architecture: local memory and foreign memory, CPUs have priority over local memory and access to foreign memory is generally accessed through an interconnect that restricts and permits access as required. (Rajput, Kumar and Patle, 2018) The final architecture type is cache-only memory access (COMA), which is very similar to NUMA however rather than accessing memory as foreign memory, the data is instead replicated and migrated to the local memory cache of the CPU making the request (Dahlgren and Torrellas, 1999).

The second type of memory allocation is distributed memory, in this architecture, memory is not shared between CPUs or nodes, instead each have individual memory, data is then transmitted through explicit communication via an interconnection, typically utilising a message passing interface (MPI). This method, although less volatile than using shared memory, requires communication commands within a program and therefore more complex code (Computing.llnl.gov, 2018).

Finally, there is distributed shared memory (DSM), an amalgamation of the previous forms of memory storage wherein each CPU or node has it's individual memory and access to it but the address space of specific data in memory is shared. This is ordinarily maintained by some form of intermediary connection that utilises a request and response methodology to maintain memory coherence and consistency, or via a MPI. It is also commonplace for a DSM architecture to provide the abstraction of one global memory from which data is referenced by a CPU or node and implicitly replicated and migrated to the respective distributed memory (Reinhardt, 1996).

Finally there are graphical processing units (GPUs), or more specifically within HPC applications, general purpose graphical processing units (GPGPUs). These are essentially specialised accelerators containing hundreds of processors designed specifically for the purpose of floating point calculations. GPUs have a higher number of cores but smaller cache memory and registers and are therefore most beneficial to applications that involve a large amount of floating point processing; medical imaging and data analytics, for example. However these are vastly unsuitable for database and SQL query based applications (Ghorpade, 2012).

Types of Programming Models

Generally speaking, there are two principal areas that define the kind of programming model used in parallel computing, these are process interaction and problem decomposition.

Essentially, the first is tied to the kind of communication between nodes or CPUs, as we have covered the basic architecture of memory types, I shall elaborate on specific communication methods, beginning with the most widely used of these, a message passing interface (MPI).

The MPI is a library specification that allows information to be passed between different processes running on multiple processors, it has been developed as a standard interface and therefore extends the portability of applications and libraries, making it ideal for a variety of different hardware designs, it is now regarded as the "de facto" standard for programming parallel systems, and is present in the majority of the top listed supercomputers worldwide, (Ref2014impact.azurewebsites.net, 2018). Essentially MPI is a distributed memory model, regardless of the physical architecture of the machine utilising it but will run virtually on any hardware platform. It is versatile in that it can be both synchronous and asynchronous at a programmers whim and furthermore, the parallelism of MPI is explicit and hence is dependent on the program itself implementing parallel algorithms.

Outside of the widely used MPI, there are two types of message passing models, synchronous, wherein a thread that sends a message to another thread will become blocked until the receiving thread responds; an example of a two way communication between threads. Whereas in a asynchronous message passing interface, the thread sending a message will not wait for a response, merely sending the data to the message bus and continuing with its own processes.

Alternatively to explicit models, implicit interaction is automatically performed by the compiler or runtime system, although this may speed up the programming of a particular process, this will generally result in less optimised parallelism within the code in less apparently parallel problems and result in system slowdowns at runtime. In the case of complex protein analysis, for example, a system utilising implicit process interactions would not be able to automatically determine an ideal method for parallelising the theoretical decompositions and mutations over time. Instead only parallelising the sequential segments of code. Therefore if an implicit programming model were to be utilised, I would highly recommend augmenting that system with some kind of functional language to provide some method by which to manage parallelism explicitly, such as Concurrent Haskell (Downloads.haskell.org, 2018).

Implicit interaction is inherently implicit in its problem decomposition, however, programming models vary too here in explicit regards. Predominantly, task parallelism, disregarding implicit variants, is split into two veins; task parallelism and data parallelism. In the former, a program considers how a process can be parallelised into distinct methods. Again, using protein analysis as an example, one might consider a single dataset of a specified number of globular proteins on each core or node of the system, but have each of those cores or nodes perform a different point-mutation and determination of the resulting structure (Rodrigues, Pires and Ascher, 2018).

Inversely, a program model may utilise data parallelism, in which each node or core will perform the same computation but each on a specific partition of a data set. As another example, we can consider the analysis of genomics data in patients to find statistically significant biological markers and indicate how patients may respond to specific drugs or there susceptibility to known complex diseases. The colossal amount of data involved can be broken into many partitions and each core or node can perform a single process, a testing algorithm, to each partition simultaneously, thus reducing overall processing time dramatically (Agapito, Guzzi and Cannataro, 2017).

Cloud HPC Consideration

Cloud services are becoming more prominent in recent times. As demand for HPC grows so to does it's capability to provide appropriate systems at cost effective prices. Determining the type of HPC system to implement within the Healthcare industry has it's apparent difficulties, namely in that the optimum architecture and program model required within the research and development of new drugs and arising diseases is not the ideal system to implement for the occupational use of front line medical professionals.

Cloud HPC services are an effective solution to this issue, services provided by leading technology companies such as Amazon, Google and Microsoft offer customisable cloud HPC systems that allow your company to design and implement customised HPC systems and therefore tailor the machines you access to the specific role it is required for. This in itself reduces the risk in the investment and ensures that even if the initial system implemented has some overlooked complications in its hardware, those problems can be mitigated by swapping specific hardware without further expenditure.

Additionally, these services have a range of methodologies to their funding; Rental of cloud based HPC can be either a flat rate subscription fee or a "pay as you go" fee, further reducing the risk of investment. Also as these systems are in commercial competition with one another, the price too is inherently competitive, and therefore vastly more affordable.

The high demand for cloud based HPC is also driving the advancement of these services, as each provider aims to provide more than its competitor the services provided inevitably expand and existing hardware updated regularly, resulting in a HPC system that is constantly upgraded over time without the drawback of providing additional capital.

Finally, cloud based HPC environments are ideal for large scale collaborative work, access to the system via the cloud means that it is accessible anywhere, without needing a specific large amount of physical space, hence promoting teams to work in tandem with one another without being restricted by location or time constraints.

Contrarily, cloud computing does have it's drawbacks. The speed of a cloud based HPC system is inherently tied to the latency of the system, where a HPC setup is typically kept in close proximity and connected via fast local area networks, cloud HPC systems can be spread across multiple data centres and are therefore subject to higher latency (Expósito et al., 2013).

Cost again is a matter of consideration, although cloud HPC systems are indeed the affordable choice when it comes to applications that are subject to change over time or for multiple applications running in tandem, the cost of using these systems are subject to how much those systems are used. Were it that a system be entirely purpose built for a specific function, ie. Gene sequencing, and that system needed to be run regularly with little downtime, then it would invariably be cheaper in the long term to do so off the cloud. This is especially so in the case of systems that utilise and/or generate large amounts of data, as typically the data storage provided by these cloud services are not unlimited and further costs may need to be taken into account in that regard.

Data security is also a concern, regardless of how secure a cloud service may be, transmitting data is still an avenue for would be hackers and hence to err on the side of caution in regards to the sensitivity of transmitted data is obviously recommended.

Bibliography

1. The openMosix HOWTO. (2018). *A very, very brief introduction to clustering.* [online] Available at: https://www.tldp.org/HOWTO/openMosix-HOWTO/x135.html [Accessed 2 Nov. 2018].
2. Agapito, G., Guzzi, P. and Cannataro, M. (2017). Parallel extraction of association rules from genomics data. *Applied Mathematics and Computation.*
3. Downloads.haskell.org. (2018). *Control.Concurrent.* [online] Available at: https://downloads.haskell.org/~ghc/latest/docs/html/libraries/base-4.12.0.0/Control-Concurrent.html [Accessed 2 Nov. 2018].
4. Dahlgren, F. and Torrellas, J. (1999). Cache-only memory architectures. *Computer*, 32(6), pp.72-79.
5. Expósito, R., Taboada, G., Ramos, S., Touriño, J. and Doallo, R. (2013). Performance analysis of HPC applications in the cloud. *Future Generation Computer Systems*, 29(1), pp.218-229.
6. Ghorpade, J. (2012). GPGPU Processing in CUDA Architecture. *Advanced Computing: An International Journal*, 3(1), pp.105-120.
7. Hasib, A. (2018). *Energy Efficient Computing on Multi-core Processors.* ph.D. Norwegian University of Science and Technology.
8. Evropskyvyzkum.cz. (2018). *HPD - Best Use Examples.* [online] Available at: https://www.evropskyvyzkum.cz/cs/storage/db6d926493a0627c7a18bbc9d49db7d1fdb153a4?uid=db6d926493a0627c7a18bbc9d49db7d1fdb153a4 [Accessed 2 Nov. 2018].
9. Ref2014impact.azurewebsites.net. (2018). *Institution: PHYESTA.* [online] Available at: https://ref2014impact.azurewebsites.net/casestudies2/refservice.svc/GetCaseStudyPDF/35271 [Accessed 2 Nov. 2018].
10. Computing.llnl.gov. (2018). *Introduction to Parallel Computing.* [online] Available at: https://computing.llnl.gov/tutorials/parallel_comp/#DistributedMemory [Accessed 2 Nov. 2018].
11. Kawalia, A., Motameny, S., Wonczak, S., Thiele, H., Nieroda, L., Jabbari, K., Borowski, S., Sinha, V., Gunia, W., Lang, U., Achter, V. and Nürnberg, P. (2015). Leveraging the Power of High Performance Computing for Next Generation Sequencing Data Analysis: Tricks and Twists from a High Throughput Exome Workflow. *PLOS ONE*, 10(5), p.e0126321.
12. Lotfi-Kamran, P., Ozer, E., Falsafi, B., Grot, B., Ferdman, M., Volos, S., Kocberber, O., Picorel, J., Adileh, A., Jevdjic, D. and Idgunji, S. (2012). Scale-out processors. *ACM SIGARCH Computer Architecture News*, 40(3), p.500.
13. Rajput, V., Kumar, S. and Patle, V. (2018). Performance Analysis of UMA and NUMA Models. *IJCSET*, 2(10), pp.1457-1458.
14. Reinhardt, S. (1996). *Mechanisms for Distributed Shared Memory.* Ph.D. UNIVERSITY OF WISCONSIN—MADISON.
15. Rodrigues, C., Pires, D. and Ascher, D. (2018). DynaMut: predicting the impact of mutations on protein conformation, flexibility and stability. *Nucleic Acids Research*, 46(W1), pp.W350-W355.