An Outlook of High Performance Computing Infrastructures for Scientific Computing

[This draft has mainly been extracted from the PhD Thesis of Amjad Ali at Center for Advanced Studies in Pure and Applied Mathematics (CASPAM), Bahauddin Zakariya University, Multan, Pakitsan 60800. (amjadali@bzu.edu.pk)]

In natural sciences, two conventional ways of carrying out studies and reserach are: *theoretical* and *experimental*. The first approach is about theoratical treatmment possibly dealing with the mathematical models of the concerning physical phenomena and analytical solutions. The other approach is to perform physical experiments to carry out studies. This approach is directly favourable for developing the products in engineering. The theoretical branch, although very rich in concepts, has been able to develop only a few analytical methods applicable to rather simple problems. On the other hand the experimental branch involves heavy costs and specific arrangements. These shortcomings with the two approaches motivated the scientists to look around for a third or complementary choice, i.e., the use of ***numerical simulations*** in science and engineering. The numerical solution of scientific problems, by its very nature, requires high computational power. As the world has been continuously enjoying substantial growth in computer technologies during the past three decades, the numerical approach appeared to be more and more pragmatic. This constituted a complete new branch in each field of science, often termed as ***Computational Science***, that is about using numerical methods for solving the mathematical formulations of the science problems. Developments in computational sciences constituted a new discipline, namely the ***Scientific Computing***, that has emerged with the spirit of obtaining **qualitative predictions** through simulations in 'virtual laboratories' (i.e., the computers) for all areas of science and engineering. The scientific computing, due to its nature, exists at the overlapping region of the three disciplines, *mathematical modeling*, *numerical mathematics* and *computer science*, as shown in Fig. (1). Thus, it requires multi-disciplinary expertise for obtaining effective and industrious outcomes through it. An excellent elaboration of this interdisciplinary blend is given by Bungartz et al. [1], shown in Fig. (2). The main enabling factors

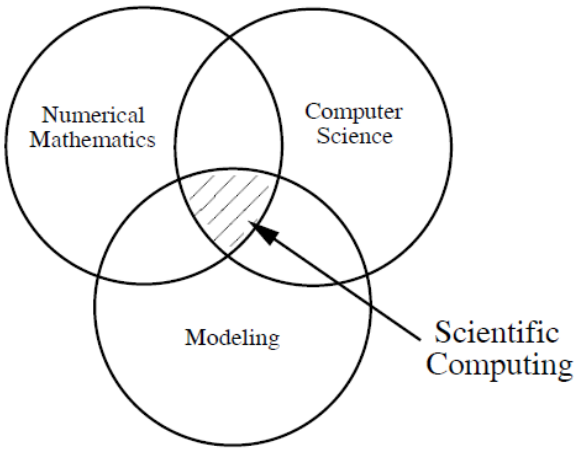


Figure 1: "Definition of scientific computing as the intersection of numerical mathematics, computer science, and modeling" (taken from the Book "Parallel Scientific Computing in C++ and MPI" by Karniadakis and Kirby)

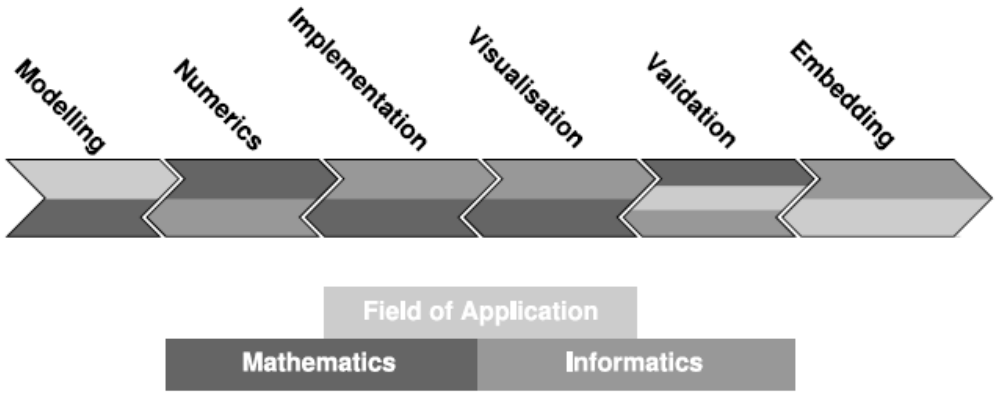


Figure 2: "The simulation pipeline and its stages, involving input from mathematics, informatics, and the respective field of application" (taken from Ref. [1] by Bungartz et al.)

for so impressive advancements in scientific computing during the past three decades are enormous growth in computing capabilities, steep decline in computing cost and development of more and more efficient algorithms [2]. It is quite interesting to note that these factors mutually serve as stimulating agents for one another in their enhancement.

A conventional methodolgy in scientific computing is to implement the solution algorithm on computers using some appropriate programming environments and libraries, and executed on some workstation or parallel computer. Some necessary steps, like for example collecting input data in some special format, could be performed in a separate *pre-processing* phase. The main computational procedure is

implemented in the principal software application program/code and on execution it produces output data files that might grow upto Megabytes to Gigabytes in their size. Finally in the last step, the *post-processing* phase, useful conclusions are drawn from the output files by making use of appropriate graphical/visualization tools. Note that software *testing* (**code verification** and **model validation**) and *maintenance* are two important aspects in scientific computing to be considered with due attention, especially for the large scale *open-source* applications which usually evolve for advancement in the science through a community effort. These considerations are not about the execution performance or speed of the code as such but they turn out to be key factors in getting productive use of the scientific application in its life cycle.

Computers have become indispensable for numerical simulations of practical problems in science and engineering. In fact, a key factor for enormous success in computational sciences has been the rapid developments in HPC capabilities during the past two decades. Thus, realization of the essential cooperation between the computer science and numerical simulation is quite important [1]. Alongwith developing a hierarchy of efficient algorithms for the scientific applications, an important need is to implement the complete application in some *high performance computing* (HPC) environment to obtain the solution in a reasonable time frame. The most commonly used and recognized form of obtaining a HPC based solution is to perform parallelization of the application, so that a number of processors work together on different parts of the problem or domain to reduce the overall time required to solve the problem. From a hardware perspective, HPC may be realized on a variety of distributed memory architectures, shared memory architectures and hybrid architectures. From a software perspective, HPC relies on utilizing efficient compilers, mathematical kernels and sophisticated parallelization libraries for an underlying architecture. Further, tuning the application with reference to an underlying architecture may also result into very fast solutions.

A variety of industrial innovations in computer technologies and methodologies for high performance computing is getting introduced year by year. These technologies motivate the computational scientists to develop innovative solutions for even faster and detailed analyses and simulations. The present manuscript overviews the spectrum of high performance computing (HPC) technologies ranging from a modern

uniprocessor to shared memory (multicore) machines, to distributed memory clusters, to hybrid parallel systems. It also lists a variety of high performance parallel programming paradigms and methodologies for the said parallel computer architectures. Some basic parallel performance metrics are also presented.

0.1 Modern Microprocessor Based Computer Systems

The basic physical organization of a modern computer, based on the *von Neumann architecture* model, comprises five units, namely Memory, Control, Arithmetic-&-Logic, Input and Output. The *central processing unit* (CPU) comprises control and arithmetic-&-logic units. The functioning of a computer is precisely the execution of *instructions* to process *data* by its CPU. *Instructions* are the primitive operations that the CPU may execute, such as moving the contents of a memory location (called as *register*) to another memory location within the CPU, or adding the contents of two CPU registers. Control unit fetches the data/instruction from the *system memory* or *main memory,* sometimes also referred to as *random access memory* (RAM). The data is then processed by the Arithmetic-&-Logic unit, **sequentially**, according to the instructions decoded by the control unit. Storing both the data and instructions in the single main memory unit is an essential feature of the von-Neumann architecture. Input and Output units provide interface between computer and the human.

Not only the CPU, the memory system of a computer also play a crucial rule for overall computer performance in performing the computations. The memory system of a modern computer is complicated one. A number of smaller and faster memory units, called *cache memories* or simply *caches*, are placed between the CPU and the main memory. These caches, existing at a number of levels, form a memory hierarchy in which *access time* and *size* increases as moved away from a level that is nearer to the CPU to a level that is farther. The idea of a cache memory is to bring only some part of the program data needed currently from main memory into the cache to speed up the data access by the CPU. The memory hierarchy (combining smaller and faster caches with larger, slower and cheaper main memory) behaves most of the time like a fast and large memory. This is mainly due to fact that the caches are to exploit the feature of *locality of memory references*, also called *principle of locality*, which is

often exhibited by the computer programs. Common types of locality of reference include *spatial locality* (local in space) and *temporal locality* (local in time). Spatial locality of reference occurs when a program accesses data that is stored contiguously (for example, elements of an array) within short period of time. Caches are used to exploit this feature of spatial locality by pre-fetching from the main memory some data contiguous to the requested one, into a cache. Temporal locality of reference occurs when a program accesses a used data item again after a short period of time (for example, in a loop). Caches are used to exploit this feature of temporal locality by retaining recently used data into a cache for some period of time. Note that the locality of reference is a property of computer programs but is exploited in memory system design through the caches. This, definitely, indicates that during coding a programmer should take care to develop the code so as to enhance both type of localities of reference for efficient cache utilization. This could be achieved by coding in a way that data is accessed in a sequential/contiguous fashion and, if required to be reused, is accessed again soon as much as possible.

A modern CPU (microprocessor) executes (at least) one instruction per clock cycle. Each different type of CPU architecture has its unique set of instructions, called its *instruction set architecture* (ISA). The instruction set architecture of a computer can be thought of the language which the computer can understand. Based on the type of ISA, two important classes of modern (microprocessor based) computer architectures are: *CISC* (Complex Instruction Set Computer) architecture and *RISC* (Reduced Instruction Set Computer) architecture. The basic CISC architecture is essentially the von Neumann architecture in the sense of storing both instruction and data inside a common memory unit. On the other hand, the basic RISC architecture has two entirely separate memory spaces for instructions and data, which is the feature that was introduced in Harvard architecture to overcome the bottleneck in von Neumann architecture due to data-instruction shared paths between CPU and the memory. CISC philosophy is that the ISA has a large number of instructions (and addressing modes, as well) with varying number of required clock cycles and execution time. Also certain instructions can perform multiple primitive operations. RISC philosophy is that the ISA has a small number of primitive instructions for ease in hardware manufacturing and thus the complicated operations are performed,

at program level, by combining simpler ones. Due to its very nature, a RISC architecture is usually experienced to be more faster and efficient than a comparable CISC architecture. However, due to continuing quest for enhancement and flexibility, today a CPU executing an ISA based on CISC may exhibits certain characteristics of RICS and vice versa. Thus, the features of CISC and RISC architectures have been morphing with each other. Classic CISC architecture examples include VAX (by DEC), PDP-11 (by DEC), Motorola 68000 (by Freescale/Motorola) and x86 (mainly by Intel). Modern CISC architecture, x86-64, based computers like Pentium (by Intel) and Athlon (by AMD) basically evolved from the classic CISC architecture x86, but they exhibit several RISC features. Currently, Xeon (by Intel) and Opteron (by AMD) are the two quite prominent market icons based on x86-64 architecture. Famous RISC architecture examples include MIPS (by MIPS Technologies), Power (mainly by IBM), SPARC (mainly by SUN/Oracle), Alpha (by DEC) and ARM for embedded systems (by ARM Ltd.).

Today, Intel and AMD are the two major vendors in the microprocessor industry, each with their own line of CPU architectures. The x86-64 CPUs from Intel and AMD, basically emerged as the CISC architecture, now incorporate a number of RISC features, especially to provide for *Instructions Level Parallelism - ILP* (details later on). Interestingly, today the microprocessors (from Intel and AMD) implement the feature of separate memory space for data and instructions for Level-1 caches (at least).

Another main specialty of a modern CPU is that a number of CPU cores are fused together on a single chip/die with a common integrated memory controller for all the cores. Initially dualcore CPU chips were introduced around the year 2005 but, as of 2012, 12/16-core CPU chips are commonly available in the market, although the price might get manifold with linear increase in the number of cores per chip. Moreover, getting the best performance out of larger number of cores in a single CPU is currently a challenging task, mainly due to memory bandwidth limitations. Mutlicore CPUs provide for more clock cycles by the summing the clock cycles contributed by each of its cores. Thus keeping the well-known *Moore's law* effective, today, to some extent. Infact, they provide for tackling the issues of high power requirements and heat dissipation realized in the case when all the cores are there in separate CPU-chips,

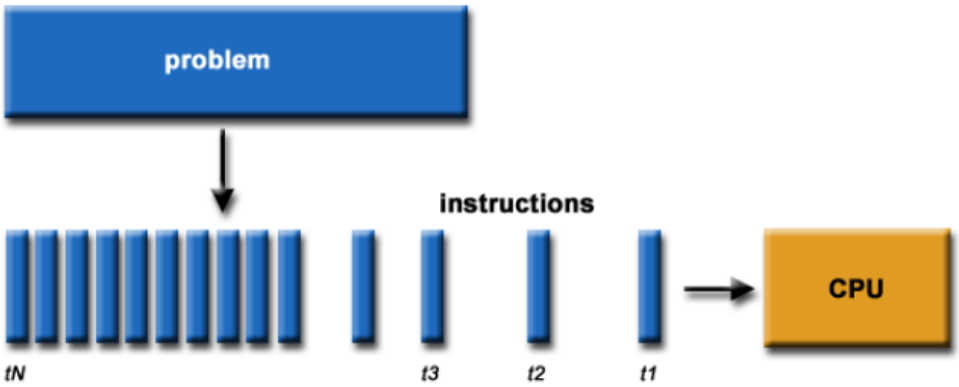
instead of being part of a single CPU-chip [3]. Increment in the clock frequency of a single CPU core (silicon based) is virtually no more feasible due to physical and practical obstacles. Multicore technology is the posed and accepted solution to this limitation.

Another sophisticated architectural innovation in a large class of modern CPUs is the multithreading facility per CPU core. A physical core act as to provide more than one (usually two) *logical processors* that might be benefited by the application in hand. The common realizations of this concept include *hyperthreading*, *symmetric multithreading* (SMT) and *chip multithreading* (CMT). A concise and beautiful introduction to this topic, also to the overall features of modern processors is given by Hager and Wellein ([4], 1−36). Implications of several of the architectural features of modern processors (especially the multicore, multithreading and ILP) are discussed in the coming sections.

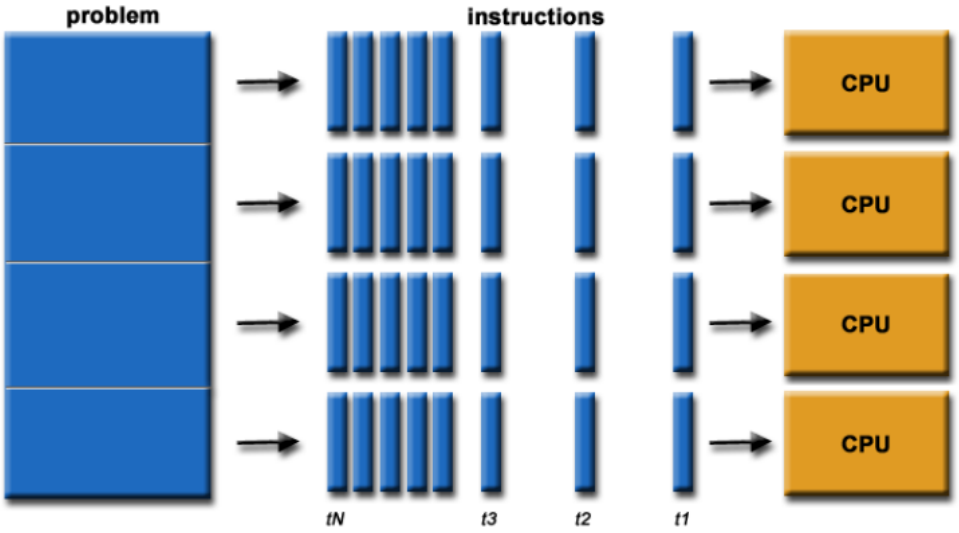
0.2 Computing Shift from Serial to Parallel

Traditional *serial* computation refers to the execution of program instructions sequentially, one after the other, on a single processor. The ever increasing demand of processing power for solving computational problems has been compelling for innovative ways of responding to the need, beyond the level of conventional serial computing. ***High Performance Computing*** **(*HPC*)** is the field that is about the quest of developing and implementing innovative methodologies and technologies, for both the hardware and software, to respond the ever increasing demand of processing capability. The most prominent way to fulfill the high performance computing needs is considered to be ***Parallel Computing***. Parallel computing refers to solve a computational problem by working on different parts of the problem simultaneously by multiple processing units. The processing units could be processors or parts of an individual processor. The different parts of the problem could be different tasks/operations, or the same task on different pieces of the problem's data. Fig. (3), taken from [5], elaborates the basic difference between serial and parallel executions.

Parallel computing not only provides for the concurrency, it might also provide the capability to solve large problems that were somehow impossible to be solve by



(a) Serial execution on a single processing unit



(b) Parallel execution on multiple processing units

Figure 3: The basic concept of serial and parallel execution of the solution program. (Figures taken from Barney)

serial computing approach. A favouable fact to parallel computing is worth to mention that the programmers/scientists can recognize or define some one or other form of potential of parallelism in the real world problems to solve them using the parallel approach [5]. Previously, a variety of 'high-end' parallel computing architectures (both distributed and shared memory systems) were the only choices as the parallel computers. They were very costly and owned by rich organizations/institutions. But with the advent of *compute clusters* (which could be composed by interconnecting ordinary microprocessor based personal computers) and multicore CPUs as the cost effective parallel computers, achieving high performance parallel computing has come down to an individual user's desktop level. These systems are composed of mass-market commodity off-the-shelf (M^2COTS) hardware components and may be refferred to as ***commodity parallel systems***. An important key point to remember in this regard is that all neccessitates the user or programmer to opt some apt methodology to take advantage of these innovations and modern trends. Making use of agglomeration of physically independent or isolated computing resources (could be referred to as *explicit parallelism*) is not the only form of parallel computing. Several hardware level architectural innovations, like instruction level parallelism (ILP), within a single processing core have given rise to, so called, *implicit parallelism*. Precisely saying, today the parallelism could be categorized as *Implicit Parallelism* and *Explicit Parallelism* [6], from a programmer's perspective. The next two sections are devoted to these two kinds of parallelism.

0.3 Implicit Parallelism

Implicit parallelism refers to the two peculiarities. One is the instruction level parallelism (*ILP*) that is implemented at the micro-architecture hardware level. The second is about use of some *automatic parallelization* standard/compiler. Obviously, this categorization is from programmer's view point in the sense that the application programmer has not to do 'much' to get benefit from it. The two forms of implicit parallelism are further explored below.

0.3.1 Instruction Level Parallelism (ILP)

ILP is a set of techniques for executing multiple instructions simultaneously within the same CPU core, through keeping different functional units/stages busy for different types/parts of instructions, or providing multiple functional units for the same operation. To elaborate, following are the three kinds of ILP:

(1) **Pipelining/Superpipelining**: For pipelining, an instruction is considered to be consisting of a number of functional stages. While an instruction completes a stage and the respective functional unit becomes free than this functional unit can be used to perform the same stage of another similar instruction with different data. This is much like an assembly line. A pipeline provides for overlapping the execution of multiple instructions. This might provide a throughput near to one instruction per clock cycle. Superpipelining is achieved by dividing each of the pipeline stages that need larger amount of execution time (e.g., the stages concerned with cache/memory access) into a number of stages to obtain fine-grained pipeline stages that require nearly equal amount of execution time. The larger number of stages allows larger number of instructions to be executed in parallel [7]. As of the year 2011, microprocessors commonly have 10 to 35-stage pipelines ([4], pp. 10).

(2) **Superscalarity**: Independent instructions requiring different functional units can be issued/executed simultaneously through dynamic instruction scheduling by the hardware at run-time (for example, simultaneous execution of add and multiply instructions, and load instructions on different functional units). This facilitates for increasing the CPU throughput by executing more than one instruction per CPU cycle ([4], pp. 13−14). The modern microprocessors further enhance the superscalarity by incorporating *out-of-order execution* feature, which dynamically decides the need and possibility of scheduling an instruction in a way that violates the instruction fetch order [7]. As of the year 2011, microprocessors are commonly 2 to 8-way superscalar.

(3) **Vectorization**: Multiple pieces of vector data (like elements of a linear array) are loaded into special registers and the same instruction is performed on all the pieces, simultaneously. This concept is also referred to as *S*IMD (*Single instruction, multiple data*). The vectorization or SIMD feature in modern, so called '*scalar*'

processors is the renaissance of the concept at a relatively smaller level, which was originally used in '*vector*' processors in 70s and 80s. The vector processors virtually 'evaded' from the market since 90s, despite of their nature more closer to computational science. As of the year 2011, microprocessors commonly have vector–registers of size up to 256 bits. Thus, two registers each having eight elements of an operand array of 32-bit integers can be operated on simultaneously (performing 8 operations in parallel) to produce the result that is stored into a third vector register. The instruction set architectures of a modern microprocessors include extensions for SIMD operations. Example of these extension include *SSE* and *AVX* from Intel, *3dNow!* from AMD and *VIS* from Sun/Oracle ([8], pp. 8).

Note that the degree of ILP heavily depends on how the program instructions depends on each other. Clearly, more the independent instructions, more would be the chances for ILP. A *dependence* affects the way the program components (statements/instructions, loop iterations, etc.) may be executed ignoring the sequence of events specified by the programmer without changing the output. Program components that are not dependent on each other can be executed in parallel and have greater probability to qualify for ILP. Common types of dependencies include *data dependence*, *name dependence* and *control dependence*. The *data* dependence or *true* dependence refers to the case when a variable content updated by an instruction is used by another instruction following it (*Read-After-Write* case). The name dependence could be either an *anti dependence*, or *output dependence*. Anti dependence refers to the case when a variable content is used by an instruction and then the variable content is updated in another instruction following it (*Write-After-Read* case). Output dependence refers to the case when a variable content is updated by an instruction and then the variable content is updated by another instruction following it (*Write-After-Write* case). The true dependences can not be eliminated, as it is necessary for the algorithm. A name dependence, on the other hand, can removed by variable/register *renaming* technique (see [3], pp. 71). Control dependence affects the *flow of control* in a program from instruction to instruction through the existence of *branches* (conditions like if/else, switch), function *calls*, etc. Modern processors also include *branch predictor* and *speculative execution* feature to allow for more parallelism beyond the limitation of flow of control [7]. Dependencies in

loops, specially the *loop-carried* dependency (where the ith iteration of the loop requires some value updated in (i-1)th iteration), more seriously affect the ILP and, in general, the implicit parallelism. The modern compilers might automatically assume some code restructuring, while generating the object code, for enhancing the degree of ILP. The compilers perform this automatic code optimization according to the level of "privilege" given to the compiler by the programmer.

0.3.2 Automatic Parallelization

This form of implicit parallelism could be achieved by using features available in a parallelization standard/compiler. *Fully automatic parallelization* has not achieved the level of maturity that a wide class of problems could be benefited from it. So this is not focused here. However, *programmer directed* form of automatic parallelization has been experienced to be useful for a large number of cases. An important peculiarity in this regards is the shared memory multithreading that can be achieved by using "only" the compiler directives of a standard like OpenMP and Cilk Plus (supported by the respective compiler) or by using a parallel language like UPC, Chapel, Fortress, X10, Co-Array Fortran, HPF, ZPL, Charm++ including recent contributions, e.g., PetaBricks and Julia. Efforts to develop automatic parallelization models is ever ongoing, like for example, SPC^3PM [9].

Note that the underlying platform should itself need to be a parallel computer for getting benefits of the parallelization models. It should be a personal computer having multicore and/or multithreaded CPUs, at least, if not more sophisticated one. The automatic parallelization might be attributed to implicit parallelism because the respective model or standard takes care of how the parallelism is actually achieved and the programmer has not to do much, expect for using compiler directives to point out the parallelizable code segments. Code segments that avoid strict dependencies and involve patterns favourable to the automatic parallelization have more chances of getting automatically parallelized.

0.4 Explicit Parallelism

Explicit parallelism is characterized by the fact that the programmer is responsible for, (1) taking care of subdividing the problem into a number of sub-problems

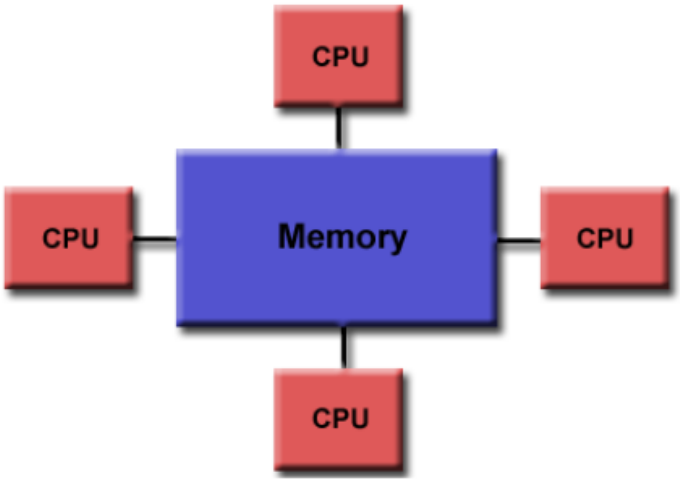
(with respect to data or tasks) for simultaneous execution on a number of processing elements, and (2) managing the coordination and synchronization among the sub-problems. The major forms of explicit parallelism are explained below.

0.4.1 Shared Memory Parallelism

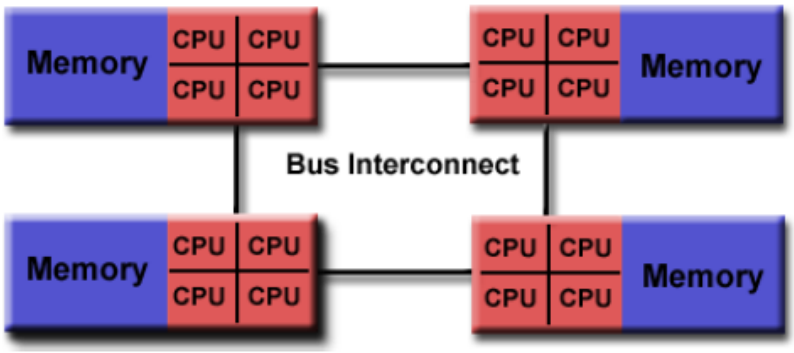
From a hardware perspective, a shared memory parallel architecture is a computer that has a common physical memory accessible to a number of physical processors. The two types of shared memory architectures are Uniform Memory Access (UMA) and Non-Uniform Memory Access (NUMA), as shown in Fig. (4a-b), taken from Barney [5]. Today the most common form of an UMA architecture is the ***Symmetric Multiprocessor (SMP)*** machine, which consists of multiple identical processors with equal level of access and access time to the shared memory. While the most common form of a NUMA architecture is the machine made by inter-linking a number of SMPs. It is characterized by the fact that the access time to different memory locations might vary for a processor.

From a programmer's perspective, the most common form of shared memory parallelism is the ***multithreading programming model***. The parallel application might involve multiple execution *threads* that share a common logical address space. Standard implementations of threads include *POSIX Threads*, *Intel Threading Building Blocks*, *Cilk Plus* and *OpenMP*. Note that, unlike OpenMP, POSIX threads are library based and parallelization with POSIX Threads is explicitly performed by the programmer. The major advantages of shared memory programming are its simplicity and uniformity because of common global address space. On the other hand, due to the same reasons, the shared memory systems are less scalable; data traffic congestion occur when higher number of processors share the same path to access the global memory. Moreover, the cost of building shared memory system with ever increasing number of processors grows exponentially.

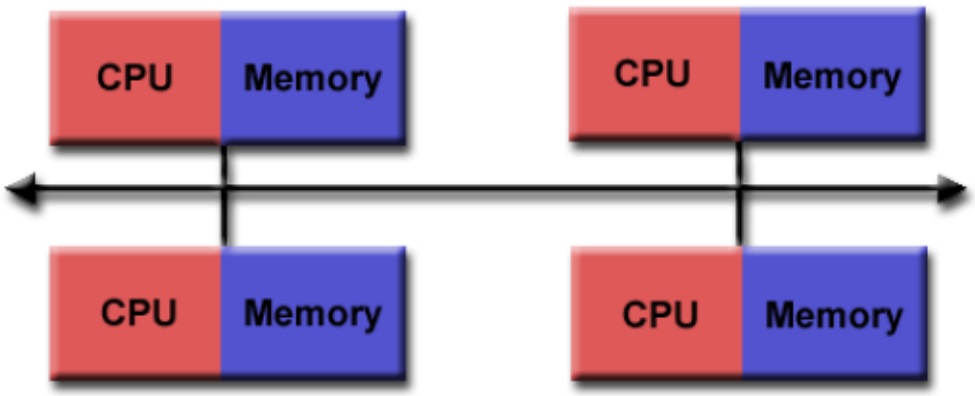
Emergence and so rapid advancements in *multicore* (now also referred to as *many-core*) CPUs have given substantial acceptance of these as a new shared memory parallel computing platform. These CPUs offer a parallel computing platform at *personal computer* (PC) and Laptop levels. As of the year 2011, PCs with 1-4 CPU



(a) Shared memory architecture UMA



(b) Shared memory architecture NUMA



(c) Distributed memory architecture

Figure 4: The basic forms parallel computer architectures. (Figures taken from Barney)

sockets, with each CPU chip having up to 16-cores are available in the commodity market, thus up to a total of 64 CPU cores could be available in a single system. Each of the multicore CPUs, itself, may be regarded as a lower-cost version of UMA-SMP and, thus, the multi-socket PC (having more than one CPU chips, each with its own integrated memory controller) is a NUMA architecture. However, getting respective parallel performance from such machines depends strongly on the algorithm and the program design. Infact, multicore CPUs are appearing to be the one on which the biggest supercomputing machines are relying by considering them as building blocks. However, in such machines sufficiently fast memory hierarchies, interconnect fabrics and I/O systems would be necessary for acceptable parallel efficiencies.

Recently a more advanced and extremely fast, but under-developing and tricky, way of computing is realized that make use of *graphical processing units* (*GPUs*) for explicit parallel computations. With a GPU installed in a computer 10 time faster speed can be achieved, at least in theory, as of the year 2011. To make use of GPUs, currently *CUDA* (for GPUs manufactured by nVidia) and *OpenCL* are the two programming models for *general-purpose graphical processing unit*s (*GPGPU*) computing.

0.4.2 Distributed Memory Parallelism

From a hardware perspective, a distributed memory parallel architecture is a computer that has a number physical processors each with its own local resources and separate memory space and requiring an *interconnection network* for mutual communication for accessing memory of other processors. A basic block diagram of this architecture is shown in Fig. (4c), taken from [5].

From a programming perspective, the most suitable form of distributed shared memory parallelism is ***multiprocessing***. Multiple processes, each allocated with a subproblem, are mapped to different processors (cores) to solve their respective subproblems. The inter-process communication (for mutual coordination and synchronization) is performed using the ***message passing model***. According to which the processes (each having a separate logical memory space) send and receive message for data sharing. This is done in a *cooperative* fashion such that any message SEND call issued by a process must has a matching message RECEIVE call issued by

the process that is supposed to receive the message. A *message passing implementation* is a library (of subroutines for a variety communication operations) that work in conjunction with the usual C/C++/Fortran compilers. Although there have been a number of library-implementations of message passing model, but today ***Message Passing Interface (MPI)*** library is the most widely used implementation. MPI library includes a variety of routines for both point-to-point and collective communications. The communication instance involving one sender and one receiver is referred to as *point-to-point communication*. This is in contrast to the *collective communication* which could be *one-to-all*, *all-to-one*, or *all-to-all*. MPI has emerged as a de-facto standard for portable and scalable parallel programming for distributed memory parallel architectures. Several free and commercial MPI implementations are available. These implementations include both general and architecture/vendor specific. Well-known examples of MPI implementations include

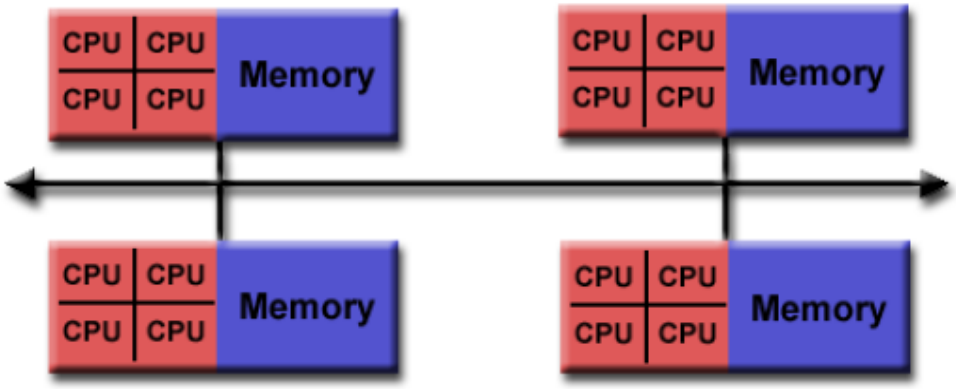
- **OpenMPI** (by Indiana University, open source)
- **MPICH** and **MPICH2** (by Argonne National Lab, open source)
- **MVAPICH** and **MVAPICH2** (by Ohio State University, free)
- **Platform MPI** (by Platform Computing, commercial)
- **Intel MPI** (by Intel, commercial)
- **MSMPI** (by Microsoft, commercial, for use on MS HPC Server 2008 OS)
- **MPJ Express** (by several including NUST−Pakistan, Java based [10]).

The advantages and disadvantages associated with distributed memory parallelism are in contrast to those of shared memory parallelism. Distributed memory systems are highly scalable and are less costly. Even, the mass market commodity off-the-shelf (M^2COTS) computer components can be used to build a cost effective system, usually called ***clusters***. On the other hand, programming for the distributed systems is more challenging and intensive to take care of splitting of data structures across the separate memory spaces of the parallel processes and for cooperative inter-process communication. In principle, the distributed memory MPI applications can execute on any shared memory architecture, as well. For example, 24 MPI processes can be executed in parallel on a 24-core machine (1 MPI process per processing core). Even for the case of shared memory space, each of the MPI process consider its memory space isolated from that of any other process.

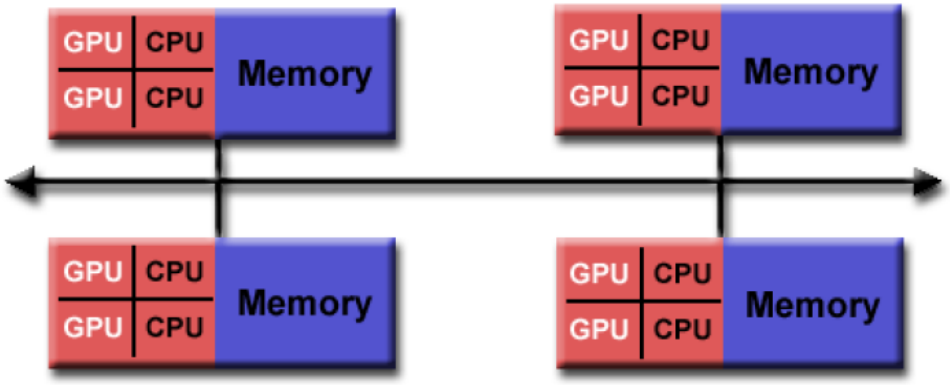
0.4.3 Hybrid Distributed-Shared Memory Parallelism

Hybrid parallel architecture refers to the system consisting of a number of machines/PCs with distributed memory interconnected via a network, where each of the machine is a shared memory computer (like SMP) itself, as shown in Fig. (5a), taken from [5]. Thus, a hybrid distributed-shared memory computer is built by interconnecting a number of SMP machines via a network. This looks practicable and easy to understand as, today, a multi-socket machine with each socket having a multicore CPU is an SMP machine. With the advent of GPGPU computing, a new layer of computing might be added in the hybrid parallel scenario. That is, each of multiple SMP machines is also equipped with one or more GPUs, as shown in Fig. (5b), taken from [5]. The high end supercomputing cluster computers follow a hybrid memory architecture and seem to be prevailing at the top positions in the predictable future.

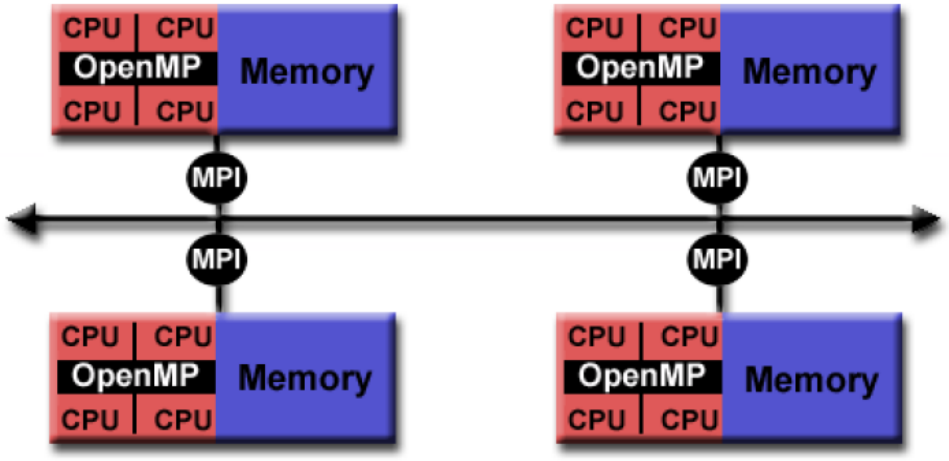
There exists a number of useful hybrid parallel programming paradigms that could be investigated for numerical simulations. MPI with OpenMP is the one among those. The idea is to use MPI for communication across the interconnected machines and use OpenMP within a single multicore machine, as shown in Fig. (5c), taken from [5]. In general, a hybrid programming approach might use more than one implementation, selecting at most one implementation out of each of the three architectural levels of parallelism: (1) MPI, (2) OpenMP/p-threads and (3) CUDA/OpenCL. A hybrid programming approach on an appropriate parallel architecture might (or might not) utilize the computational resource more efficiently and effectively than with any one of the participating parallel programming approach.



(a) Hybrid distributed-shared memory architecture



(b) GPU equiped hybrid distributed-shared memory architecture



(c) A hybrid programming model with MPI + OpenMP

Figure 5: Hybrid parallelism. (Figures taken from Barney)

0.5 Cluster Computing

As discussed that, a cluster is composed by interconnection of individual PCs using a interconnection network (simply called interconnect) such that the interconnected machines acts like a single computer. Common cluster types (with respect to their usage) include *compute* clusters, *high-availability* clusters and *load-balancing* clusters. As the present work is concerned solely with the compute clusters, so the word cluster is used to refer to the compute cluster only. The compute cluster is proven to be very cost effective distributed memory parallel architecture. More precisely, today, the cluster is a hybrid memory parallel architecture given that each of the interconnected machines is an SMP (being a multi-socket and/or multicore machine). The interconnected machines that constitute the cluster are called as ***nodes***. Two necessary types of cluster nodes are ***compute nodes***, where the parallel computations are performed, and ***head node***, which is responsible for system management, handling of user login and job submissions.

In a typical scenario, cluster users log-on to the front-end node of the cluster via public Internet and compile their programs using compilers/environments of their choice (from among the available ones) on the front-end node and submit their *batch jobs* for execution. The front-end node manages some *queues* of user requests and schedules the submitted jobs to execute them on the compute node/s according to some pre-specified priority mechanisms using a job scheduling software, also called *batch scheduler*. Examples of job scheduling software include

- Portable Batch System (PBS)
- Oracle/Sun Grid Engine (SGE)
- Platform LSF (Load Sharing Facility)
- MAUI Cluster Scheduler
- MOAB Cluster Suite
- TORQUE Resource Manager
- Simple Linux Utility for Resource Management (SLURM).

In complement to the batch job submission style, a cluster may have the facility of *interactive job* submission style, usually on some specified nodes for debugging, testing and short jobs. A cluster may also have some scalable distributed *monitoring sys-*

tem for statuses of resources in the cluster. Examples of monitoring software include Ganglia, MOAB Cluster Suite etc. For big clusters, with several hundreds/thousands of nodes, the responsibilities of the head node are distributed among a different type of nodes:

- one **administration/management** node
- one or more **Login** nodes
- one or more **Gateway** or I/O nodes (for efficient handling of data bulks).

The cluster interconnect could be as simple as **Fast Ethernet** (of 100Mbps of bandwidth) and **Gigabit Ethernet** (of 1000Mbps of bandwidth). The cluster may also be equipped with a specialized interconnect technology (of high bandwidth and/or low latency), like **10-Gigabit Ethernet**, **Infiniband**, **Myrinet** or **Quadrics**. If a specialized network is available in the cluster then it is configured to be used for inter-process MPI communications among the parallel processes mapped onto different compute nodes. In such a case, the Ethernet network is used for system/job management purposes. The two major characteristics of an interconnect are *bandwidth* and *latency*. Bandwidth refers to data transfer rate i.e., quantity of data transferred per unit of time, usually expressed in *Mbps* (Mega bits per second). Latency refers to transfer time for minimal (zero byte) data between two points, usually expressed in *microseconds*.

The clusters, as the hybrid memory architectures, offer the most cost effective solution to fulfill the need of high performance parallel computing capabilities for numerical simulations. Because of their effectiveness, cost-effectiveness and scalability, 410 (82%) supercomputing machines out of the world's top 500 known supercomputing machines as of November 2011 are clusters, as announced by the TOP500 project [11]. However, it is worth to mention that clusters compliment rather than compete with the more sophisticated parallel computing architectures, usually called ***Massively Parallel Processing* (*MPP*)** machines, which are only 89 (17.8%) in number out of the total of top 500 machines, as of November 2011. As an evidence of this fact, note that the total number of CPU cores in 410 clusters is 5804063 and that of in 89 MPPs is 3378812. Similarly the overall total maximum speed attained by 411 clusters is 50192818 GFLOPS and that of by 89 MPPs is 23823974 GFLOPS.

This means that, on average, each cluster has 14156 CPU cores and 122421 GFLOPs maximum speed, whereas an MPP has 37964 CPU cores and 267685 GFLOPs maximum speed. Thus, on the TOP500 list [11], an MPP has 2.68 times more number of CPU cores and 2.18 times more speed than those of a cluster, on average.

Moreover, the dominant choices of the CPUs are the 64-bit architecture (x86-64) based **Xeon** (from Intel) and **Opteron** (from AMD) for the high performance computing machines. This is evident from the fact that around 90% of all the CPUs in the top 500 machines belong to these two series of CPUs [11].

As the use of clusters is becoming so economical and widespread that very small clusters within a single desktop chassis are also appearing in the commodity market, for example the Limulus (LInux MULti-core Unified Supercomputer) project [12], that can work like very big clusters (obviously at small scale). Interestingly, a low-cost cluster according to the Limulus Project Design (having four nodes with four quadcore processors of desktop category) has been demonstrated to achieve 200 GFLOPS [13]. Such a 16-core cluster could be outperform a 16-core SMP machine that have all of its cores on the same system board (irrespective of the number of sockets), especially for the scientific applications (like sparse linear solvers, for example) requiring high memory bandwidth [14–16].

0.6 Grid and Cloud Computing

Parallel applications, based on distributed memory models, can be categorized as either *loosely coupled*, or *tightly coupled* applications. Loosely couple applications, sometimes also referred to as *embarrassingly parallel* applications, require very few or virtually no communication among the parallel processes. Therefore, these processes might be mapped to geographically dispersed processing units, inter-connected using some specified networking technology. Such an agglomeration of remote computers inter-connected, possibly on the Internet, with certain others attributes as well, is termed as *grid*. The embarrassingly parallel applications like Monte-Carlo simulations are the best candidate for exploiting the maximum processing potential of grids. The tightly coupled applications, like the PDEs solvers, which require very frequent inter-process communications can be executed more efficiently on the parallel computers which have all of its processing units either inter-connected using some

local network topology (e.g., clusters, MPPs, etc.), or remain within a box (like SMP machines).

Grids not only provide for the sharing of remote 'computational resources' but they enable certain other types of resource sharing as well. Further details on *grid computing* can be found in [17] and [18]. Note that another type of computing, namely the *cloud computing* is also emerging, in which not only the high performance computing but mostly a vast variety of other forms of computing are provided through the Internet as "*services*", instead of "products" [19]. Interestingly the services on *clouds* are provided as metered facilities. This has given rise to the debate whether an in-house HPC facility is preferable or the HPC service from a commercial cloud for medium scale applications. The examples of HPC services on clouds range from a single core computer to a cluster of a few hundred CPU cores, to a cluster of GPUs [20].

0.7 Developing Efficient Parallel Programs

The operating system (OS) of choice for most of the HPC community is **Linux**. 91.4% of the top 500 supercomputing machines are based on Linux [11]. The parallel computers are equipped with a number of modern compilers of C, C++ and FORTRAN languages, at least. Intel, PGI, Pathscale, Absoft and GNU are some well-known compiler providers. For shared memory programming, modern compilers include OpenMP implementation. Also the POSIX thread library could be used. *Cilk Plus* is a new standard, included with C/C++ compilers from Intel, for directive based parallelization. For distributed memory programming an MPI implementation is installed on the parallel computer using a set of C,C++ and FORTRAN compilers. A variety of performance tools and libraries can also be considered. Intel's *Math Kernel Library* (MKL) and AMD's *Core Math Library* are among the libraries that provide highly efficient subroutines for useful mathematical operation.

During the program development, an appropriate **code debugging software tool** might be used to find out the errors to obtain the correct final results from the computations. The list of well-known *debuggers* includes

- **TotalView** (by TotalView Tech, commercial, parallel, memory debugger)

- **Distributed Debugging Tool-DDT** (by Allinea, commercial, parallel)
- **PGDBG** (by Portland, commercial, parallel)
- **GDB** (by GNU, free, serial, Command-Line)
- **Inspector** (by Intel, commercial, parallel, also a memory debugger)
- **Memcheck** (by Valgrind, free, memory debugger).

Once a working parallel code has been developed, it should be considered to tune up for enhancing the overall performance and optimal resource utilization [21]. Tuning refers to finding out *hot spots* in the program (where it consumes large portions of any resource it requires, especially its total execution time) and removing the concerning *bottlenecks* (which use the computing resource inefficiently) [5]. It is interesting that the *90/10 locality rule* often works, according to which a program consumes 90% of the total number of required CPU cycles for 10% of its code [3]. Analysis of code performance and resource utilization could be done through some appropriate **performance profiling/tracing software tools**. A healthy list of well-known parallel performance analysis tools includes

Profiling Tools:

- **mpiP** (by Lawrence Livermore National Lab, free)
- **PGPROF** (by Portland Group, commercial)
- **Vtune Amplifier** (by Intel, commercial)
- **PAPI** (by Uni. of Tennessee at Knoxville, free, hardware performance counter)
- **SCALSCA** (by several including Julich Supercomputing Center, free, a very sophisticated analysis tool)

Tracing Tools (more expert than profiling):

- **Open|SpeedShop** (by Krell Institute, free)
- **TAU** (Tuning and Analysis Utilities, by University of Oregon, free)
- **VampirTracer** (by TU Dresden, free)
- **Trace Collector and Analyzer** (by Intel, commercial)
- **MPE** (MPI Parallel Environment, by Argonne National Lab, free)
- **KOJAK** (by several including Julich Supercomputing Center, free).

The parallelization approach of choice for the present work is MPI based parallel processing. Therefore, the discussion of parallel performance would focus around

the distributed/hybrid memory architectures. In general, performance of a parallel program in a distributed/hybrid memory with a given data size depends on many factors including:

- **Processor** (total number, number per node, speed)
- **Memory** (capacity, bandwidth, latency, caching effects)
- **Interconnect** (type, bandwidth, latency)
- **Environment Capabilities** (OS, compilers, implementation, library)
- **Algorithm, Data Structures and Memory Access Pattern**
- **Granularity** (i.e., computation to communication ratio) [5].

The factors effecting the overall program performance are interrelated and quite complex. Therefore, to meet the challenge of developing efficient parallel programs that make optimal use of the available resources, certain design consideration are important to be taken into account. These consideration include problem and program understanding, problem decomposition (data/task decomposition), I/O and communication requirements and patterns, load balancing, granularity, cost and limits of parallelization. Some detailed discussions on these consideration are presented by Hager and Wellein [4], and Barney [5]. Two important considerations for efficient solutions are discussed below.

0.7.1 Proficient Domain/Task Decomposition

For a good parallel performance the balancing of workload and communication efficiency among the parallel processes are two quite necessary objectives. Domain decomposition techniques are usually based on the graph partitioning algorithms. Some comprehensive discussions on domain decomposition techniques and strategies are presented by Hendrickson and Kolda ([22], 2000), Schloegel et al. ([23], 2003), Magoules ([24], 2007) and Seal and Alurue ([25], 2008). The well known software packages for domain decomposition include METIS [26], Chaco [27], SCOTCH [28] and JOSTLE [29].

0.7.2 Exploitation of Locality of Reference

It is well recognized that the number of CPU clock cycles required for a typical main memory access is much larger (sometimes more than 30 times larger) than the

number of CPU clock cycle required for any *floating point arithmetic operation* (even for the square root and transcendental function evaluation to some extent) [30] [31]. An elegant approach for developing efficient programs is to write the source code so as to make efficient utilization of the *multilevel cache memory system* (commonly available in the modern CPUs). Multi-level cache systems in modern CPUs provide for exploitation of the phenomenon of *locality of reference*. This also necessitates the programmer to understand the memory system (i.e., when the data retain in the caches) and memory access pattern of the program. Thus, the programmer can restructure the code to enhance the locality of reference so that the code attempts for caching the data in the way that a very large number of data accesses are satisfied from the caches and a very less number of data access are needed to be satisfied from the main memory [31]. Quoting two examples of the ways that enhance the locality of reference: (1) all the loops on the multidimensional arrays should be traversed such that the order of accessing the array elements matches with the order of the storage in the memory. Recall that Fortran stores the array elements with column-major order whereas C/C++/Java store the array element with row-major order. (2) By keeping lesser number of arrays (and data sizes in general) in use, it reduces the *cache foot-print*, as well. The cache foot-print of a code segment at a certain moment refers to the amount of working space it requires at that moment during the execution of the code segment. Clearly, more the number of arrays or data structures in use, larger would be the cache foot-print. Smaller cache foot-print sizes have greater probability of getting fitted into the cache and speed-up the execution.

0.7.3 Efficient Inter-Process Communication

A serious overhead in a parallel solution on clusters is related to the latency of communication network. Network latency for communications among processes may seriously harm the efficiency and scalability of the parallel programs [32]. Therefore, while developing a parallel program special care should be taken to minimize the overheads related to inter-process communications over the network. Following are a number of strategies in this regard discussed in [33]. A description of a similar set of strategies has been explained by Hager and Wellein ([4], pp. 244-250).

1. Communication-Efficient Domain Partitioning

2. Single Dispatch of Message to the Receiver

3. Maximization of Local Computations

4. Overlapping of the Communications with Computation

0.8 Parallel Performance Metrics

A number of metrics are available in literature and commonly used to quantify performance of parallel programs. These metrics include "*total execution time*", "*relative speedup*" and "*relative efficiency*". In this work, the "relative speedup" and "relative efficiency" will simply be called "speedup" and "efficiency", respectively. Execution time consists of computation and communication time, both. It is "the elapsed wall clock time from the start of execution of first process of a parallel program to the end of execution of its last process". Simply knowing the execution time of any code or code segments could be done through a variety of timer functions available in the language and implementation. Relative **speedup**, $\S$, of a parallel program is "the ratio of elapsed time, τ_1, taken by one process to solve a problem to the elapsed time, τ_n, taken by n processes" to solve the same problem, i.e.,

$$\S = \frac{\tau_1}{\tau_n}. \tag{0.1}$$

The relative **efficiency**, $\mathcal{E}$, is defined as,

$$\mathcal{E} = \frac{\S}{n}. \tag{0.2}$$

In general, speedup is observed less than n and efficiency is observed between 0 and 1. In an ideal case,

$$\tau_n = \frac{\tau_1}{n}, \quad \S = n, \quad \text{and} \quad \mathcal{E} = 1. \tag{0.3}$$

Sometimes so called "*super-linear speedup*" is observed where speedup is greater than n. This phenomenon is caused by the cache efficiency with smaller data sizes on the n processors as compare to the single processor case. ***Scalability*** is another characteristic of parallel programs that measure how much efficiency is sustained when the processing resources and the problem size are both increased in proportion to each other ([34], pp. 208−218). Some relatively more rigorous theoretical discussions of the performance metrics, also the *Amdahl's law*, *Gustafson-Barsis' law*,

Karp-Flatt metric, *isoefficiency metric* and *refined performance models* can be found at ([4], pp. 123−130) and ([35], pp. 161−173). Based on different approach and applicable to possibly a different situation, each of these models might help to provide the indication of performance extent of a given parallel application. Study of performance models and experiments indicate that for a given problem, the overall speedup increases with increase in number of processing elements until an extent of number of processing elements (relative to the given problem size) is reached. Further increase in the number of processing elements bring the point of *diminishing returns*. To gain further speedup the problem size would need to be enlarged. The scalability analysis performed indicates that how the performance metric, speedup, of the parallel program varies with the increase in number of processes for a given problem size.

Bibliography

[1] H. J. Bungartz, M. Mehl, and C. Zenger, "Computer science and numerical fluid mechanics - An essential cooperation," in *Notes on Numerical Fluid Mechanics* (E. Hirschel and E. Krause, eds.), vol. 100 of *Notes on Numerical Fluid Mechanics and Multidisciplinary Design*, pp. 437–450, Springer-Verlag Berlin / Heidelberg, 2009.

[2] R. Löhner, *Applied CFD Techniques: An Introduction Based on Finite Element Methods*. John Wiley and Sons, Inc., 2nd ed., 2008.

[3] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 4th ed., 2006.

[4] G. Hager and G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, 2010.

[5] B. Barney, *Introduction To Parallel Computing*. Livermore Computing [Online]. `https://computing.llnl.gov/tutorials/parallel_comp/` [Accessed: 29 July 2011].

[6] L. Tierney, "Implicit and explicit parallel computing in R," in *COMPSTAT 2008* (P. Brito, ed.), pp. 43–51, Physica-Verlag HD, 2008.

[7] J.-L. Gaudiot, J.-Y. Kang, and W. W. Ro, "Techniques to Improve Performance Beyond Pipelining: Superpipelining, Superscalar, and VLIW," *Advances in Computers*, vol. 63, pp. 1–34, 2005.

[8] G. Hager and G. Wellein, "Modern processors," in *Introduction to High Performance Computing for Scientists and Engineers*, pp. 1–36, CRC Press, 2010.

[9] M. A. Ismail, S. H. Mirza, and T. Altaf, "A parallel and concurrent implementation of Lin-Kernighan heuristic (LKH-2) for solving traveling salesman problem for multi-core processors using SPC3 programming model," *International Journal of Computer Scientce and Applications*, vol. 2, no. 7, pp. 34–43, 2011.

[10] "MPJ Express (an open source Java message passing library)." `http://mpj-express.org/` [Accessed: 30 November 2011].

[11] "TOP500 Project." `https://www.top500.org/` [Accessed: 25 November 2011].

[12] "The Limulus Project." `http://limulus.basement-supercomputing.com/` [Accessed: 25 December 2011].

[13] "The Nexlink Limulus Cluster." `http://limulus.basement-supercomputing.com/wiki/CommercialLimulus/` [Accessed: 23 December 2011].

[14] "The Limulus Project FAQs." `http://limulus.basement-supercomputing.com/wiki/LimulusFAQ` [Accessed: 23 December 2011].

[15] D. Eadline, "Benchmarking a multi-core processor for HPC," in *The Cluster Monkey Project*, July 12, 2011. `http://www.clustermonkey.net//content/view/306/1/` [Accessed: 23 December 2011].

[16] D. Eadline, "Exercising multi-core," in *Linux Magazine*, September 08, 2010. `http://www.linux-mag.com/id/7855/` [Accessed: 23 December 2011].

[17] B. Wilkinson, *Grid Computing: Techniques and Applications*. CRC Press, 2010.

[18] M. Creel and W. L. Goffe, "Multi-core CPUs, clusters, and grid computing: A tutorial," *Computational Economics*, vol. 32, pp. 353–382, 2008.

[19] J. Rhoton, *Cloud Computing Explained*. Recursive Press, 2nd ed., 2011.

[20] "Amazon Elastic Compute Cloud (Amazon EC2)." `http://aws.amazon.com/ec2/` [Accessed: 25 November 2011].

[21] G. Hager and G. Wellein, "Optimization techniques for modern high performance computers," in *Computational Many-Particle Physics* (H. Fehske, R. Schneider, and A. WeiBe, eds.), vol. 739 of *Lecture Notes in Physics*, pp. 731–767, Springer-Verlag Berlin / Heidelberg, 2008.

[22] B. Hendrickson and T. G. Kolda, "Graph partitioning models for parallel computing," *Parallel Computing*, vol. 26, no. 12, p. 1519Ű1534, 2000.

[23] K. Schloegel, G. Karypis, and V. Kumar, "Graph partitioning for high-performance scientific simulations," in *Sourcebook of Parallel Computing* (J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, and A. White, eds.), pp. 491–541, Elsevier Science, USA, 2003.

[24] F. Magoules(Editor), *Mesh Partitioning Techniques and Domain Decomposition Techniques*. Civil-Comp Ltd. Stirling, UK, 3rd ed., 2007.

[25] S. Seal and S. Alurue, "Spatial domain decomposition methods in parallel scientific computing," in *Handbook of Parallel Computing - Models, Algorithms and Applications* (S. Rajasekaran and J. Reif, eds.), pp. 44/1–24, Chapman and Hall/CRC, USA, 2008.

[26] "METIS – Family of Graph and Hypergraph Partitioning Software." `http://glaros.dtc.umn.edu/gkhome/views/metis` [Accessed: 25 November 2011].

[27] "Chaco: Software for Partitioning Graphs." `http://www.sandia.gov/~bahendr/chaco.html` [Accessed: 25 November 2011].

[28] "SCOTCH – Software Package for Graph Partitioning." `http://www.labri.u-bordeaux.fr/perso/pelegrin/scotch/` [Accessed: 25 November 2011].

[29] "JOSTLE – graph partitioning software." `http://staffweb.cms.gre.ac.uk/~c.walshaw/jostle/` [Accessed: 25 November 2011].

[30] C. Pancratov, J. M. Kurzer, K. A. Shaw, and M. L. Trawick, "Why computer architecture matters," *IEEE-CISE-1521-9615/08*, 2008.

[31] C. Pancratov, J. M. Kurzer, K. A. Shaw, and M. L. Trawick, "Why computer architecture matters: Memory access," *IEEE-CISE-1521-9615/08*, 2008.

[32] W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith, "Toward realistic performance bounds for implicit CFD codes," in *Proceedings of Parallel CFD'99* (D. E. Keyes, A. Ecer, J. Periaux, N. Satofuka, and P. Fox, eds.), pp. 233–240, Elsevier, 1999.

[33] A. Ali, H. Luo, A. Hassan, K. S. Syed, and M. Ishaq, "On parallel performance of a discontinuous Galerkin compressible flow solver based on different numerical fluxes," *AIAA-2011-51*, 2011.

[34] A. Grama, G. Karypis, V. Kumar, and A. Gupta, *Introduction to Parallel Computing*. Pearson/Addison-Wesley, 2nd ed., 2003.

[35] M. J. Quinn, "Performance analysis," in *Parallel Programming in C with MPI and OpenMP*, pp. 159–177, McGraw-Hill, 2003.