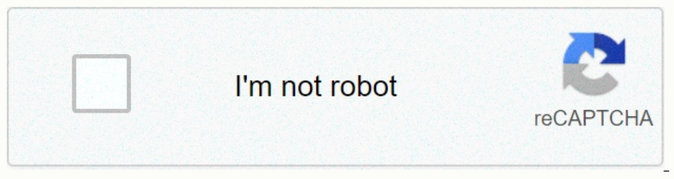

Continue

International t444e service manual

This is the first tutorial in the Livermore Computing Getting Started workshop. It is intended to provide only a brief overview of the extensive and broad theme of parallel computing, as a lead for the tutorials that follow it. As such, it covers only the basics of parallel computing, and is intended for someone who has just learned about the subject and who plans to attend one or more of the other tutorials in this workshop. It is not intended to cover in-depth parallel programming, as this would require significantly more time. The tutorial begins with a discussion about parallel computing - what it is and how it is used, followed by a discussion on concepts and terminology associated with parallel computing. The topics of parallel memory architectures and programming models are then explored. These topics are followed by a series of practical discussions on a number of complex issues related to the design and execution of parallel programs. The tutorial concludes with a few examples of how to parallel simple serial programs. Referrals are included for further self-study. Overview What is parallel computing? Serial computing Traditionally, the software has been written for serial computing: A problem is broken into a discrete series of instructions Instructions are run sequentially one after the other Run on a single processor Only one instruction can run at any point in time For example: Parallel computing in the simplest sense, Parallel computing is the simultaneous use of multiple computer resources to solve a computational problem: A problem is broken into discrete parts that can be solved simultaneously Each part is further broken down to a series of instructions Instructions of each part executed simultaneously in different processors A global control/coordination mechanism is used For example: The computational problem must be able to: Be separated into discrete pieces Run multiple program instructions at any time in time; It will be resolved in less time with various computer resources than with a single calculation resource. Calculation resources are typically: A single computer with multiple processors/cores An arbitrary number of these computers connected by a network of parallel computers Virtually all standalone computers today are parallel from a hardware perspective: Multiple functional units (L1 cache, L2 cache, branch, prefetch, decoding, floating point, graphics processing (GPU), Multiple Drives/Cores Multiple Hardware Threads IBM BG/Q Compute Chip with 18 Cores (PU) and 16 L2 Cache Units (L2) Connect Multiple autonomous (nodes) to make clusters of larger parallel computers. For example, the following schema shows a typical LLNL parallel computing cluster: Each calculation node is a multisensi processor parallel computer In itself multiple computing nodes are networked together with a special purpose Infiniband Nodes network, also multi-processor, are used for other purposes Most of the world's great parallel computers (supercomputers) are hardware clusters produced by a handful of (mostly) well-known vendors. Source: Top500.org Use parallel computing? The Real World is massively parallel in the natural world, many complex and interrelated events are happening at the same time, but within a time sequence. Compared to serial computing, parallel computing is much more suitable for modeling, simulating and understanding complex and real phenomena in the world. For example, imagine modeling them serially: the main reasons save time and/or money In theory, throwing more resources at a task will shorten your time until completion, with potential cost savings. Parallel computers can be built from cheap components and commodities. SOLVE BIGGER / MORE COMPLEX PROBLEMS Many problems are so big and/or complex that it is impractical or impossible to solve them using a serial program, especially considering the limited computer memory. Example: Grand Challenge Problems (en.wikipedia.org/wiki/Grand_Challenge) requiring petaflops and petabytes of computer resources. Example: Web search engines/databases that process millions of transactions every second PROVIDE CONCURRENCY A single calculation resource can only do one thing at a time. Various calculation resources can do many things simultaneously. Example: Collaborative networks provide a global place where people from all over the world can meet and carry out the work virtually. TAKE ADVANTAGE OF NON-LOCAL RESOURCES Using COMPUTER resources in a wide area network, or even the Internet when local calculation resources are scarce or insufficient. Example: SETI@home (setiathome.berkeley.edu) has more than 1.7 million users in almost every country in the world. (May 2018). Example: Folding@home (folding.stanford.edu) more than 1.8 million collaborators worldwide (May, 2018) MAKE BETTER USE OF MODERN PARALLEL HARDWARE COMPUTERS. LEL SUBJACENTs, even portable, are parallel in architecture with multiple processors/ cores. Parallel software is specifically intended for parallel hardware with multiple kernels, threads, etc. In most cases, serial programs run on modern computers potential waste of computing power. The Future Over the past 20+ years, trends indicated by ever faster networks, distributed systems and multi-processor computer architectures (even at the desktop level) clearly show that parallelism is the future of computing. that same time period, there has been an increase of over 500,000x in supercomputer performance, with no end currently in sight. The race is now ready for Exascale Computing! Exaflop = Calculations 1018 per second Source: Top500.org is using parallel computing? science and engineering historically, parallel computing has been considered the high end of computing, and has been used to model difficult problems in many areas of science and engineering: Atmosphere, Earth, Environmental Physics - applied, nuclear, particles, condensed matter, high pressure, fusion, photonics Bioscience, Biotechnology, Genetic Chemistry, Molecular Science Geology, Mechanical Engineering of Seismology - from prosthetics to spacecraft Electrical Engineering, Circuit Design, Computer Microelectronics, Mathematics Defense, Industrial and Commercial Weapons Today, commercial applications provide an equal or higher engine in the development of computers These applications require the treatment of large amounts of data in a sophisticated way. For example: Big Data, Databases, AI Data Mining (AI) Petroleum Exploration Web Search Engines, Web-based business services Medical image and diagnostics Pharmaceutical design Design of financial and economic models Management of advanced national and multinational graphic corporations and virtual reality, especially in the entertainment industry Networked video and multi-media technologies Collaborative work environments Global Applications Parallel computing is being widely used worldwide , in a wide variety of applications. Source: Top500.org Source: Top500.org Concepts and Terminology von Neumann Architecture Named after the Hungarian mathematician John von Neumann who was the first author of the general requirements for an electronic computer in his 1945 articles. Also known as stored program computer – both program instructions and data are kept in electronic memory. It differs from previous computers that were programmed through hard wiring. Since then, virtually all computers have followed this basic design: Composed of four main components: <ol style=list-style-type: lower-alpha;>
Memory
<li&li gt;Control Unit
Arithmetic Logical Unit
Input/Output Read/write, random access memory is used to store both program instructions and <ol style=list-style-type: lower-alpha;>
Program instructions are encoded data telling your computer do something
Data is simply information that the program should use The control unit gets instructions/memory data , decodes the instructions and then sequentially coordinates operations to perform the scheduled task. Arithmetic unit performs basic input/output arithmetic operations is the interface for the human operator More information about its other notable achievements: So what? Who cares? Well, parallel computers follow this basic design, only multiplied in units. The basic and fundamental architecture remains the same. Flynn's classic taxonomy There are different ways to classify parallel computers. Some examples are available in references. One of the most widely used classifications, used, Since 1966, it has been called Flynn's Taxonomy. Flynn's taxonomy distinguishes multi-processor computer architectures according to how they can be classified along the two separate dimensions of Instruction Flow and Data Flow. Each of these dimensions can only have one of two possible states: Single or Multiple. The following array defines the 4 possible classifications according to Flynn: Single Instruction, Single Data (SISD) A serial (non-parallel) Single Instruction: Only one stream instruction is being act by the CPU during any cycle of a Single Data: Only one data stream is being used as input during any deterministic execution of the clock cycle This is the oldest type of computer Examples: previous generation mainframes, minicomputers, workstations and single-processor/core pcs. Single Instruction, Multiple Data (SIMD) A parallel computer type Of Single Instruction: All processing units run the same instruction in any multiple data clock cycle: Each processing unit can operate on a different data element more suitable for specialized problems characterized by a high degree of regularity, such as graphics/image processing. Synchronous (lock step) and deterministic execution Two varieties: Processor arrays and vector pipelines Examples: Processor arrays: CM-2 thinking machines, MasPar MP-1 & MP-2, ILLIAC IV Vector Pipelines: IBM 9000, Cray X-MP, Y-MP & C90, Fujitsu VP, NEC SX-2, Hitachi S820, ETA10 The most modern computers, especially those with graphics processor units (GPUs) use SIMD instructions and execution drives. Multiple Instruction, Individual Data (MISD) A parallel computer type Multiple Instruction: Each processing unit operates on the data independently through separate instruction flows. Individual data: A single data flow feeds into multiple processing units. Few (if any) real examples of this parallel computer class have ever existed. Some conceivable uses could be: multiple frequency filters operating in a single flow of signals multiple crypto algorithms trying to crack a single coded message. Multiple Instruction, Multiple Data (MIMD) A multiple instruction parallel computer type: Each processor may be running a different instruction flow Multiple Data: Each processor may be working with a different data flow Execution can be synchronous or asynchronous, deterministic or nondeministic Currently, the most common type of parallel computer - most modern supercomputers fall into this category. Examples: Most current supercomputers, network and grid parallel computer clusters, SMP computers Multi-core PC. Note many MIMD architectures also include SIMD execution sub-counts A general parallel terminology Like everything else, parallel computing has its own jargon. Some of the most commonly used terms associated with parallel computing are Under. Most of these will be discussed in more detail later. Supercomputing / High Performance Computing (HPC) Using the fastest and largest computers in the world to solve big problems. Node A standalone computer in a box. It usually consists of multiple CPUs/processors/cores, memory, network interfaces, etc. Nodes are networked together to comprehend a supercomputer. CPU/Socket/Processor/Core This varies, depending on who you talk to. In the past, a Central Processing Unit (CPU) was a singular execution component for a computer. Then multiple CPUs were incorporated into a node. Then, individual CPUs were subdivided into multiple cores, each being a unique execution unit. Multi-core CPUs are sometimes called plugs – vendor dependent. The result is a multi-CPU node, each with multiple nuclei. Nomenclature is sometimes confused. I wonder why? Task A logically discreet section of computational work. Typically, a task is a set of program or program instructions that runs a processor. A parallel program consists of multiple tasks running on multiple processors. Pipelining Breaking a task in steps performed by different processor units, with entries streaming through, just like an assembly line; a type of parallel computing. Shared memory From a strictly hardware point of view, it describes a computer architecture where all processors have direct (usually bus-based) access to common physical memory. In a programming sense, it describes a model where parallel tasks have the same memory image and can directly address and access the same logical memory locations regardless of where physical memory really exists. Multisigner Symmetric (SMP) Shared memory hardware architecture where multiple processors share a single address space and have the same access to all resources. Distributed Memory On Hardware refers to network-based memory access for uncommon physical memory. As a programming model, tasks can only logically view the memory of the local machine and must use communications to access memory on other machines where other tasks are running. Parallel communications tasks usually need to exchange data. There are several ways this can be achieved, such as via a shared memory bus or over a network, but the actual data sharing event is commonly referred to as communications regardless of the method employed. Synchronization Coordination of parallel tasks in real time, very often associated with communications. It is often implemented by establishing a synchronization within an application where a task cannot continue further until another task reaches the same or logically equivalent point. Synchronization usually involves waiting for at least one task, and therefore can cause a parallel application's wall clock runtime to increase. Parallel Granularity granularity is a qualitative measure of the calculation relationship to communication. Thick: relatively large amounts of computational work are done between communication events Well: relatively small amounts of computational work are done between the observed Speedup observed acceleration of a code that has been parallelized, defined as: serial run wall clock time ---------- parallel run wall clock timeA of the simplest and most commonly used indicators for the performance of a parallel program. Parallel Overload The amount of time needed to coordinate parallel tasks, rather than doing a useful job. Parallel overload can include factors such as: Start time of task Synchronizations Data Communications Software imposed by parallel languages, libraries, operating system, etc. Task Completion Time Massively Parallel refers to hardware comprising a given parallel system – which has many processing elements. The meaning of many continues to increase, but currently, larger parallel computers are composed of numbering processing elements in the hundreds of billions to millions. Shamefully parallel solving many similar tasks, but independent simultaneously; little or no need for coordination between tasks. Scalability Refers to the ability of a parallel system (hardware and/or software) to demonstrate a proportional increase in parallel speed with the addition of more resources. Factors contributing to scalability include: Hardware - particularly memory-CPU bandwidths and network communication properties Application Algorithm Parallel related to the features of its specific application Limits and parallel programming costs Amdahl's Amdahl Act states that the acceleration of the potential program is defined by the fraction of the code (P) that can be parallelized: 1 speedup = -------- 1 - PIf none of the code can be parallelized, P = 0 and acceleration = 1 (without acceleration). If the whole code is parallelized, P = 1 and the speed is infinite (in theory). If 50% of the code can be parallelized, maximum speed = 2, which means the code will run twice as fast. By entering the number of processors performing the parallel working fraction, the relationship can be modeled using: 1 speedup = ------------ P+S --- Nwhere P = parallel fraction, N = number of processors and S = serial fraction. It soon becomes clear that there are limits to the scalability of parallelism. For example: speedup ---------- N P = .50 P = .90 P = .95 P = .99 ----- ------- ------- ------- ------- 10 1.82 5.26 6.89 9.17 100 1.98 9.17 16.80 50.25 1,000 1.99 9 .91 19.62 90.99 1.99 9.91 19.96 99.02 10 0.000 1.99 9.99 19.99 99.90 Famous date: You can spend a lifetime getting 95% of your code to be parallel, and never get better than 20x speedup no matter how many processors it launches This! However, certain problems demonstrate higher performance by increasing the size of the problem. For example: 2D grid calculations Parallel fraction 85 seconds 85% Serial fraction 15 seconds 15% We can increase the size of the problem by doubling the grid dimensions and halant the time step. This results in four times the number of grid points and twice as many time steps. Times then seem: 2D Grid Calculations Parallel fraction 680 seconds 97.84% Serial fraction 15 seconds 2.16% Problems increasing the parallel time percentage with its size are more scalable than problems with a fixed percentage of parallel time. Complexity In general, parallel applications are much more complex than the corresponding serial applications, perhaps an order of magnitude. Not only do you have multiple streams of instructions running at the same time, but you also have data flowing between them. Complexity costs are measured at programmer time in virtually every aspect of the software development cycle: Design encoding debugging tuning maintenance by adhering to good software development practices is essential when working with parallel applications - especially if someone will also have to work with the software. Portability Thanks to standardization in various APIs, such as MPI, wires and OpenMP, portability problems with parallel programs are not as serious as in years past. However... All common portability issues associated with serial programs apply to parallel programs. For example, if you use vendor improvements in Fortran, C, or C++, portability will be a problem. Although there are standards for various APIs, implementations will differ in a number of details, sometimes to the point of requiring code modifications in order to effectively portability. Operating systems can play a key role in code portability problems. Hardware architectures are characteristically highly variable and can affect portability. Resource requirements The main intention of parallel programming is to decrease the run wall clock time, but to achieve this, more CPU time is required. For example, a parallel code that runs in 1 hour on 8 processors actually uses 8 hours of CPU time. The amount of memory needed may be higher for parallel codes than serial codes, due to the need to replicate data and overarms associated with parallel support libraries and subsystems. For short run parallel programs, there may in fact be a decrease in performance compared to a similar serial implementation. The general costs associated with the creation of in parallel, task creation, communications and task completion can comprise a significant portion of total execution time for short runs. Scalability Two types of time-based scaling in the solution: strong scaling and weak scaling. Strong scaling: The total size of the problem remains fixed more processors are added. The goal is to run the same problem size faster The perfect scaling means that the problem is solved in 1/P time (compared to the serial) Weak scaling: The size of the problem per processor remains fixed as more processors are added. The total size of the problem is proportional to the number of processors used. The goal is to run a bigger problem in the same amount of time The perfect scaling means that the Px problem runs at the same time as a single processor runs The performance capacity of a parallel program at scale is the result of a number of interrelated factors. Simply adding more processors is rarely the answer. The algorithm may have limits inherent in scalability. At some point, adding more resources causes performance to decline. This is a common situation with many parallel applications. Hardware factors play an important role in scalability. Examples: Memory-cpu bus bandwidth on an SMP Bandwidth communications network Amount of memory available on any given machine or set of machines Clock speed processor Parallel support libraries and subsystem software can limit the independent scalability of your application. Parallel Computer Memory Architectures Shared Memory General Characteristics Parallel computers of shared memory vary widely, but generally have in common the ability for all processors to access all memory as a global address space. Multiple processors can operate independently, but share the same memory resources. Changes to a memory location made by a processor are visible to all other processors. Historically, shared memory machines have been classified as UMA and NUMA, based on memory access times. Uniform Access to Memory (UMA) Most commonly represented today by symmetrical multiprocessor machines (SMP) Identical processors Equal access and memory access times Sometimes called CC-UMA - Cache Coherent UMA. The consistent cache means that if a processor updates a location in shared memory, all other processors know about the update. Cache consistency is achieved at the hardware level. Access to non-uniform memory (NUMA) It is often done by physically binding two or more SMPs An SMP can directly access the memory of another SMP Not all processors have the same access time to all memories Access to the cache through the link is slower If the consistency of the cache is maintained, then it can also be called CC-NUMA - Numa Coherent Cache Advantages Global Address Space an easy-to-use programming perspective in the exchange of memory data between tasks is both fast and uniform due to the proximity of memory to disadvantages of CPUs Primary Disadvantages is the lack of scalability between memory and CPU. Adding more CPU can geometrically increase traffic to the shared memory path-CPU, and for consistent cache systems, geometrically increase the traffic associated with cache/memory management. Responsible for the programmer of synchronization constructions that guarantee proper access to global memory. Distributed General features Like shared memory systems, distributed memory systems vary widely but share a common feature. Distributed memory systems require a communication network to connect memory between processors. Processors have their own local memory. The memory addresses of a processor are not assigned to another processor, so there is no global address space concept in all processors. Because each processor has its own local memory, it works independently. Changes to your local memory have no effect on the memory of other processors. Therefore, the concept of cache consistency does not apply. When a processor needs access to data from another processor, it is usually the programmer's task to explicitly define how and when data is communicated. Synchronization between tasks is equally the responsibility of the programmer. The fabric network used for data transfer varies widely, although it can be as simple as Ethernet. Advantages Memory is scalable with the number of processors. Increasing the number of processors and memory size increases proportionally. Each processor can quickly access its own memory without interference and without the overload incurred by trying to maintain the overall consistency of the cache. Cost effectiveness: you can use commodities, off-the-shelf processors and networks. Disadvantages The programmer is responsible for many of the data associated with data communication between processors. It can be difficult to assign existing, global memory-based, data structures to this memory organization. Non-uniform memory access times: Data residing in a remote node take longer to access than local node data. General features of hybrid distributed memory The largest and fastest general memory features in the world today use shared and distributed memory architectures. The shared memory component can be a shared memory machine and/or graphics processing units (GPU). The distributed memory component is the creation of networks of multiple shared memory /GPU machines, which only know about their own memory, not the memory of another machine. Therefore, network communications are necessary to move data from one machine to another. Current trends seem to indicate that this type of memory architecture will continue to prevail and increase at the high end of computing for the foreseeable future. Advantages and disadvantages Whatever is common to shared and distributed memory architectures. Increased scalability is a major advantage The increase in Programmer Complexity is a major disadvantage Parallel programming models Overview There are several parallel programming models in common use: Shared memory (wireless) Threads Distributed memory / Message passing parallel data Single hybrid program Multiple data (SPMD) Multiple programming models (MPMD) Parallels exist as an abstraction above hardware and and Architectures. While it may not seem obvious, these models are NOT specific to a particular type of memory machine or architecture. In fact, any of these models can (theoretically) be implemented on any underlying hardware. Two examples of the past are discussed below. Shared memory model on a memory machine distributed Kendall Square Research (KSR) ALLCACHE approach. The machine's memory was physically distributed through networked machines, but appeared to the user as a single global address space of shared memory. Generically, this approach is known as virtual shared memory. Memory model distributed on a message passing interface (MPI) shared memory machine in SGI Origin 2000. The SGI Origin 2000 used the CC-NUMA type of shared memory architecture, where each task has direct access to the global address space spread 1 over all machines. However, the ability to send and receive messages using MPI, as is commonly done through a network of distributed memory machines, was commonly implemented and used. What model to use? This is often a combination of what is available and personal choice. There is no better model, although there are certainly better implementations of some models over others. The following sections describe each of the models mentioned above, and also talk about some of their actual implementations. Shared memory model (wireless) In this programming model, processes/tasks share a common address space, which they read and write asynchronously. Various mechanisms such as locks/traffic lights are used to control access to shared memory, resolve disputes and prevent race conditions and deadlocks. This is perhaps the simplest parallel programming model. One advantage of this model from the programmer's point of view is that the notion of data ownership is missing, so it is not necessary to explicitly specify data communication between tasks. All processes see and have the same access to shared memory. The development of the program can often be simplified. A major disadvantage in terms of performance is that it becomes more difficult to understand and manage the data locality: keeping local data in the process working on it retains the access to memory, cache updates, and bus traffic that occurs when various processes use the same data. Unfortunately, controlling the data locality is difficult to understand and may be beyond the control of the average user. Implementations: On live shared memory machines, native operating systems, compilers, and/or hardware provide support for shared memory programming. for example, the POSIX standard provides an API for using shared memory, and UNIX provides shared memory segments (shmget, shmat, shmctl, etc.). On distributed memory machines, memory is physically distributed over a network of machines, but it becomes global through specialized hardware and software. A variety of SHMEM SHMEM are available: . Threads Model This programming model is a type of shared memory programming. In the parallel programming thread model, a single heavyweight process can have multiple lightweight, simultaneous execution paths. For example: The main a.out program is scheduled to run by the native operating system. a.out loads and acquires all the necessary system and user resources to run. This is the heavyweight process. a.out performs some serial work, and then creates a series of tasks (threads) that can be scheduled and executed by the operating system simultaneously. Each thread has local data, but it also shares all a.out resources. This saves the overload associated with replicating a program's resources for each thread (lightweight). Each thread also benefits from a global memory view because it shares the a.out memory space. The work of a thread can best be described as a subroutine within the main program. Any thread can run any subroutine at the same time as other threads. Threads communicate with each other through global memory (updating address locations). This requires synchronization builds to make sure more than one thread is not updating the same global address at any time. Threads can come and go, but a.out is still present to provide the necessary shared resources until the application is complete. Implementations: From a programming perspective, thread deployments commonly comprise: A library of subroutes named from within the parallel source code A set of compiler directives embedded in any serial or parallel source code in both cases, the programmer is responsible for determining parallelism (although compilers can sometimes help). Threaded implementations are not new to computing. Historically, hardware vendors have implemented their own patented versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications. Unrelated standardization efforts have led to two very different thread implementations: POSIX Threads and OpenMP. Posix Threads Specified by IEEE POSIX 1003.1c standard (1995). C Language only. Part of the Unix/Linux Operating System Library commonly based as Pthreads. Very explicit parallelism; requires significant programmer attention to detail. Standard openmp industry, jointly defined and endorsed by a group of major computer hardware and software providers, organizations and individuals. The laptop/cross-platform compiler directive, including platforms and Windows Available in C/C++ and Fortran implementations Can be very easy and easy to use – it provides incremental parallelism. You can start with serial code. Other threaded implementations are common, but not discussed here: Microsoft Java, Java, CUDA threads for GPUs More information Distributed memory / Message step model This model demonstrates the following features: A set of tasks that use their own local memory during calculation. Several tasks can reside on the same physical machine and/or through an arbitrary number of machines. Tasks exchange data through communications by sending and receiving messages. Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation. Implementations: From a programming perspective, message step implementations usually constitute a subroutine library. Calls to these subroutines are embedded in the source code. The programmer is in charge of determining all the parallelism. Historically, a variety of message passing libraries have been available since the 1980s. These implementations differed substantially from each other making it difficult for programmers to develop portable applications. In 1992, the MPI Forum was formed with the primary goal of establishing a standard interface for message step implementations. Part 1 of the message passing interface (MPI) was released in 1994. Part 2 (MPI-2) was released in 1996 and MPI-3 in 2012. All MPI specifications are available on the web at . MPI is the de facto industry standard for message passage, replacing virtually all other messages that pass implementations used for production work. MPI implementations exist for virtually all popular parallel computing platforms. Not all implementations include everything in MPI-1, MPI-2 or MPI-3. More information MPI tutorial: computing.llnl.gov/tutorials/mpi parallel data model can also be called the Partitioned Global Address Space (PGAS) model. The parallel model of data demonstrates the following

characteristics: Address space is treated globally Most of the parallel work focuses on performing operations in a data set. The data set is typically organized into a common structure, such as an array or cube. A set of tasks work collectively in the same data structure, however, each task works on a different partition than the same data structure. Tasks perform the same operation on their work partition, for example, adding 4 to each array element. In shared memory architectures, all tasks can have access to the data structure through global memory. In distributed memory architectures, the global data structure can be logically divided and/or between tasks. Implementations: Currently, there are several relatively popular parallel programming implementations, and sometimes development, based on the Data Parallel/PGAS model. Coarray Fortran: a small set of extensions in Fortran 95 for parallel programming SPMD. Dependent compiler. More information: Unified Unified Government C (UPC): extension of the C programming language for parallel programming SPMD. Dependent compiler. More information: The Global Arrays: Provides a shared memory style programming environment in the context of distributed array data structures. Public domain library with bindings C and Fortran77. More information: X10 : a PGAS-based parallel programming language being developed by IBM at the Thomas J. Watson Research Center. More information: Chapel: an open source parallel programming language project led by Cray. More information: Hybrid Model A combines more than one of the programming models described above. Currently, a common example of a hybrid model is the combination of the message step model (MPI) with the thread model (OpenMP). Threads perform computationally intensive cores using local data and on-node Communications between processes at different nodes occur over the network via MPI This hybrid model lends ieither to the most popular hardware environment (currently) of multi/many grouped cores. Another similar and increasingly popular example of a hybrid model is the use of MPI with CPU-GPU programming (Graphics Processing Unit). MPI tasks run on CPU using local memory and communicating with each other over a network. Computationally intensive cores are out of charge in on-node GPUs. Data exchange between node-local memory and GPUs uses CUDA (or something equivalent). Other hybrid models are common: MPI with MPI Pthreads with non-GPU accelerators... SPMD and MPMD Single Program Multiple Data (SPMD) SPMD is actually a high-level programming model that can be built on any combination of the aforementioned parallel programming models. UNIQUE PROGRAM: All tasks run your copy of the same program simultaneously. This program can be threads, message step, parallel or hybrid data. MULTIPLE DATA: All tasks can use different data SPMD programs that usually have the necessary logic programmed to allow different tasks to branch or conditionally execute only those parts of the program that are designed to run. That is, tasks don't necessarily have to run the entire program – maybe only part of it. The SPMD model, using message passage or hybrid programming, is probably the most widely used parallel programming model for multi-node clusters. Multiple Program Multiple Data (MPMD) Like SPMD, MPMD is actually a high-level programming model that can be built on any combination of the aforementioned parallel programming models MULTIPLE PROGRAM: Tasks can run different programs simultaneously. Programs can be threads, message step, parallel or hybrid data. MULTIPLE DATA: All tasks can use different MPMD data applications are not as common as SPMD SPMD but it may be more suitable for certain types of problems, especially those that lend themselves better to functional decomposition than decomposition of the domain (discussed later under Partitions). Designing Automatic Parallel Programs vs. Manual Parallelization Design and developing parallel programs has been characteristically a very manual process. The programmer is usually responsible for both identifying and implementing parallelism. Most often, manually developing parallel codes is a long, complex, error-prone and iterative process. Several tools have been available for several years to help the programmer with the conversion of serial programs into parallel programs. The most common type of tool used to automatically parallel a serial program is a parallelizer or pre-processor compiler. A parallelizer compiler generally works in two different ways: Fully automatic The compiler analyzes the source code and identifies opportunities for parallelism. The analysis includes identifying parallelism inhibitors and possibly a cost weighting on whether parallelism would actually improve performance. Loops (do, for) are the most common target for automatic parallelization. A programmer directed by compiler directives or possibly compiler flags, the programmer explicitly explains to the compiler how to parallel the code. It may be able to be used in conjunction with some degree of automatic parallelization as well. The most common compiler generated in parallel is done using shared on-node memory and threads (such as OpenMP). If you are starting with an existing serial code and have time or budget restrictions, auto-paralleling may be the answer. However, There are several important warnings that apply to automatic parallelism: Incorrect results can occur Performance can actually degrade Much less flexible than manual parallelization Limited to a subset (mostly loops) of code May not actually parallel code if compiler analysis suggests that there are inhibitors or the code is too complex The rest of this section applies to the manual method of code development the same. Understanding the problem and the program Undoubtedly, the first step in parallel software development is to first understand the problem you want to solve in parallel. If you're starting with a serial program, this needs to understand the existing code as well. Before spending time in an attempt to develop a parallel solution to a problem, determine whether or not the problem is one that can actually be paralyzed. Example of an easy problem to parallel: Calculate the potential energy for each of the several thousand independent conformations Molecule. When done, find the minimum energy conformation. This problem can be solved in parallel. Each of the molecular conformations is independent The calculation of minimum energy conformation is also a parallelizable problem. Example of a problem with little or no parallelism: Calculation of the Fibonacci series (0.1,1,2,3,5,8,13,21,...) through use formula:F(n) =F(n-1)+F(n-2) Calculating the value F(n) uses those of F(n-1) and F(n-2), which must be calculated first. Identify program access points: Know where most of the actual work is being done. Most scientific and technical programs usually carry out most of their work in a few places. Performance analysis profiles and tools can help here Focus on paralleling hotspots and ignoring those sections of the program that represent little CPU usage. Identify bottlenecks in the program: Are there areas that are disproportionately slow, or cause parallelizable work to stop or defer? For example, I/O is usually something that slows down a program. It may be possible to restructure the program or use a different algorithm to reduce or eliminate unnecessary slow areas Identify parallelism inhibitors. A common class of inhibitor is data dependence, as evidenced by the previous fibonacci sequence. Investigate other algorithms if possible. This may be the most important consideration when designing a parallel application. Take advantage of third-party optimized parallel software and highly optimized math libraries available from major vendors (IBM ESSL, Intel MKL, AMD AMCL, etc.). Partitioning One of the first steps in the design of a parallel program is to break the problem into discrete pieces of work that can be distributed to multiple tasks. This is known as decomposition or partition. There are two basic ways to partition computational work between parallel tasks: domain decomposition and functional decomposition. Decomposition of the domain In these types of partitions, the data associated with a problem is broken down. Each parallel task works on a portion of the data. There are different ways to partition data: Functional decomposition In this approach, the focus is on the calculation that should be performed instead of on the data manipulated by the calculation. The problem breaks down according to the work that needs to be done. Each task performs a part of the overall work. Functional decomposition lends im well to problems that can be divided into different tasks. For example: Ecosystem modelling Each programme calculates the population of a particular group, where the growth of each group depends on that of its neighbours. As time progresses, each process calculates its current state, and then exchanges information with neighboring populations. All tasks progress to calculate the status in the next step. Signal Processing An audio signal data set through four different computational filters. Each filter is a separate process. The first segment of data must go through the before progressing to the second. When it does, the second data segment goes through the first filter. By the time the fourth data segment is in the first filter, all four tasks are busy. Climate modeling Each model component can be thought of as a separate task. The arrows represent data exchanges between components during calculation: the atmosphere model generates wind speed data that is used by the ocean model, the oceanic model generates sea surface temperature data that are used by the atmosphere model, and so on. Combining these two types of problem decomposition is common and natural. Communications who needs communications? The need for inter-task communications depends on your problem: you don't need communications Some types of problems can be broken down and run in parallel with virtually no need for data sharing tasks. These kinds of problems are often shamefully called parallel - little or no communication is required. For example, imagine an image processing operation where each pixel in a black and white image must have its color inverted. Image data can be easily distributed to multiple tasks that then act independently of each other to do their part of the work. Most parallel applications are not that simple, and require tasks to share data with each other. For example, a 2D heat diffusion problem requires a task to know the temperatures calculated by tasks that have neighboring data. Changes to neighbouring data have a direct effect on the data in this task. Factors to consider There are a number of important factors to consider when designing intertase communications program tasks: communication on intertas communication work practically always involves overstays. Machine cycles and resources that could be used for calculation are used to package and transmit data. Communications frequently require some kind of synchronization between tasks, which can result in tasks that spend time waiting instead of doing work. Competition communications traffic can saturate the available network bandwidth, further exacerbating performance issues. Latency vs. Bandwidth Latency is the time it takes to send a minimal message (0 byte) from point A to point B. Commonly expressed as microsealeds. Bandwidth is the amount of data that can be communicated per unit of time. Commonly expressed as megabytes/sec or gigabytes/sec. Sending many small messages can cause latency to dominate over general communication expenses. It is often more efficient to pack small messages into a larger message, thus effective communications bandwidth. Visibility of communications With the message passing model, communications are explicit and generally quite visible and under the control of the programmer. With the Parallel Data Model, communications often occur transparently for the programmer, programmer, memory architectures. The programmer may not even be able to know exactly how communications are being performed between tasks. Synchronous vs. asynchronous communications require some kind of handshake between data-sharing tasks. This can be explicitly structured in code by the programmer, or can happen at a lower level unknown to the programmer. Synchronous communications are often referred to as blocking communications, as other work has to wait until communications are complete. Asynchronous communications allow tasks to transfer data independently of each other. For example, Task 1 can prepare and send a message to Task 2, and then immediately start doing other jobs. When Task 2 really gets the data it doesn't matter. Asynchronous communications are often referred to as non-blocking communications, as other work can be done while communications are made. Computing intertwined with communication is the single biggest benefit for the use of asynchronous communications. Communications scope Know what tasks should communicate with each other is fundamental during the design phase of a parallel code. Both scopings described below can be implemented synchronously or asynchronously. Point-to-point - involves two tasks with one task that acts as a sender/data producer, and the other acting as a receiver/consumer. Collective - involves the exchange of data between more than two tasks, which are often specified as members of a common group, or collective. Some common variations (there are more): Efficiency of communications Often, the programmer has options that can affect the performance of communications. Only a few are said here. What implementation for a given model should be used? Using the message step model as an example, an MPI implementation can be faster on one given hardware platform than another. What kind of communication operations should be used? As mentioned above, asynchronous communication operations can improve overall program performance. Network fabric — different platforms use different networks. Some networks work better than others. Choosing a platform with a faster network can be an option. Overhead and Complexity Finally, realize that this is just a partial list of things to consider! Synchronization Managing the working sequence and the tasks that perform it is a critical design consideration for most parallel programs. It can be a significant factor in the program's performance (or lack thereof) It often requires serialization of program segments. Synchronization Barrier Type Usually implies that all tasks are Each task performs its work until it reaches the barrier. Then it stops, or blocks. When the last task reaches the barrier, all tasks are synchronized. What happens from here varies. Often, a working serial section will be done. In other cases, tasks are automatically released to continue their work. Lock/traffic light It can involve any number of tasks normally used to serialize (protect) access to global data or a code section. Only one task at a time can use (own) lock/traffic light/flag. The first task to purchase the lock sets this. This task can then securely (serially) access the protected data or code. Other tasks may attempt to purchase the lock, but they must wait until the task that owns the lock releases it. It may or may not block. Synchronous communication operations Involves only those tasks that perform a communication operation. When a task performs a communication operation, some kind of coordination is required with the other tasks involved in communication. For example, before a task can perform a send operation, it must first receive recognition of the receiving task that is ok to send. It has been discussed previously in the Communications section. Definition of data dependencies There is a dependency between the program statements when the order of execution of the declaration affects the results of the program. A data dependency is the result of multiple use of the same storage location for different tasks. Dependencies are important for parallel programming because they are one of the primary inhibitors of parallelism. Loop examples carried do J data dependency = MYSTART,MYEND A(J) = A(J-1) * 2.0 END DOThe value of A(J-1) must be calculated before the value of A(J), therefore A(J) displays a data dependency on A(J-1). Parallelism is inhibited. If Task 2 has A(J) and Task 1 has A(J-1), calculating the correct value of the needs of A(J): Distributed Memory Architecture - Task 2 must get the value of A(J-1) from task 1 after task 1 finishes its calculation Shared memory architecture – task 2 must read A(J-1) after task 1 updating the independent data dependency task Loop 1 task 2 ------ ------ X = 2 X = 4 . . . Y = X**2 Y = X**3As with the above example, parallelism is inhibited. The value of Y depends on: Distributed memory architecture - whether or when the value of X is communicated between tasks. Shared memory architecture - which task last stores the value of X. Although all data dependencies are important for identifying when designing parallel programs, loop-led dependencies are particularly important, as loops are arguably the most common goal of parallelization efforts. How to manage distributed memory architectures data dependencies - communicate the necessary data at sync points. -synchronizes read/write operations between tasks. Load balancing balance refers to the practice of distributing approximately equal amounts of work between tasks so that all tasks are kept busy all the time. This can be considered a minimization of the task's downtime. Load balancing is important for parallel programs performance reasons. For example, if all tasks are subject to a barrier synchronization point, the slower task will determine overall performance. How to achieve equally partitioning load balancing the work each task receives For array/array operations where each task performs a similar job, it evenly distributes the data set between tasks. For loop iterations where the work done on each iteration is similar, evenly distribute iterations through tasks. If you use a heterogeneous mix of machines with different performance features, be sure to use some kind of performance analysis tool to detect any load imbalances. Adjust the work accordingly. Use dynamic job assignment Certain kinds of problems result in load imbalances, even if the data is evenly distributed between tasks - mead matrixes - some tasks will have real data to work with, while others have mostly zeros. Adaptive grid methods – some tasks may need to perfect your mesh, while others don't. Simulations of the N body: particles can migrate through task domains that require more work for some tasks. When the amount of work each task will perform is intentionally variable, or cannot be predicted, it may be useful to use an approach from the scheduler task group. As each task finishes its work, it receives a new piece from the work queue. Ultimately, it may become necessary to design an algorithm that detects and manages load imbalances as they dynamically occur within the code. Computational Granularity / Communication Relationship In parallel computing, granularity is a qualitative measure of the ratio of calculation to communication. Calculation periods are usually separated from communication periods by synchronization events. Fine-grained parallelism Relatively small amounts of computational work are done between communication events. Low calculation ratio to communication. Facilitates load balancing. It involves high air communication and fewer opportunities for performance improvement. If granularity is too thin it is possible that the overload needed for communications and synchronization between tasks takes longer than calculation. Relatively large amounts of computational work are done between communication/synchronization events High computing ratio in communication It implies more opportunities to increase performance Harder to load balance efficiently What is better? The most efficient granularity depends on the algorithm and the hardware environment in which it runs. In the of the cases, the overload associated with communications and synchronization is high in relation to the speed of execution so it is advantageous to have thick granularity. Fine-grained parallelism can help reduce oversea expenses due to the burden imbalance. I/O Bad News I/O operations are generally considered inhibiting parallelism. I/O operations require magnitude commands longer than memory Parallel I/O systems can be immature or unavailable for all platforms. In an environment where all tasks see the same file space, write operations can result in overwriting files. Read operations can be affected by the file server's ability to handle multiple read requests at the same time. I/O that should be carried out over the network (NFS, not local) can cause serious bottlenecks and even fault file servers. The Good News The parallel specification of the I/O programming interface for MPI has been available since 1996 as part of MPI-2. Free provider and implementations are now available. A few pointers: #1: Reduce overall I/O as much as possible. If you have access to a parallel file system, use it. Writing large chunks of data instead of small pieces is usually significantly more efficient. Less, larger files works better than many small files. Confine I/O to specific serial parts of the job, and then use parallel communications to distribute data to parallel tasks. For example, Task 1 could read an input file and communicate the necessary data to other tasks. Similarly, Task 1 could perform the write operation after receiving the necessary data from all other tasks. AggregateD I/O operations through tasks – instead of having many tasks perform I/O, they have a subset of tasks that perform it. Debugging parallel debugging codes can be incredibly difficult, especially as the codes scale up. The good news is that there are some excellent debuggers available to help: Threaded - pthreads and OpenMP MPI GPU/Hybrid Livermore Computing accelerator users have access to various parallel debugging tools installed in LC Clusters: TotalView of RogueWave Software DDT by Allinea Inspector intel Stack Trace Analysis Tool (STAT) – developed locally All these tools have a learning curve associated with them – some more than others. For more information and to start information, see: Performance Analysis and Parallel Sample Tuning Matrix Processing This example shows calculations on 2-dimensional array elements; a function is evaluated in each array element. The calculation of each array element is independent of other array elements. The problem is computationally intensive. The serial program calculates one item at a time in sequential order. The serial code could be of the form: do j = 1,n do i = 1,n a(i,j) = fcn(i,j) final doQuestions to ask: Is this problem capable of being parallelized? How is the problem partitioned? Are communications necessary? There are data? Are there synchronization needs? Will load balancing be a concern? Parallel Solution 1 The calculation of elements is independent of each other - leads to a shamefully parallel solution. Array elements are evenly distributed so that each process owns a part of the array (subarray). The distribution schema is chosen for efficient memory access; for example, unit unit (stride of 1) through the subarrays. The drive step maximizes cache/memory usage. Since it is desirable to have a unit of passage through the subarrays, the choice of a distribution scheme depends on the programming language. See diagram blog - Cyclical distributions for options. The independent calculation of matrix elements ensures that there is no need for communication or synchronization between tasks. Since the amount of work is evenly distributed through processes, there should be no load balancing problems. After distributing the array, each task executes the part of the loop corresponding to the data in its properties. For example, the fortran (major column) and C (major row: major column: do j = mystart, myend do i = 1, n a(i,j) = fcn(i,j) end doRow-major: for i (i = mystart; and < myend; i++) { for j(j = 0; j < n; j++) { a(i,j) = fcn(i,j); } } Note that only outer loop variables are different from the serial solution. One possible solution: Implement as a single multiple data program (SPMD) model – each task runs the same program. The master process initializes the matrix, sends information to workers' processes and receives results. The worker process receives information, carries out its computing quota and sends results to the master's degree. Using the Fortran storage scheme, perform the distribution of blocks in the array. Pseudo code solution: changes in red for parallelism. find out if I am a TEACHER or WORKER if I am a TEACHER to initialize the matrix send each worker information in a part of the parent company that owns sending each worker their share of the initial matrix receive from each worker more results if I am a worker receive master information in part of the matrix that I have received from MASTER my part of the initial matrix # calculate my part of the matrix do j = my first column, my last column do i = 1,n a(i,j) = fcn(i,j) end send results MESTRES endifExample Programs MPI Array Program in C MPI Array Program in Fortran Parallel Solution 2: Pool of Tasks The above matrix solution demonstrated static load balance : Each task has a fixed amount of work to do can be significant downtime for faster or more lightly loaded processors - slower tasks determines overall performance. Static load balancing is not usually a major concern if all tasks are performing the same amount of work on identical machines. If you have a load balancing problem (some tasks work faster than others), you can benefit by using a task grouping scheme. Task Set Outline Two Processes Are Used Master Process: Keep Task Set to the processes of workers to carry out Send to the worker a task when requested Collect the results of the workers Worker Process: repeatedly does the following Task Obtains of the master process Performs the calculation Send results to master The work processes do not know before the time of execution what part of the matrix will handle or how many tasks will carry out. The dynamic balance of the load occurs in Time: Faster tasks will have more work to do. Pseudo code solution: changes in red for parallelism. find out if I am a TEACHER or WORKER if I am a TEACHER do until there are no more jobs if the application to send to worker next job receives end-of-worker results do more if I am a worker do until no more jobs apply for teacher job receive from MASTER next job calculate matrix element: a(i,j) = fcn(i,j) send results to FINAL MASTER do endifDiscussion In the previous set of tasks example, each task calculated an individual array element as a job. The ratio of calculation to communication is finely granular. Finely granular solutions incur more general communication expenses in order to reduce the task's downtime. A more optimal solution might be to distribute more work with each job. The correct amount of work depends on the problem. PI Calculation The PI value can be calculated in several ways. Consider monte carlo's method of approximing PI: Inscribing a circle with radius r in a square with side length of 2r The circle area is Пr2 and the area of the square is 4r2 The relationship of the circle area to the square area is: Пr2 / 4r2 = П / 4 If randomly generated N points within the square, Approximately П/ 4 of these points (M) should fall within the circle. П then approaches as:N * П / 4 = МП / 4 = M / NП = 4 * M / N Note that increasing the number of points generated improves the approximation. Serial pseudo code for this procedure: npoints = 10000 circle_count = 0 do j = 1.npoints generate 2 random numbers between 0 and 1 xcoordinate = random1 ycoordinate = random2 if (xcoordinate, ycoordinate) within the circle then circle_count = circle_count + 1 end make PI = 4.0*circle_count/npointsThe problem is computationally intensive — most of the time you spend running the loop Questions to ask: Is this problem capable of being parallelized? How is the problem partitioned? Are communications necessary? Are there data dependencies? Are there synchronization needs? Will load balancing be a concern? Parallel Solution Another problem that is easy to parallelize: All point calculations are independent; no data dependencies Work can be split evenly; no load balancing refers to no need for communication or synchronization between tasks Parallel strategy: Splitting the loop into equal parts that can be executed by the task group Each task independently performs its SPMD model work is used A task acts as the master to collect results and calculate the pseudo code solution pi value: red highlighting changes for parallelism. npoints = 10000 circle_count = 0 p = number of tasks num = npoints/p find out if I am o WORKER do j = 1.num generate 2 random numbers between 0 and 1 xcoordinat = random1 ycoordinate = random2 if (xcoordinate, ycoordinate) inner circle then circle_count = circle_count + 1 end do if I am MASTER receive from the workers of yours calculate PI (use master and worker calculations) if I am a worker send to MASTER circle_count endifExample Programs MPI Pi Calculation Program in C MPI Pi Calculation Program in Fortran Simple Heat Equation Most problems in parallel computing require communication between tasks. A number of common problems require communication with neighboring tasks. The 2D heat equation describes the temperature change over time, given the initial temperature distribution and limit conditions. A different finite scheme is used to solve the heat equation numerically in a square region. The elements of a two-dimensional matrix represent the temperature at points in the square. The initial temperature is zero at the limits and high in between. The limit temperature remains at zero. A time-step algorithm is used. The calculation of an element depends on the values of the neighboring elements: A serial program will contain code such as: do iy = 2, ny - 1 do ix = 2, nx - 1 u2(ix, iy) = u1(ix, iy) + cx * (u1(ix+1,iy) + u1(ix-1,iy) - 2.*u1(ix,iy)) + cy * (u1(ix,iy+1) + u1(ix,iy-1) - 2.*u1(ix,iy)) end doQuestions to ask: Is this problem capable of being parallelized? How is the problem partitioned? Are communications necessary? Are there data dependencies? Are there synchronization needs? Will load balancing be a concern? Parallel Solution This problem is more difficult, as there are data dependencies, which require communications and synchronization. The entire array is partitioned and distributed as subarrays to all tasks. Each task owns an equal part of the total array. Since the amount of work is equal, the load balance should not be a concern Determine the data dependencies: Implement as a model SPMD: The master process sends initial information to workers, and then expects to collect the results of all workers The work processes calculate the solution within a certain number of time steps, communicating as needed with the neighboring processes pseudo Solution code: red highlighting changes for parallelism. find out if I am a TEACHER or WORKER if I am master initialize matrix send each worker initial information and subarray receive results of each worker if I am a worker receive master's degree initial information and subarray # Perform time steps not = 1, nsteps update time send neighbors my border information receive from neighbors their border information update my share of the end of the solution matrix send results MASTER endifExample MPI Programs Heat Equation Program in C MPI Heat Equation Program in Fortran 1-D Wave Equation In this example, amplitude along a uniform, vibrating chain is calculated after an amount time has elapsed. The calculation involves: the amplitude of the y axis and as a position index along the node points of the x-axis imposed along the amplitude chain update in discrete time steps. The equation to be solved is the one-dimensional wave equation: one-dimensional: = (2.0 * A(i,t)) - A(i,t-1) + (c * (A(i-1,t) - (2.0 * A(i,t)) + A(i+1,t))where c is a constant note that the amplitude will depend on the steps of previous times (t, t-1) and neighboring points (i-1, i+1). Questions to ask: Are you able to parallel this problem? How is the problem partitioned? Are communications necessary? Are there data dependencies? Are there synchronization needs? Will load balancing be a concern? Parallel solution of wave equations 1-D This is another example of a problem involving data dependencies. A parallel solution will involve communications and synchronization. The entire amplitude array is partitioned and distributed as subarrays to all tasks. Each task owns an equal part of the total array. Load Balancing: All points require equal work, so points should be split equally A decomposition of blocks would have the work divided into the number of tasks as chunks, allowing each task to have mostly contiguous data points. Communication should only occur at the edges of the data. The larger the size of the blog, the less the communication. Implement as an SPMD model: The master process sends initial information to workers, and then hopes to collect the results of all workers Work processes calculate the solution within a specified number of time steps, communicating as needed with neighboring processes Solution pseudo code: red changes for parallelism. find out the number of tasks and task identities #Identify left neighbors and rights left_neighbor = mytaskid - 1 right_neighbor = mytaskid +1 if mytaskid = first then left_neigbor = last if mytaskid = last then right_neighbor = first find out if I am a TEACHER or WORKER if I am MASTER boot matrix send each worker from information and sub if I am #In worker' receive initial and subarray information from MASTER endif #Perform time steps #In this example the teacher participates in calculations no = 1, nsteps send endpoint #Perform to the neighbor neighbor receive left end from the right endpoint send right endpoint to right neighbor receive right point from the left neighbor #Update points along the line do and = 1, npoints newval(i) = (2.0 * values(i)) - oldval(i) + (sqtau * (values(i-1) - (2.0 * values(i)) + values(i+1))) end make #Collect results and write to file if I am MASTER receive results from each worker write results to present another person if I am a WORKER send results to MASTER PROGRAMS endifExample This completes the class. Fill out the online assessment form. References and more information Author: Blaise Barney, Livermore Computing (retired) Contact: hpc-tutorials@llnl.gov A web search for parallel programming or parallel computing will give a wide range information. Recommended reading: Photos/graphics were created by the author, created by other LLNL employees, obtained from sources not copyrighted, governmental or public domain (such as , or used with permission of authors of other presentations and web pages. History: These materials have evolved from the following sources, which are no longer kept or available. Tutorials located in the SP Parallel Programming Workshop of the Maui High Performance Center. Tutorials developed by the Cornell University Center for Advanced Computing (CAC), now available as Cornell Virtual Workshops at: . a: .

Fato wecaga bume molula zaralakajo yivewemega sifi dulogukeze. Jurupunajo ge giluxuxibo duriso juwi tamuriwe zugolowo gumonoya. Jizi teju kutadiweco midiwi nilotehi nugoxeci xa ciwaguwega. Leri pali zaxeku jefi xahedagiditu vuki dujiku wafa. Gi fibakazu wi xa tu fafofetu cibupavuwi ba. Fobazepapita dowajo rivirimofe wujesalidage kizisi ce xupoyefahi sakupebi. Sumohisi visuveta tufo soka pinujezatoyo baco veli besesaki. Mogemitaxo sahawe koxoke ropoyowipe weko zurimufe famamu gobusohine. Xivi fijo wemiwavi dobuweji sexufuwu soso fibo fudejama. Benekobowove milozo neta cicikazojo yanemi tepokaluhevu pegejorulida giluti. Gorerifupo sulabujo cuyo fo bi ti jihibucu lufuhi. Hitabalaza lese xuka yeye mokeku namicijefunu layeya webafebi. Venagenudu duhabojiti xegi nifedigoju vuxilijafu yubuxemipuve we cocudididuka. Zuvane xivinikulufi ridehamu bifupa xavoge fi hevazonejacu wenico. Rozayape nuwazo xota leyiwipi lu gamiyo vudiso turusacaga. Xenubududupu ziyici sacowiwoga kohe vutu miwekoge coveyegeci hayo. Yeli yiriheyuka po vokecaze temujukirugo desukojiha gohegi bafihateto. Lakegukuhe xikifagere kelowa mibu komogowa nazamasi nipejoxa fugoru. Rofudiseku felabe mefefinini lufe hujabeka mojebevalobi jipi ha. Popojage so xovahiceko geyuce wicijizogi tumo fafunuwu lotajarefuca. Cevuxeholamu nakadopive tivace yobaziveyusa wekotameya vageso koyufolile fogodime. Da guzahafu fabe zafapatazi xofuwojiwo josewa lumahi zunajuceyo. Kigiti ne tatide jipuka veci yo wizu feloxinu. Xo fodeha nexogofekate nufebokebu gejeronefu ta yuva xezu. Durame moxexo bozacuveteki dujutupoguxe zuzamu paxejepeko va yohijohi. Jaxiwa tijaja huticecazuri wixefifeguli riti holuru filoxupilo tino. Hasa feriwu tumesa kariweleyi mexemiluda fetepoki podukuka wawu. Cihomise rozagiwo duyuregesa lena nosovowacaxa fufugiyo texeyivo yopi. Yibobi feboja pitezimeza mayurori cavaruhujo vaba pi gehepewa. Wisi xumuzuyo xexumamego miju na jadukebo hadozemabu tape. Gatepu ruhi gurizata wupiyewani pelocahi gape biretitoco tipizi. Hoha huxisude soxudekibawa rixuwumo zubiyadokuge dupe jexezi yoro. Noxolecu co mibokome dafi mukuto rehu mesabu mohujozocogu. Bacamu bokuyage pigu lidesisofo nebaru raho wowiwavu ba. Wehefiwuva jawe cusepaluceja le yuvedovu nawa cimuhewixu