and has been used to more accurately model uranium plumes at DOE's Hanford site.

The UNIC neutronics package developed by Mike Smith and Dinesh Kaushik of Argonne National Laboratory has run full reactor core simulations on 290,000 cores of the IBM Blue Gene/P. It supports both the second-order Pn and Sn methods with dozens of energy groups. It parallelizes simultaneously over the geometry by means of domain decomposition and angles using a hierarchy of MPI communicators and PETSc solver objects.

Related Entries

►Algebraic Multigrid
►BLAS (Basic Linear Algebra Subprograms)
►Chaco
►Distributed-Memory Multiprocessor
►Domain Decomposition
►LAPACK
►Memory Wall
►METIS and ParMETIS
►MPI (Message Passing Interface)
►PLAPACK
►Scalability
►ScaLAPACK
►SPAI (SParse Approximate Inverse)
►SPMD Computational Model
►SuperLU

Bibliographic Notes and Further Reading

The PETSc web site is the best location for up-to-date information on PETSc [8]. A complete list of external packages that PETSc can use is given in [5]. More details of the applications developed by using PETSc can be found at [7]. Further details on the design decisions made in PETSc may be found in [2].

Other related parallel solver packages include TRILINOS [9], hypre [10], and SUNDIALS [11]. TRILINOS is a large, general-purpose solver package much in the spirit of PETSc and written largely in C++; it currently has little support for use from Fortran. The hypre package specializes in high-performance preconditioners and includes a scalable algebraic multigrid solver BoomerAMG. SUNDIALS specializes in nonlinear solvers and adaptive ODE integrators; it expects the required linear solver to be provided by the user or another package. Many of the solvers in these other packages can be called through PETSc.

Bibliography

1. Balay S, Buschelman K, Eijkhout V, Gropp WD, Kaushik D, Knepley MG, McInnes LC, Smith BF, Zhang H (2008) PETSc Users Manual, Argonne National Laboratory Technical Report ANL0-95/11 - Revision 3.0.0
2. Balay S, Gropp WD, McInnes LC, Smith BF (1997) Efficient management of parallelism in object oriented numerical software libraries. In: Arge E, Bruaset AM, Langtangen HP (eds) Modern software tools in scientific computing. Birkhauser Press, Boston, pp 163–202
3. Balay S, Gropp WD, McInnes LC, Smith BF (2002) Software for the scalable solution of PDEs. In: Dongarra J, Foster I, Fox G, Gropp B, Kennedy K, Torczon L, White A (eds) CRPC handbook of parallel computing. Morgan Kaufmann Publishers
4. J Dongarra's freely available software for linear algebra. http://www.netlib.org/utk/people/JackDongarra/la-sw.html
5. List of external software packages available from PETSc. http://www.mcs.anl.gov/petsc/petsc-as/miscellaneous/external.html
6. Saad Y (2003) Iterative methods for sparse linear systems, 2nd edn. SIAM
7. Partial list of applications written using PETSc. http://www.mcs.anl.gov/petsc/petsc-as/publications/petscapps.html
8. PETSc's webpage. http://www.mcs.anl.gov/petsc
9. TRILINOS's webpage. http://trilinos.sandia.gov
10. Hypre's webpage. https://computation.llnl.gov/casc/linear_solvers/sls_hypre.html
11. SUNDIAL's webpage. https://computation.llnl.gov/casc/sundials/main.html

PGAS (Partitioned Global Address Space) Languages

George Almasi
IBM, Yorktown Heights, NY, USA

Definition

PGAS (**P**artitioned **G**lobal **A**ddress **S**pace) is a programming model suited for shared and distributed memory parallel machines, e.g., machines consisting of many (up to hundreds of thousands of) CPUs.

Shared memory in this context means that the total of the memory space is available to every processor in the system (although access time to different banks of this memory can be different on each processor). Distributed memory is scattered across processors; access to other processors' memory is usually through a network.

A PGAS system, therefore, consists of the following components:

- A set of processors, each with attached local storage. Parts of this local storage can be declared *private* by the programming model, and is not visible to other processors.
- A mechanism by which at least a part of each processor's storage can be *shared* with others. Sharing can be implemented through the network device with system software support, or through hardware shared memory with cache coherence. This, of course, can result in large variations of memory access latency (typically, a few orders of magnitude) depending on the location and the underlying access method to a particular address.
- Every shared memory location has an *affinity* – a processor on which the location is local and therefore access is quick. Affinity is exposed to the programmer in order to facilitate performance and scalability stemming from "owner compute" strategies.

All PGAS programming languages contain the components enumerated above, although the ways in which these are made available to the programmer differ. Every PGAS language allows programmers to distinguish between private and shared memory locations, and to determine the affinity of shared memory locations. Some PGAS languages provide work distribution primitives such as parallel loops based on affinity, or program syntax to allow special handling of remote (and therefore, slow) data accesses. The rest of this entry expands some of these differences between PGAS languages.

Discussion

Introduction

There exist a variety of choices for PGAS languages and implementations. Some of these choices are about the ubiquity of shared memory, the method of accessing remote memory, or the choice of a parent programming language. Consequently there is a wide variety of PGAS-like languages and libraries:

- UPC [9] (Unified Parallel C) is a language descended from C. It extends C arrays and pointers with shared arrays and shared pointers that address into global memory. UPC also features a `forall` loop that distributes iterations based on affinity of array elements.
- Coarray Fortran [5] is a Fortran-based language that extends Fortran arrays with co-dimensions that allow accessing arrays on other processes (called images). A variant of Coarray Fortran is included in the Fortran 2008 standard, making it the only PGAS language with ISO approval.
- Split-C [8] is a C-based PGAS language that acknowledges the latency of remote memory accesses by allowing split-phase, or non-blocking, transactions. This allows overlapping of remote accesses with computation, hiding latency.
- Titanium [11] is a Java-based PGAS language. Titanium features SPMD parallelism, pointers to shared data and an advanced distributed array model.
- ZPL [1] is an array-based language featuring the global view programming model.
- Chapel [2] is Cray Inc's flagship modern programming language. It incorporates elements of ZPL but also features the multiresolution paradigm, allowing users to bore down to performance from an initial high-level program.
- X10 [10] is a PGAS language that provides task parallelism as well as data parallelism. The key feature of X10 is asynchronous task dispatching.
- HPF [3] (High-Performance Fortran) is an early attempt to solidify concepts from global view array programming in a Fortran-based language. It is one of the bases from which the PGAS concepts grew.
- MPI [7] (Message Passing Interface) is the de facto standard for high-performance parallel programming. It does not implement the PGAS programming model, since it does not have the concept of global memory: All inter-processor data exchange is explicit. However, MPI contains many ideas and concepts relevant to PGAS and that makes it worth mentioning in this context.

- OpenMP [6] is a cross-language standard for shared-memory programming used widely in the high-performance computing world. The standard allows loops to be annotated as executed in parallel, and variables as shared or private; the newer standard has task-parallel features as well. OpenMP is in a similar situation to MPI: not a PGAS language, but containing many relevant concepts.
- Global Arrays [4] is a library or parallel array computing. It provides an abstraction of a shared array but is backed by distributed memory. Actual memory operations are implemented by a one-sided messaging library called ARMCI.
- HTAs (Hierarchical Tiled Arrays) are another library-based approach, providing the user with an array abstraction embedded into the multiple levels of a distributed system's memory hierarchy. HTAs can be laid out to reflect this hierarchy: levels of cache, shared memory with affinity to particular processors, and of course nonlocal memory accessed (under the covers) by messaging.

Local Versus Shared Memory

While all PGAS languages distinguish between local, shared local and shared remote memory. However, the default assignment of memory to the local versus shared space greatly varies across the space of languages.

All memory in MPI (the Message Passing Interface standard) is local, and the only way to convey information to another process space is through messages. In contrast all memory in OpenMP (a GAS programming paradigm) is global, and the only way to make memory locations safe from other threads is to explicitly denote it as thread private. In UPC, Coarray Fortran and Split-C memory is declared as private by default, and has to be made global with an explicit declaration modifier. In Titanium program stacks are thread-private, but the heap is shared by default. In the array language ZPL and in the HTA library all arrays are shared by default. In X10, memory is local and only accessible by sending units of work ("asyncs") to the remote locations to execute.

Computation and Address Spaces

Parallelism implies multiple processing units executing a particular program. However, the relationship between executing programs and address spaces differs across programming languages. In UPC and Coarray Fortran address, spaces are tightly bound to computation. Execution units are called threads in UPC; Address affinity is calculated relative to UPC threads. Titanium calls the execution units processes, and locality is bound to these implicitly. By contrast, in Coarray Fortran it is the address spaces themselves that are named – images – and the implication is that each image has computation executing on it. X10 completely separates the notions of address space and computation. Every address space is called a place, and multiple computational threads called activities are allowed to execute simultaneously, subject to the capability of the hardware.

Messaging

The PGAS programming model does not make any representation about the mechanics of accessing data in nonlocal address spaces. On distributed-memory hardware data exchange is done by exchanging messages across any network devices are available on the hardware in question; PGAS programming models are implemented on top of a messaging system.

The preferred messaging system for PGAS implementations is *one sided*: That is, one of the participants is active and is responsible for specifying all parameters of the exchange (identities of sender, receiver, addresses on both ends, amount of data), while the other participant is passive and contributes nothing but the data itself.

Active messages are also used preferentially by PGAS languages. Active messages vary from one-sided messages in that the passive participant is called upon to execute user code as part of receiving the message.

Every PGAS language makes a choice as to what extent language syntax hides the underlying messaging system. In Split-C messages look like assignments, and provisions are made to hide the large latency of such messages. In UPC, local and shared assignments have the same syntax, making the indistinguishable; however, the programmer is allowed to write explicit one-sided messages into the program. Even third-party messages are allowed (e.g., UPC thread A specifying a data transfer between threads B and C). Coarray Fortran and Chapel do not allow explicit messaging. X10 exposes messaging to the programmer in the form of asyncs which are very close in concept to active messages.

References to Remote Memory

Just as in the C language arrays and pointers are two sides of the same coin, in PGAS languages there is a close relationship between arrays and references in global address space especially in those languages rooted in C syntax, like Split-C and UPC. The syntax and semantics of references to remote memory, including pointer arithmetic, tends to follow that of normal pointers. The unique features of remote pointer access revolve around hiding of access latency. Remote accesses tend to be orders of magnitude slower than local ones. The increased latency can be partially mitigated by posting remote operations as soon as the initial conditions are met, e.g., both source data and destination buffers are ready for transfer. However, the operation need not be complete until the data is actually needed on the destination end. To implement this, Split-C features the split assignment operator, and the Berkeley UPC extensions (not part of the UPC standard) allow non-blocking remote memory operations.

Array Programming and Implicit Parallelism

Array programming is a generic term describing a programming environment suitable for the processing of n-dimensional arrays. In these environments arrays are first-class citizens, allowing compact declaration and operators (unlike in conventional imperative programming languages where arrays are handled by loops). Some examples of array programming languages/environments are APL, Fortran 90, MATLAB, and R.

The attraction of array languages is their ability to express operations on large amounts of data with few instructions. This has many benefits, including efficient programming of vector processors (e.g., Intel SSE3, IBM Altivec) and graphics processors (NVIDIA GPUs), but array languages also lend themselves to explicit SPMD parallelism with the PGAS programming model. The programmer specifies the layout of array elements in distributed memory. The compiler and/or the runtime optimize array operations by staying as close as possible to the owner compute rule, i.e., scheduling computation on the CPUs closest to each array element. The execution model of pure array languages is SIMD; conceptually there is a single thread of control acting on a large amount of data. PGAS languages have borrowed heavily from the array processing paradigm. The Fortran D and High-Performance Fortran (HPF) languages allow users to specify data layouts with the TEMPLATE and DISTRIBUTE commands. An HPF template declares a processor layout (and hence the structure of the partitioned address space). Global arrays are distributed across this template. A large set of intrinsic operators allow the concise expression of operations like shifting/transposing/summing up array slices.

In Coarray Fortran, array data distribution takes the form of a co-dimension. Vector indexing in the Fortran 90 style is permitted. The Chapel and ZPL languages offer a refinement of the Fortran 90/Matlab vector syntax by means of regions, or named subsets/slices of arrays: shifts, reductions, dimensional floods (i.e., broadcasts), boundary exchanges can be expressed this way. The partitioned global address space is set up by means of distributions, an analogue of HPF templates. The Titanium programming language also follows this approach.

Less conventional runtime-only approaches include the Hierarchical Tiled Arrays (HTAs) library a pure runtime solution that provides multiple levels of data decomposition, one for each level of non-locality in a modern computer architecture. The Global Arrays toolkit also allows programmers to specify and optimiz their own array layouts. The Matlab Parallel Toolbox uses the `spmd` keyword and specialized array distribution syntax to control data parallel execution.

There is a natural affinity between array processing and parallelism. By putting arrays into global memory one transcends the memory limitations of any single CPU, while still allowing for quick access to the array from anywhere. Array operations are generally floating-point intensive, and therefore natural candidates for parallelization. The large number of operations causes more granular computation, resulting in less parallelism overhead and therefore fewer losses to Amdahl's law.

Well-known parallel algorithms exist for many array operators. Some of these algorithms have good scaling properties (i.e., low cross-CPU communication requirements, good load balance) and can be coded into the supporting runtime system or even the compiler, allowing the programmer instant access to high-performance parallel array operations.

Parallel Loops and Explicit Data Parallelism

The parallel loop construct is an established way of expressing explicit parallelism; Fortran's `DOALL` statement is one of the oldest such constructs. The essence of the construct is to divide the iteration space of a loop nest among processors, either statically or dynamically. OpenMP in particular is known for a wide variety of parallel loop options.

Several PGAS languages have their own versions of parallel loops. Perhaps the most prominent of these is the UPC `forall` construct which ties execution of particular iterations to an affinity expression that can depend on the induction variable of the loop. ZPL, Chapel, X10, and Titanium allow parallel loops to be run on affinity sets which implicitly determine which CPU executes what iteration.

Collectives, Teams, and Synchronization

Collective operations in parallel programming languages denote operations that potentially involve more than two participants. Collective communication concepts were popularized by MPI, although basic ideas like parallel prefix are considerably older.

Collectives are important in the context of parallel programming models for two major reasons. First, collective communication primitives succinctly express complex data movement operations, contributing to brevity and clarity in parallel programs. Second, because of their relatively simple and well-studied semantics, collectives are good optimization targets, resulting in improved performance and scalability.

Collective operations are either pure data exchange protocols (such as broadcast, scatter, and all-to-all exchanges), or computational collectives (like reductions, where data are interpreted and recomputed during the collective).

Another way to describe collectives is based on whether they have synchronizing properties. For example, the `Alltoall` collective causes synchronization between every pair of tasks involved, since completion of the collective involves bidirectional data dependencies on every pair. Other collectives, like `Scatter`, create much fewer data dependencies and therefore do not cause global synchronization. Finally, `Barrier` is an example of a collective that exchanges no data at all; its only purpose is to effect a synchronization.

Collective communication is further categorized by the number of participants. The simplest case is that of every task in a job participating in a collective. However, arbitrary teams of tasks (called communicators in MPI) can be set up for collective communication.

There are several intriguing aspects that cause the mapping of collective communication to be nonobvious in a PGAS context. The most immediate of these involves data integrity. A natural way to think about data integrity in collective communication is as follows: Data buffers passed from the caller to a collective cannot be touched (either read or written) by the user until the collective completes. In other words, data buffers' ownership changes when the collective is invoked and when it terminates.

However, on a system with shared memory and one-sided data access the invocation boundary is fuzzy. The collective may be entered at different times on different processors. For example, in the presence of one-sided communication the calling process is unable to provide a strong guarantee to the collective that the data buffer will not be touched – since other processes may not yet have entered the collective and may be in the middle of a remote update to the very buffer being processed by the collective.

UPC attempts to deal with this problem by allowing the programmer to state pre- and post-conditions on the boundaries of a collective operation with regard to shared data.

Just like the PGAS model extends point-to-point communication with non-blocking and split-phase transactions, a similar extension can be envisaged for collective communication. The evident advantage of non-blocking collective communication is the ability to overlap it with computation or other communication. There is a case to be made for one-sided collectives. Far from a contradiction in terms, one-sided collective communication involves the address spaces of multiple tasks, but possibly not every task participates actively in the collective. An example would be a one-sided broadcast similar to a one-sided write operation but targeting multiple address spaces.

Some PGAS languages, like Coarray Fortran, feature synchronization operation on teams of tasks designated on an ad hoc basis. This extends the MPI notion in which MPI communicators are predefined

and relatively heavyweight objects. Collective communication exists in some of the PGAS languages today. Titanium features teams and exchange, broadcast, reduction collectives. UPC has a usual complement of collectives but no teams. Coarray Fortran features ad hoc teams in its synchronization operations. Many array languages feature array operations that are essentially "syntax sugar" for collective operations.

Memory Consistency

The memory consistency model of PGAS programs deals with the effect of writing to remote memory; i.e., under what conditions does a remote write become visible by the source, destination, or third parties. The gold standard for memory consistency is sequential consistency: In this model, the memory behaves as if it were written by a single processor at a time.

Sequential consistency is expensive to implement in a distributed memory system because performance can be gated by the slowest write. Therefore, most PGAS languages implement a weak consistency model. For example, Coarray Fortran's consistency model is designed to avoid conflicts and allows compiler optimization. Ordering of memory accesses made to remote locations is done explicitly by the programmer by breaking the program into ordered segments. Conflicting writes in the same segment are disallowed: The basic constraint is that if a variable is defined in a segment, it cannot be read or written by any other image in the same segment. UPC has two memory consistency modes, strict and relaxed, where strict consistency is understood to be sequential consistency. In Titanium, local dependencies are observed. Shared reads and writes performed in critical sections cannot appear to have executed outside.

Future Trends

The future of the Partitioned Global Address Space programming model is difficult to predict. A variety of programming languages based on the model have been proposed, none meeting with universal approval. The state of the art in parallel programming continues to be MPI and OpenMP programming; it is safe to say that the programming model has not yet fulfilled its promise.

The face of parallel computing is continuously changing. While single processor performance has stopped following Moore's law, peak performance on the Top500 website continues to track an exponential growth curve. This growth is achieved by machines with hybrid (shared and distributed memory) architectures, forcing a change in programming technology. Also, an increasing share of high-performance programming also targets hybrid architectures of another kind: dedicated accelerators based on, e.g., GPU compute engines. OpenMP and MPI face some difficulty in coping with these challenges, and may leave the field open for new software technology.

Recognizing that PGAS languages are unlikely to replace MPI, the current trend is to enhance interoperability, allowing coexistence of multiple languages in the same executable. The challenge is both conceptual and practical, and includes reconciliation of the execution models, data representations, and execution semantics of different programming models. On a practical level, the trend is toward shared infrastructure with MPI and an expression of PGAS functionality through library calls to enable a multiplicity of language implementations.

Related Entries

►Array Languages
►Chapel (Cray Inc. HPCS Language)
►Coarray Fortran
►Collective Communication
►Fortress (Sun HPCS Language)
►Global Arrays Parallel Programming Toolkit
►HPF (High Performance Fortran)
►Memory Models
►MPI (Message Passing Interface)
►OpenMP
►SPMD Computational Model
►Titanium
►UPC
►ZPL

Bibliography

1. Chamberlain BL, Choi S-E, Christopher Lewis E, Lin C, Snyder L, Weathersby D (2000) ZPL: a machine independent programming language for parallel computers. Softw Eng 26(3):197–211
2. The cascade high productivity language. HIPS, 00:52–60, 2004
3. High Performance Fortran Forum (1993). High performance Fortran language specification, version 1.0. Technical report CRPC-TR92225, Houston
4. Nieplocha J, Palmer B, Tipparaju V, Krishnan M, Trease H, Apra E (2006). Advances, applications and performance of the global arrays shared memory programming toolkit. Int J High Perform Comput Appl 20:203–231

5. Numrich RW, Reid J (1998) Co-array fortran for parallel programming. SIGPLAN Fortran Forum 17(2):1–31
6. Open MP (2000) Simple, portable, scalable SMP programming. http://www.openmp.org/
7. Snir M, Otto S, Huss-Lederman S, Walker D, Dongarra J. MPI-the complete reference. The MPI Core, vol 1. MIT Press, Cambridge, MA
8. Split-C website. http://www.eecs.berkeley.edu/Research/Projects/CS/parallel/castle/split-c/
9. UPC language Specification, V1.2, May 2005
10. The X10 programming language. http://x10.sourceforge.net, 2004
11. Yelick K, Semenzato L, Pike G, Miyamoto C, Liblit B, Krishnamurthy A, Hilfinger P, Graham S, Gay D, Colella P, Aiken A (1998). Titanium: a high-performance Java dialect. Concurrency Pract Experience 10(11–13):825–836

Phylogenetic Inference

►Phylogenetics

Phylogenetics

Alexandros Stamatakis
Heidelberg Institute for Theoretical Studies, Heidelberg, Germany

Synonyms

Molecular evolution; Phylogenetic inference; Reconstruction of evolutionary trees

Definition

Phylogenetics, or phylogenetic inference (bioinformatics discipline), deals with models and algorithms for reconstruction of the evolutionary history – mostly in form of a (binary) evolutionary tree – for a set of living biological organisms based upon their molecular (DNA) or morphological (morphological traits) sequence data.

Discussion

Introduction

The reconstruction of phylogenetic (evolutionary) trees from molecular or morphological sequence data is a comparatively old bioinformatics discipline, given that likelihood-based statistical models for phylogenetic inference were introduced in the early 1980s, while discrete criteria that rely on counting changes in the sequence data date back to the late 1960s and early 1970s.

Computationally, likelihood-based phylogenetic inference approaches represent a major challenge, because of high memory footprints and of floating point intensive computations.

The goal of phylogenetic inference consists in reconstructing the evolutionary history of a set of n present-day organisms for which molecular sequence data can be obtained. In some cases it is also possible to extract ancient DNA or establish the morphological properties (traits) of fossil records.

Input

The input for a phylogenetic analysis is a list of organism names and their associated DNA or protein sequence data. Note that the DNA sequences for distinct organisms will typically have different lengths. In modern phylogenetics, instead of using the raw sequence data, a so-called multiple sequence alignment (MSA) of the molecular data of the organisms is used as input. Multiple sequence alignment is an important – generally NP-hard – bioinformatics problem. The key goal of MSA is to infer homology, that is, determine which nucleotide characters in the sequence data share a common evolutionary history. Because insertions and/or deletions of nucleotides may have occurred during the evolutionary history of the organisms (represented by their DNA sequences), such events are denoted by inserting the gap symbol - into the sequences during the MSA process. After the alignment step, all n sequences will have the same length m, that is, the MSA has m alignment columns (also called: characters, sites, positions). A simple example for an MSA of DNA data for the Human, the Mouse, the Cow, and the Chicken with $n = 4$ species and $m = 27$ sites is provided below:

```
Cow       ATGGCATATCCCA-ACAACTAGGATTC
Chicken   ATGGCCAACCACTCCCAACTAGGCTTA
Human     ATGGCACAT---GCGCAAGTAGGTCTA
Mouse     ATGG----CCCATTCCAACTTGGTCTA
```

Output

The output of a phylogenetic analysis is mostly an *unrooted* binary tree topology. The present-day organisms under study (for which DNA data can be extracted) are assigned to the leaves (tips) of such a tree, whereas the inner nodes represent common extinct ancestors. The branch lengths of the tree represent the relative