Chapel: Productive Parallel Programming at Scale[*]

Bradford L. Chamberlain (bradc@cray.com)
Cray Inc.; 411 First Ave S., Suite 600; Seattle, WA 98104

Within the High Performance Computing (HPC) community, application performance is king. HPC users target large-scale parallel machines in hopes of getting performance improvements proportional to the hundreds or thousands of processors on which their codes will execute. In order to fulfill this promise, their application codes must be written and executed in a scalable manner, avoiding unnecessary overheads as larger numbers of processors are applied to the problem.

Most HPC codes authored during the past decade have been written using a portable message-passing interface called MPI. It supports a rich set of routines for communicating between cooperating executables, including send/receive transfers, reductions, and all-to-all exchanges. While MPI has successfully supported scalable performance for several classes of applications on recent parallel architectures, it has some challenges in its future. The first relates to emerging parallel architectures that contain novel features such as multicore processors and increasingly heterogeneous capabilities. At this point it is not clear whether MPI will be sufficiently lightweight, flexible, and hierarchical to be a good match for such architectures. Even if it is, the second challenge relates to a growing dissatisfaction with MPI's negative impact on user productivity—particularly the ease with which HPC codes can be written, read, modified, tuned, ported, and maintained. Because MPI involves cooperating executables, programmers must write their code in a "fragmented" manner, manually dividing logical computations and data aggregates between the distinct program instances that comprise their application. This detail management results in a significant amount of bookkeeping code which can obfuscate an algorithm's intent and increase the brittleness and fragility of a code. While several parallel languages have been developed with the goal of improving this situation (e.g., HPF, ZPL, UPC, Co-Array Fortran, Titanium, NESL, …), most of them have had restrictions in their programming and execution models that have limited their generality and, by extension, their adoption.

At Cray, we are developing a new parallel language named Chapel that strives to improve HPC user productivity by enhancing parallel programmability without sacrificing the performance, portability, or generality currently enjoyed by MPI users. Our strategy for doing this is to define Chapel to support a *global view* of control and data, to include concepts for reasoning about and controlling *locality*, and to support language abstractions using a *multiresolution* approach. In this report, I will describe these themes and their impact on scalability for the application as well as for the programmer.

We define a language to have a global view of control if: (*i*) its entry point is executed using a single logical thread, and (*ii*) any additional parallelism is introduced using well-defined constructs in the language. These are not particularly unique features; note for example that Java meets these requirements. However, in the HPC realm, global-view models of control are a rarity. For example, programming models like MPI that are based on cooperating executables, including the common Single Program/Multiple Data (SPMD) style, rely on the fact that a number of concurrent threads will execute each program's entry point. Languages such as HPF and ZPL developed during the 1990's support a global view of control, yet their parallelism is limited due to the fact that their execution models are still SPMD in nature. In contrast, a Chapel program begins with a single task and supports the creation of additional tasks through arbitrarily-nested **begin** statements. Dynamic groups of tasks can be joined together using a compound **sync** statement. Tasks can also be created and joined in a more structured manner using **cobegin** and **coforall** statements. Synchronization between tasks is expressed in a data-driven manner through the use of **single** (assignment) and **sync** variables. This support for unrestricted anonymous task parallelism in Chapel permits programmers to use a unified set of mechanisms to express parallelism from the coarsest levels of an algorithm to the finest. Moreover, these tasks can be implemented using multiple nodes within a machine, multiple cores within a processor, or even parallel operations within a core such as vector instructions or multithreading support.

On modern parallel machines, locality—or more accurately, affinity between threads and the data values they access—plays a crucial role in achieving scalability because of the relative latency required to access memory that is local to a processor as compared to memory which is remote. To expose these concerns to the user, Chapel defines a concept known as a *locale*—a unit of the target architecture that supports processing and storage capabilities, and for which the locale's tasks will have (relatively) uniform access to its memory. As an example, on a commodity parallel architecture, a multicore processor or SMP node and its associated memory might be considered a locale. The primary role of the locale is to distinguish between data and tasks that are "here" (local and cheap) versus "there" (remote and expensive).

[*] This work is supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0001.

Google Seattle Conference on Scalability, June 14, 2008.

The Chapel language contains a **locale** type that is used to represent the architectural locale concept at runtime. When running a Chapel program, the user specifies the number of locales on which they want it to execute via a command-line flag, and this request is passed along to the machine's job scheduler. Within a Chapel program, the programmer can refer to *Locales*, a compiler-provided array of **locale** type that represents the machine resources on which the program is running. Like any other array, *Locales* can be accessed, reshaped, and sliced to refer to the machine resources in a way that logically suits the program in question. A program's initial task starts executing on *Locales*[0]. Tasks and data values can then be created on other locales using an **on** clause, which indicates that the subsequent statement should be executed by the specified locale. An on clause can also be specified using a variable reference, in which case the statement will be executed on the same locale in which the variable is stored. This permits a task's affinity to be expressed in a data-driven manner.

Chapel also supports a global view of data, implying that distributed arrays can be declared in terms of their global size and accessed via global indices. This is in contrast to distributed-memory programming models (as well as many partitioned global address space models) in which array data must explicitly be broken into per-node chunks. Chapel's global-view data structures are defined using *domains*—a first-class language concept that represents a set of indices. Domains can be thought of as representing an array's size and shape without any of the data representing the array's values. In addition to traditional rectilinear arrays whose domain indices are integer tuples, Chapel also supports associative domains that can store indices of any value type, opaque domains whose indices are anonymous, and sparse domains that represent sparse subsets of other domains. In Chapel, domains are used to declare and reallocate arrays, to define iteration spaces, and to express slices or sub-arrays. Domains and arrays support **forall** loops which result in parallel iteration over their component indices/elements. Whereas a coforall loop explicitly generates a task per iteration, forall loops typically result in coarser-grained parallelism, to amortize parallel overheads across multiple iterations. Domains, arrays, and forall loops form the basis of Chapel's data parallel features which support a higher level of abstraction than the task parallel features described previously.

Each domain can optionally be defined in terms of a distribution that specifies: (*i*) how its indices are distributed between the locales, (*ii*) how each locale should store the domain's indices in memory, as well as the array elements corresponding to those indices, and (*iii*) an interface that supports operations on domains and arrays, including indexing, slicing, and parallel iteration. Advanced Chapel users can author their own distributions. A distribution can be thought of as a recipe that maps high-level operations such as forall loops and domain slicing down to the tasks and on clauses required to specify the distributed parallel execution. Most distributions will be parameterized by architectural characteristics such as the number of cores per locale or bytes per cache line in order to define operations on the distributed data structures that suit a target machine's characteristics. It is Chapel's goal to allow most users to write global-view code in terms of domains, arrays, and their supported operations, while leaving the more arduous task of implementing distributions to parallel computation specialists. In this way, distribution libraries can be created by experts, allowing users to trivially plug in different choices. Such changes have an impact on the program's parallel implementation and performance without affecting its semantics.

Distributions serve as an example of Chapel's multiresolution design, in which programmers can write code as abstractly or as close to the machine as they require: programmers can start by writing a parallel code on one locale, making use of the multiple processing cores within that locale. Once their programs are working, they can add **distributed** clauses to their domain declarations to indicate that the domains and their arrays should be distributed, but leaving the choice of the distribution up to the compiler. As programmers want to assert more control over the distribution, they may choose to select a distribution from the standard library, such as *BlockCyclic.* As they tune their programs further, they may choose to implement their own distributions in order to take advantage of domain-specific characteristics in their applications. Our goal is to achieve enough performance through distributions to allow programmers to stop here. However, programmers who are hampered in some way by the distribution concept can also choose to avoid it altogether and program the locales directly, using the task parallel features and fragmented data structures as in current parallel programming models. We hope that this will not generally be necessary, yet we also believe that giving the user the ability to abandon high-level abstractions when desired supports general parallel programming. This is in sharp contrast to previous data parallel languages which provided high-level abstractions, but typically had no way to avoid them to get closer to the machine. We believe that such flexibility is important for generality, and possibly also to achieve maximum performance and scalability.

We have designed Chapel's features to be an asset to scalability in several ways. First, we have defined the language to be fairly well-abstracted from the target machine's resources and mechanisms. This is in sharp contrast to programming models like MPI, whose user-level concepts expose implementation details such as synchronization, copying, and buffering. While such mechanisms may be useful in certain contexts, specifying them makes it challenging to take advantage of communication mechanisms that are more capable—such as architectural support for single-sided messaging—or lighter-weight—such as communication between the cores of a multicore SMP node. In contrast, Chapel exposes concerns like parallelism, data placement, and locality, yet without binding the concepts to a specific level of the architecture or a particular implementation mechanism. This permits the compiler to map Chapel's abstract features to the many levels and features of the target architecture as best suits the program. Moreover, scalability typically requires experimentation on the user's part in order to optimize performance. For this reason, Chapel's features designed to improve programmability go a long way toward helping users tune their codes to achieve maximum performance and scalability on large numbers of processors.