Coarse Grained Parallel Function Evaluation

Phillip Rulon and Saul Youssef
Supercomputer Computations Research Institute
Florida State University
Tallahassee, Florida 32306-4052

Parallel function evaluation is discussed as a generalization of event level parallelism in high energy physics. A facility for parallel function evaluation using the Types package requires only a single subroutine call, does not require separate serial and parallel source codes and produces an executable image which can either be run serially or in parallel. Performance, practical experience and future developments are discussed.

Introduction

It is well known that detector simulation and analysis are amenable to "event level" parallelism where independent events are analyzed or simulated by independent processes. This approach has effectively solved the main production computing problems in HEP—event simulation and reconstruction. In spite of this success, however, an important class of problems do not benefit from event level parallelism either because one is waiting for a particular event (debugging a simulation or event displays, for example) or because the application is not a loop over events. Examples from recent work [1] include detecting errors in GEANT geometries (GVerify), absorbing CAD objects into GEANT (OCTAGON), generating histograms or n-tuples from Zebra files (zHist) and physics optimized calorimeter clustering (CTool). Although these problems all require significant amounts of CPU time, none are amenable to event level parallelism.

CTool is a good example of non—event level parallelism. The main calculation in CTool is the minimization of a function \( F \) in a six dimensional space where evaluating the function once may take many minutes or even hours on current machines. Here, one can take advantage of parallelism by evaluating \( F \) at several points at once and then using the results to choose a new set of points until a minimum is found. Although the evaluations of \( F \) are independent, the points are not known initially and so this class of algorithms cannot be executed by completely independent processes. Note, however, that this problem could be solved if one had "parallel function evaluation" where \( F \) could be applied to a set in parallel resulting in an output set of corresponding results. It turns out that parallel function evaluation is also a convenient solution to all of the non-event level problems we have investigated.

Since parallel function evaluation clearly solves a broad class of problems not amenable to event level parallelism, we have introduced such a solution using Types [2] to represent input and output sets. The dParallel package allows parallel function evaluation to be invoked with a single subroutine call supplying only the set to be acted upon, the function to be evaluated in parallel and the set to be returned. Unlike many other parallel programming systems, no new syntax is introduced besides this single call. In particular,
Using dParallel

Parallel function evaluation requires a way to represent sets where the elements of a set may be complicated data structures (events, for example). This sort of data abstraction is available within the Types package. Types is a complete data abstraction system with i/o, operating system commands and machine to machine transport of objects. In Types, an abstract object like an ordered set is created and manipulated symbolically. For instance, once an ordered set “A” is created, it is subject to all ordered set (“dOrSet”) operations such as putting things into the set or taking them out. All such operations are done with subroutine or function calls within standard Fortran (or C). For example,

Call dOrAdd (Thing, A)
adds object “Thing” to ordered set A, or

Call dOrGet (k,A, AnElement)
fetches the k’th element of A and puts it into object AnElement. Of course, all Types objects can be broken down and ultimately mapped into the Fortran intrinsic types INTEGER, REAL and CHARACTER. An object such as “AnElement” in the last example is often allocated as it is created in which case it would be referred to as “o(AnElement)” rather than “AnElement”. For example,

Call dOrGetA (o, k,A, AnElement)
fetches the k’th element of A and puts it in the allocated object “o(AnElement)”. Here, the “o” argument to dOrGetA acts as a conventional reminder that the created object is allocated and needs to be deallocated at some point. One of the main benefits of Types is allowing encapsulation of high level functions. For example, a central detector tracker may work in the following way:

Call CDTracker (o, CentralDetector, Tracks)
where “CentralDetector” is an object containing central detector event information and the output o(Tracks) is a set of central detector tracks. In contrast with the normal Fortran style, this is typically done in Types with no global communication between CDTracker and the calling program. Given such a subroutine, all one needs to do to run in parallel is to

Call dParallel (o, CDset,CDTracker, SetsOfTracks)
where CDset is a set of CentralDetector objects. When this call returns, each element of o(SetsOfTracks) contains the tracks made from the corresponding CentralDetector object. A program containing such a call can be compiled and linked to the standard Types library producing an executable program called, for instance, “Test”. Test can then be run under dbx, or run as a serial program by
or run in parallel on, say, ten machines, by
% Test -p10

In the parallel case, the ten least loaded machines are chosen from a list kept in the users home directory. dParallel may be called recursively so that CDTracker may also have dParallel calls and may express further parallelism. In the first release of dParallel, only one level of parallelism is expressed at a time and further recursive calls to dParallel are executed serially. However, we are currently working on a new implementation of dParallel which allows multiple levels of parallelism to be executed simultaneously. For example, a tracking program may be written expressing parallelism over events, over tracking devices, over segments within a tracking device, over coordinates within a tracking device, over tracks and over hits. When such a code is executed, a particular dParallel call will either use more of the available nodes or run serially based purely on run time considerations.

**Implementation**

The most successful and far reaching decision in implementing dParallel is the decision to use symmetric executable images. A single executable program functions as the serial version or the parallel supervisor or a parallel remote task. The current implementation uses the standard Unix utilities “rexec” and “select” to launch and control remote processes. Interprocess communication is done through standard Unix input, output and error streams. A substantial effort has gone into error handling so that any exception in the supervisor or in any of the remote tasks causes all remote tasks and the supervisor to be killed after reporting the error. For instance, a “control C” from the terminal is trapped so that all remote tasks are killed before killing the root process.

The performance of a dParallel job depends on the time needed to fetch initial load information, to start remote processes, to send and receive objects over the network, and the time to do the computations on the remote tasks. Space considerations prevent a detailed discussion of performance here, but, roughly speaking, a dParallel program taking one minute or more of wall clock time will run at better than 90% efficiency on our current hardware (IBM 340s on an ethernet). This means that dParallel is indeed useful for daily programming problems (e.g., running a 1 hour job in 2 minutes) as well as large scale production computing. As an example of a production application, we have modified the GVerify package to use dParallel. GVerify is a 20 thousand line application written with extensive use of Types (Types itself is approximately 60 thousand lines long). Parallelizing GVerify required modifying only three small subroutines and immediately gave essentially full parallel efficiency verifying the SDC GEANT geometry.

dParallel has quickly become a routine part of our programming repertoire and will remain an important component of new software packages. Current work is concentrated on further reducing communications and startup overhead and in implementing recursive parallel execution. dParallel is available via anonymous ftp from freehep.scri.fsu.edu.
References

[1] For GVerify, see S. Youssef, "Detecting Errors in GEANT Geometries", SCRI technical report FSU-SCRI-90-174; for OCTAGON, see these proceedings and S. Youssef, "Some Problems in the Geometric Representation of Detectors", Workshop on Detector and Event Simulation in High Energy Physics, NIKHEF, Amsterdam, 1991; for CTool, see these proceedings. More information about these software packages can be found in FreeHEP (via the world-wide-web or via ftp from freehep.scri.fsu.edu).