Determining Temperature Differential to Prevent Hardware Cross-Contamination in a Vacuum Chamber

Goddard Space Flight Center, Greenbelt, Maryland

When contamination-sensitive hardware must be tested in a thermal vacuum chamber, cross-contamination from other hardware present in the chamber, or residue from previous tests, becomes a concern. Typical mitigation strategies involve maintaining the temperature of the critical item above that of other hardware elements at the end of the test.

A formula for relating the pumping speed of a chamber, the surface area of contamination sources, and the temperatures of the chamber, source, and contamination-sensitive items has been developed. The formula allows the determination of a temperature threshold about which contamination will not condense on the sensitive items. It defines a parameter alpha that is the fraction given by (contaminant source area)/[chamber pumping speed × (time under vacuum)$^{0.5}$]. If this parameter is less than 10^{-6}, cross-contamination from common spacecraft material will not occur when the sensitive hardware is at the same te,mperature as the source of contamination (The chamber is isothermal within 5 °C.).

Knowing when it becomes safe to have the hardware isothermal permits faster and easier thermal transitions when compared with maintaining an arbitrary temperature differential between parts. Furthermore, the standard temperature differential may not be adequate under some conditions (alpha>10^{-4}).

This work was done by David Hughes of Goddard Space Flight Center. For further information, contact the Goddard Innovative Partnerships Office at (301) 286-5810. GSC-16244-1

SequenceL: Automated Parallel Algorithms Derived from CSP-NT Computational Laws

Chip manufacturers and developers of parallel and/or safety-critical software could benefit from this innovation.

Goddard Space Flight Center, Greenbelt, Maryland

With the introduction of new parallel architectures like the cell and multicore chips from IBM, Intel, AMD, and ARM, as well as the petascale processing available for high-end computing, a larger number of programmers will need to write parallel codes. Adding the parallel control structure to the sequence, selection, and iterative control constructs increases the complexity of code development, which often results in increased development costs and decreased reliability.

SequenceL is a high-level programming language — that is, a programming language that is closer to a human's way of thinking than to a machine's. Historically, high-level languages have resulted in decreased development costs and increased reliability, at the expense of performance. In recent applications at JSC and in industry, SequenceL has demonstrated the usual advantages of high-level programming in terms of low cost and high reliability. SequenceL programs, however, have run at speeds typically comparable with, and in many cases faster than, their counterparts written in C and C++ when run on single-core processors. Moreover, SequenceL is able to generate parallel executables automatically for multicore hardware, gaining parallel speedups without any extra effort from the programmer beyond what is required to write the sequential/single-core code.

A SequenceL-to-C++ translator has been developed that automatically renders readable multithreaded C++ from a combination of a SequenceL program and sample data input. The SequenceL language is based on two fundamental computational laws, Consume-Simplify-Produce (CSP) and Normalize-Transpose (NT), which enable it to automate the creation of parallel algorithms from high-level code that has no annotations of parallelism whatsoever. In our anecdotal experience, SequenceL development has been in every case less costly than development of the same algorithm in sequential (that is, single-core, single process) C or C++, and an order of magnitude less costly than development of comparable parallel code. Moreover, SequenceL not only automatically parallelizes the code, but since it is based on CSP-NT, it is provably race free, thus eliminating the largest quality challenge the parallelized software developer faces.

Compiling functional code to C++ is not new. Compiling functional code to readable C++ that runs in parallel is much more of a challenge, and that was the majority of this effort. For current purposes in this effort, readability of the generated code is crucial, in case the human programmer wishes to add annotations, or to inspect the code for verification purposes. Moreover, by compiling to C++ it is assured that SequenceL can be used in any application where C++ could be used.

SequenceL has been found to discover all potential parallelisms automat-