SANDIA REPORT

SAND2011-7604
Unlimited Release
Printed October, 2011

Hierarchical Resilience with Lightweight Threads

Kyle B. Wheeler, Sandia National Laboratories

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2011-7604
Unlimited Release
Printed October, 2011

Hierarchical Resilience with Lightweight Threads

Kyle B. Wheeler

Abstract

This paper proposes methodology for providing robustness and resilience for a highly threaded distributed- and shared-memory environment based on well-defined inputs and outputs to lightweight tasks. These inputs and outputs form a failure "barrier", allowing tasks to be restarted or duplicated as necessary. These barriers must be expanded based on task behavior, such as communication between tasks, but do not prohibit any given behavior.

Observations

One of the trends in high-performance computing codes seems to be a trend toward self-contained functions that mimic functional programming. Software designers are trending toward a model of software design where their core functions are specified in side-effect free or low-side-effect ways, wherein the inputs and outputs of the functions are well-defined. This provides the ability to copy the inputs to wherever they need to be—whether that's the other side of the PCI bus or the other side of the network—do work on that input using local memory, and then copy the outputs back (as needed).

This design pattern is popular among new distributed threading environment designs. Such designs include the Barcelona STARS system, distributed OpenMP systems, the Habañero-C and Habañero-Java systems from Vivek Sarkar at Rice University, the HPX/ParalleX model from LSU, as well as our own Scalable Parallel Runtime effort (SPR) and the Trilinos stateless kernels. This design pattern is also shared by CUDA and several OpenMP extensions for GPU-type accelerators (e.g. the PGI OpenMP extensions).

Beneficial Runtime Information

If programmers are willing to design their algorithms in that way, they are conceding a great deal of potential flexibility and providing the runtime environment with additional information about their program's behavior.

First, such programs are essentially informing the runtime that the input memory for a given task, at the point where they spawn that task—either synchronously or asynchronously—is in a coherent state. More than just coherent, it is in a state that can be used to spawn work.

Additionally, if the function is spawned asynchronously, the programmer is declaring that the thread may technically use any method to achieve the output—whether the computation happens locally, on a GPU, or across the network is immaterial, as long as the output ends up where the spawning function has specified.

Leveraging the Emergent Design

Because the input is in a state that can be used to spawn work, the runtime is free to store a copy of that input somewhere—anywhere from elsewhere in memory to non-volatile RAM to on disk or some other reliable storage media. If the thread should, then, die for any reason—the node went down, the link went down, the answer generated doesn't meet correctness criteria, etc.—the runtime has all the information necessary to re-spawn that task somewhere else and hand it the same input. Not only could the task be re-spawned in

the case of failure, but multiple copies of it could be spawned in multiple locations, if that is deemed useful. In addition to a well-defined input, certain additional conditions must be true in order to enable both of these opportunities, which will be addressed in a moment.

While the spawned task may use any method to compute the output, there is one quality that distinguishes it, and that is that it must, by definition, exist for the entire interval between spawning and reporting the output.[1] It need not be *alone*, or even executing, for that entire interval, however. It is free to spawn both more threads with well-defined inputs and outputs—referred to hereafter as "functional tasks"—and threads without well-defined inputs and outputs, or "shared-state tasks".

New Task Categories

These two new categories of task define the limits of how they are handled by the runtime. Shared-state tasks, because they share state with each other and with their parent, may not move away from their shared state. If the parent task must migrate (and it is hard to envision when this would be a good idea, for performance reasons), all of the tasks that share that state must move with it. Additionally, should any of these shared-state threads fail for any reason, all of the threads that share that same state must also die, because the state they share can no longer be trusted. In that sense, the parent functional task is dependent upon the child shared-state tasks, just as they are dependent upon it.

When a functional task spawns another functional task, that new task, unlike the shared-state tasks, is free to execute anywhere, because of it's well-defined inputs and outputs. The dependence relation to the parent is different, however. Should the child functional task die, it can be restarted, as long as the inputs have been saved. If the parent functional task dies, however, the child functional task must also die. This is because the parent's state is suspect, and so the input state of the child is also suspect. The parent will, in all likelihood, be restarted, and will regenerate not only the child's input, but will respawn the child.

The Partitioned Hierarchy

In this sense, we have a partitioned hierarchy of threads. These partitions define reliability domains: anything within a domain shares a fate with everything else within that domain. This is illustrated in Figure 1.

This hierarchy is important, because we can define thread behavior and consequences of that behavior in terms of the hierarchy. For instance, the description of task behavior so far has assumed that tasks are entirely side-effect free, where side-effects can be considered to be nearly all forms of inter-task communication. Such communication creates a two-way

[1] Continuations may appear to violate this rule, but in that they share the same output-generation requirement, they can be considered to be different phases of the same task.

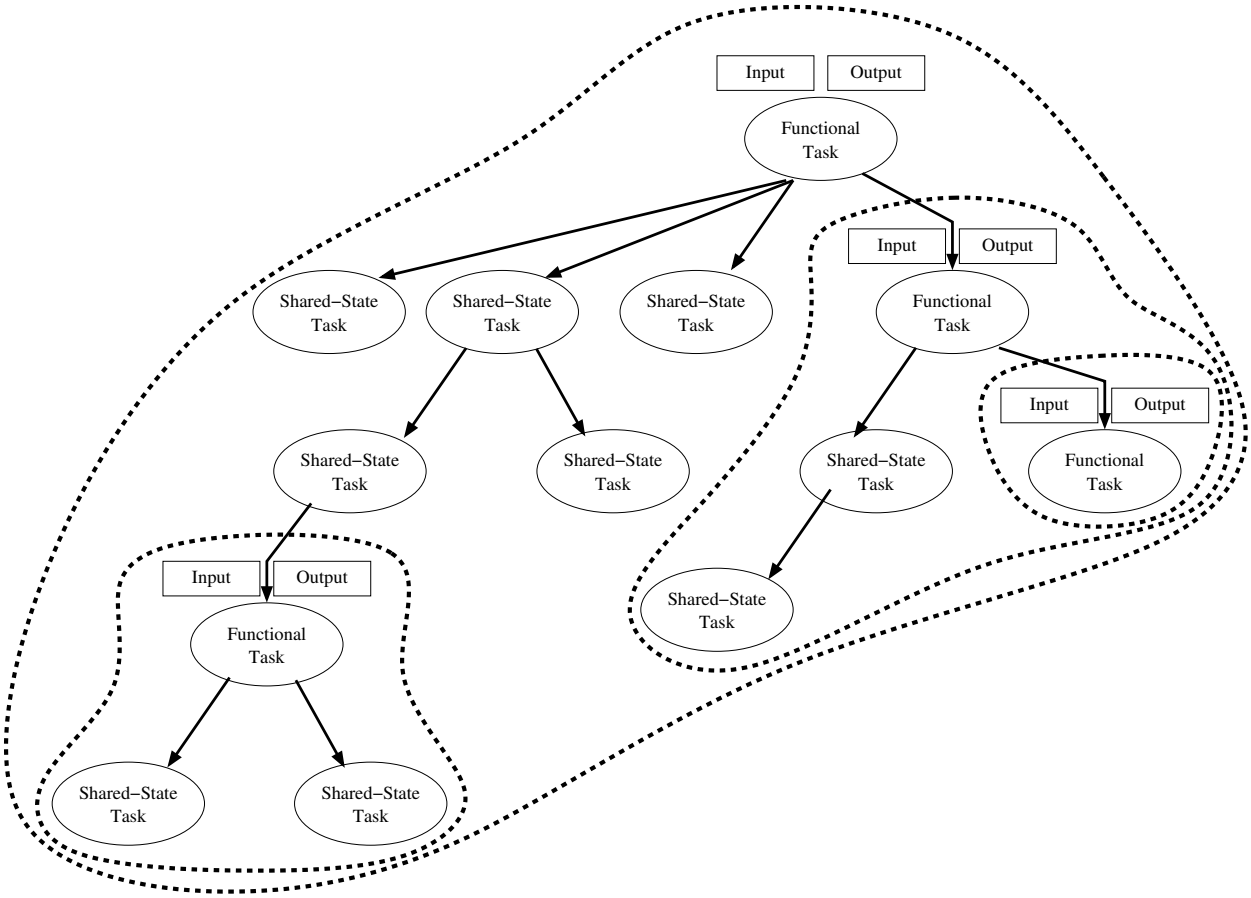

Figure 1. Partitioned Hierarchy with Reliability Domains

dependence relation: tasks that communicate have altered each other's state, and so if either of the participants in that communication lose that state, the other is needed to re-participate in regenerating that state. When communication creates a dependence relation, it can be viewed as simply expanding and merging the respective tasks resilience domain.

It is possible, if the communication is considered "reliable" and "independent" (i.e. generated independent of the recipient) that this dependence may be broken—communication itself may be stored safely and re-played, provided that certain conditions are met, such as that the timing of the communication is not part of the communication. It is also possible, in very limited cases (such as periodic sanity-checking) that the communication does not affect any *significant* state within either task and therefore should not establish a dependence relation of any kind. Any of these special-case situations may not be easy to detect by the runtime, but can be declared to the runtime by the programmer easily enough.

The other obvious exception is the parent-child relationship of functional tasks, wherein the parent is communicating to the child via its input and the child is communicating to the parent via its output.

A key point to be made here is that this hierarchy is not *limiting* behavior, but rather is a way of recording the consequences of behavior upon resiliency.

Micro-checkpointing

This partitioned hierarchy, with it's defined input/output boundaries, also defines a set of hierarchical micro-checkpoints, and with them, a hierarchy of failure, recoverability, and resilience.

Interestingly, these micro-checkpoints don't always need to be copied into additional storage, and when it is copied, it need not be copied into particularly non-volatile storage. The choice of whether or where to store these checkpoints may be made at runtime based on the expected probability of failure. In many cases, nodes in large computers have early-warning indicators that they are becoming unreliable. Such indicators include one-bit (corrected) ECC errors and/or increasing CPU temperature, among other things. If a node is deemed "risky", more precautions may be taken to ensure that the computation intended to execute on that node is reliable—that may include storing related checkpoints more securely, among other things.

Avoiding sending the checkpoints to expensive storage improves performance and likely reduces power consumption, but increases risk—it flattens the resilience hierarchy. If the inputs are not stored, the child task cannot be restarted, and so if the child dies the parent must die as well. Interestingly, these checkpoints have a lifetime. Once the output is generated, the input no longer needs to be stored, and may be discarded.

While the micro-checkpoints may simply kept in memory, duplicated across the network in other node memories, sent to NVRAM, they may also be sent to disk. It may be beneficial to do both: keep most checkpoints in fast local memory, but gradually flush outstanding checkpoints (for longer-lived tasks) to disk. One consequence of the expected pattern of sending micro-checkpoints to disk is that the load placed on those disks is significantly reduced. Traditional checkpoints are particularly challenging for even custom-designed filesystems, while a slow trickle of checkpoints will most likely achieve better disk performance due to the reduced disk load.

Task Teams

The hierarchy of tasks is also useful for establishing a well-defined and understandable definition of task "teams". These teams may define the boundaries of things like memory- and CPU-affinity, synchronization and collective operation scope (i.e. for eurekas, barriers, etc.), migration, and so forth. Threads, then, can technically belong to multiple teams: each team belongs to its parent team. This hierarchy also defines a simple set of operations that can be performed upon such teams; it is easy to wait for a team to finish (cease to

exist, i.e. return its output), it is easy to spawn a new team, and membership in a team is defined by particular operations (such as communication), but has resiliency consequences. To avoid consequences, teams are best expanded internally, by their originating functional task spawning more team members.

This relationship reflects existing patterns of task behavior. For example, parallel loops can appropriately be called "teams", because one typically needs to wait for all iterations of a loop to complete before the parent can continue. The most efficient means of spawning parallel loop iterations is in a tree, which naturally includes them in the same team and provides the necessary opportunity for efficient exit detection.

DISTRIBUTION:

1 MS 0899 RIM-Reports Management, 9532

Sandia National Laboratories