Agent-Based Distributed Parallel Processing

Jason Byassee
jason.byassee@trw.com

Traditional solutions to large-scale signal processing involve massive supercomputers consisting of multiple processors. Data is processed in a pipelined fashion that can incorporate multiple machines and numerous computing stages. The limitations to this approach include flexibility, scalability, cost and fault tolerance. Our research is focused on a new approach to signal processing that utilizes a computing cluster; a network of small personal computers connected via a high speed network medium. In this system, a processing task is partitioned and divided among a family of lightweight agents. These agents are distributed throughout the cluster and compete for computing resources. This approach to signal processing is advantageous in that the system operates as an autonomous entity. Agents execute as a collaborative team, working around node failures and system bottlenecks. Additional computing resources can be added and exploited dynamically, enhancing both the flexibility and scalability of the system.

1 Agent Model

The goal of this effort is to implement signal processing algorithms in parallel using a family of mobile agents. To this end, task parallelism serves as the foundation for our agent-based parallel programming model. At the system level, task parallelism represents an object-oriented design strategy, whereby both data and the operations on those data are encapsulated in a single entity. Adhering to this paradigm, each task has the ability to operate independently, yet communicate and exchange data with other tasks.

![Figure 1 – Task Parallelism](image)

The advantages of the task parallelism model for distributed parallel processing are twofold. First, the mapping of tasks to processors is truly dynamic, remaining completely independent of the algorithm. Second, this model is inherently fault tolerant in that there is no central data store or control mechanism. In the context of task parallelism, an agent is a natural task abstraction. The
task agent encapsulates the task data, operations to be performed on that data, and the necessary communication mechanisms.

Agent task patterns are of particular interest in parallel processing, where the focus is on partitioning and delegating tasks among agents. The Master-Slave pattern is a common task design model incorporated in a broad domain of parallel applications. This Master-Slave model is based on a divide and conquer strategy in which a master delegates tasks to one or more slaves that in turn are distributed throughout the system and work in parallel. The standard agent-based implementation involves master and slave agents. The execution sequence is as follows [6]:

1) The master agent creates a slave agent.
2) The slave agent moves to a remote host and performs a task.
3) The slave agent returns to the master.
4) The slave agent passes task results to the master.

Our agent-based architecture relies on a family of agents to perform processing tasks. Each task is partitioned and divided among the agents in the family. The agent family is a derivation of the Master-Slave pattern, where agents are organized in a parent-child hierarchy. Parent (master) agents delegate tasks to child (slave) agents. Child agents, in turn, can delegate tasks to new child agents. The following figure depicts this agent organization:
The approach taken with task agents in this architecture differs from the intelligence-based approach of many agent systems. The agents here are explicitly lightweight with basic heuristics. Agent interactions are clearly defined and simple. At an individual level, each agent is a negligible entity. From a system perspective, the small pieces (agents) produce tangible results in collaboration. This approach is loosely based on the principles of artificial life; a system is modeled after living organisms [1].

In this system, lightweight agents are akin to a colony of ants. In an ant colony, individual ants collaborate to locate food, relay a “food discovered” message, and finally transport the food back to the hive. The actions of an individual ant, out of context, seem to be meaningless. The behaviors and interactions of each ant are extremely simple, yet the operation, taken as a whole, is remarkably efficient. This biological entity conforms to the notion that the whole is greater than the sum of the parts. An agent system modeled in this manner provides inherent fault tolerance. Basic heuristics and simple interactions provide the potential for systematic recovery from lost pieces, or agents.

2 Application Programmer Interface (API)

A primary design goal of this architecture is to provide a simple interface for the algorithm developer. With this in mind, our goal is to hide all details about both the state of execution and the execution environment. For example, if a node fails during execution and some number of agents are lost, agent recovery should be a transparent process. Agent distribution, current location and network communication details are incorporated in the underlying software infrastructure. This is conceptually similar to a remote proxy, where clients (agents) interact with servers (other agents) via
a local representative. In our design, the local proxy is not a remote agent stub, but rather a link to a true remote agent.

2.1 Agent Components

Task Agents are associated with two components: 1) the Agent Link, and 2) the Agent Monitor. Each Task Agent is created with a corresponding Agent Link that serves as the agent’s interface. All communication is performed via the Agent Link class. The Agent Link encapsulates network communication detail with methods that include connect(), write() and read(). The Agent Monitor performs fault detection (a premature agent death) by monitoring the parent and child(ren). When a dead child agent is detected, the recovery process (agent regeneration) is a coordinated effort between the Agent Monitor and the Agent Link, remaining transparent to the Task Agent itself.

![Figure 4 – Task Agent Components](image)

2.2 Application Implementation

Application implementation involves writing code for a specific algorithm. The programmer creates a specific Task Agent class (for example, a FFTAgent that performs a portion of a Fast Fourier Transform) that inherits from the Abstract Agent class. The algorithm specific code is placed in the overridden doJob() method. Each agent may need to create one or more child agents (depending on the specific algorithm) in order to further partition the task. Child agent creation is a three-step process:

1) Create a new child agent of the desired type:

   `<Type>Agent childAgent = new <Type>Agent();`

2) Create a link to connect to this new child agent:

   `AgentLink childLink = new AgentLink();`

3) Connect this agent (denoted by the keyword “this”) to the new child by calling the connect() method of the Agent Link class:

   `childLink.connect(this, childAgent); // connect “this” agent to the child`
Task Agents communicate via Agent Links. Each agent is provided a link to its parent (as a class member variable - myParentLink). Data is read from immediate family members (parent and children) via the “read<Type>()” method. Data is written to family members via the writeData() method.

```c
<Type> input = myParentLink.read<Type>(); // read input data from parent agent
...
childLink.writeData(data); // write data to a child
...
<Type> childResult = childLink.read<Type>(); // read result from child
myParentLink.writeData(result); // write result back to parent agent
```

3 Fault Tolerance

Fault tolerance in distributed systems is a difficult problem that remains an ongoing subject of industry research. The two phases of fault tolerance involve 1) fault detection, and 2) fault recovery. Recovery poses a significant challenge due to the distributed nature of the execution environment. There currently exist numerous fault recovery methods that incorporate some form of global checkpointing. When a fault is encountered, the entire system is “rolled back” to the last consistent state. The drawbacks here are 1) significant overhead in coordination between all tasks in a distributed system (which is necessary to periodically record “global snapshots”), and 2) the potential domino effect of rollbacks in the attempt to reach a consistent system state [5].

Our system enables fault tolerance with a design that enforces a functional programming paradigm at the algorithm level. This model is conducive to fault tolerance in that it both simplifies and isolates error recovery. Functional programming introduces the notion of referential transparency. Each task is implemented as a “lightweight” entity that remains side-effect free. Prior research, with respect to functional programming, has provided the influence for our programming model [4]:

“Function invocation constitutes a natural rollback point... The function to be computed, its arguments, and the destination of the function’s result are all the information necessary to restart the function, and can be saved by the function’s caller at low cost... Because of referential transparency, the result of the restarted computations is identical to that of the original computation, merely delayed in time.”

From the system perspective, this paradigm allows an isolated asynchronous rollback recovery implementation at the task level. In contrast with global checkpointing, rollback is isolated (task localized) to each individual task. It should be noted that we have avoided both the redundancy associated with replication and the uncertainty linked with forward error recovery. This scheme is made possible by a functional model in which 1) data exchange is limited to passing input parameters and returning results and 2) program results are independent of function execution sequence.

4 Service Infrastructure

The software infrastructure that supports the agent-based model consists of various service components. Each service is available on each node in the system so as to maintain a completely distributed, fault tolerant execution environment. An example of one such service is resource
supervision that integrates availability and load balancing components. A resource directory (Registry) is maintained on each node that contains load and state information. The directory is updated via a discovery process when new resources are added to the system. When a node becomes unavailable (via a crash or voluntary withdraw), the local registry is updated by the first agent that has attempted (and failed) to communicate with an agent on that remote node.

A team of mobile agents serves to communicate and disseminate status information among nodes. These agents travel throughout the system, operating autonomously. The following diagram depicts the load balancing implementation:

![Diagram of load balancing implementation]

5 Deployment

The first deployment of our system is in an office network environment, where desktop machines serve as agent hosts. In this environment, agents “borrow” unused processing cycles from idle machines. This platform is designed to serve as a rigorous test of the fault recovery mechanisms, due to the fact that agents are vulnerable to a myriad of dynamic changes in the execution environment.

In this deployment, the software is packaged in a Java applet that is embedded in a corporate web page (Agent Home Page). When a participant navigates to the Agent Home Page, the applet is downloaded and initialized. Surfing to this page enables the participant’s machine to both host agents and initialize parallel processing jobs. The applet runs as a background process where agents are hosted intermittently, essentially “coming and going” transparently to the user. This is a harsh environment, where agents can be killed at any point during execution. Machines are continuously being added and removed from the network.

The following figure depicts the applet on the Agent Home Page. The applet consists of a display and two buttons. The display indicates (measured by heads) the number of agents currently running on the local machine. The “Go Away” button, when pressed, immediately destroys all agents running on the machine. The “Tell Me More” button, when pressed, provides participants with information about this experiment and allows an agent-based parallel processing job to be initialized.
The primary goal of this deployment is to test the fault recovery mechanisms of our agent-based architecture. The test application is a simple lexicographic quicksort on a string of characters. Data was collected over several weeks of continuous operation. A fault rate of 38% was incurred during this time period. In this context, the fault rate refers to jobs in which one or more agents were lost. These lost agents are a result of a participant(s) either 1) explicitly killing hosted agents or 2) simply closing the web browser (and thereby terminating the applet).

Of the jobs that did not incur faults, the success rate (correct lexicographic sort) was close to optimal (99%). This result supports the design and implementation of task agent communication and coordination. The data was dynamically partitioned and distributed among the agent families, whose members operated on various nodes throughout the network.

Of the jobs that did incur faults, the success rate was 93%. Upon further analysis, the error rate was proportional to the size of the task agent family (i.e. the more agents, the larger propensity for error). Ongoing study pinpointed to two likely causes for this margin of error. First is the physical network connection, where congested lines result in latencies that exceed timeout values. This problem was deemed to account for a small fraction of the error rate. Second, and most important, is the threading mechanism. Our software infrastructure, written in Java, makes heavy use of Java’s threading package. Each task agent is associated with three threads of execution – one for the agent and two for the agent monitor (the component used for fault detection and recovery). In addition, local services (agent servers, thread pools, network interface objects, etc.) utilize some number of threads. It is not uncommon for 1000 threads to be in the “runnable” state simultaneously on a single node. At different thresholds, we noted indiscriminate behavior in the thread scheduler where various threads were “starved” for some period of time, essentially not receiving an adequate time slice.
Following this first test, the software was deployed in a viable signal processing environment consisting of a dedicated 32-node, Intel-based Linux cluster connected via Myrinet. We first aimed to address and improve the threading performance. In this environment, the implementation involved the raw application (as opposed to the applet described previously). Both Java green and Linux (2.2.5-15smp) native threads tested with Blackdown and Sun JDK 1.2.2 exhibited the same inconsistent behavior as observed previously. Using Java green threads we often observed “busy waits”, where a thread in a tight loop did not relinquish CPU control, and hence all other threads were starved. With Linux native threads, hard limits (due to architectural limitations) were attained and therefore introduced severe restrictions. Although our code was extensively tested and analyzed for deadlock and starvation bugs, this possibility remained, which led to the following trial.

Our next test involved compiling the code into a native executable using TowerJ [7], which adheres to more of a lightweight process model. This experiment proved to be very successful with respect to thread execution and stability. Data was again collected over several weeks of continuous operation, where selected agents were killed at random. The error detection and recovery rates were very successful, improving to 100% and 99.5% respectively. These encouraging results support the fault tolerant design both in terms of the functional programming model and the actual detection and recovery implementation.

A second application implemented on our agent-based system is a parallel Fast Fourier Transform algorithm. This application is designed to test the scalability of the system. It differs from that of the first application (quicksort) in 1) the granularity of each task agent (fine vs. coarse), and 2) the volume of data. We were able to scale up to a 32 million point FFT at which point memory limitations surfaced, most often observed at the nodes that host the higher level (with respect to the hierarchy) agents. On these select nodes, the application exceeded the threshold of available virtual memory (~256M memory + ~128M swap). As data is passed down the agent tree and buffered, memory is eventually exhausted. These results point to a limitation of the current design. A single data entry point (top-level agent) is susceptible to data overload. A straightforward solution using temporary storage violates the fault tolerant nature of the design. A viable, yet challenging solution, is to partition the input data and provide multiple ingestion points in the agent tree hierarchy. This is a subject of current research.

6 Research Direction

We are currently working to enhance our agent infrastructure to provide additional features and broader capabilities. We are exploring the expansion of our model to encompass a more diverse range of applications. The current agent tree structure has limited applicability (it is ideal for divide and conquer algorithms including Wavelets and FFTs). Our goal here is to develop a general-purpose methodology to connect, or “wire together”, task agents. In this context, we are exploring Functional Reactive Programming [2,3] as a methodology for capturing events while adhering to a side-effect free paradigm.

References


