A Scalable and Efficient Peer-to-Peer Run-Time System for a Hardware Independent Software Platform

by Sean Halle

Abstract

This paper introduces a run-time system that is scalable, efficient, and implements an intermediate format for hardware-independent parallel software. It is a peer-to-peer system, having no centralized functions, with peers organized into a tree-of-fully-connected-graphs. The coordination overhead (processing plus number of control messages) grows logarithmically with the number of processing nodes, allowing for scaling to millions of nodes. The run-time load-balances by dynamically dividing data and choosing the peer within a group (fully connected graph) which is best suited to process each piece.

The run-time system implements the computation model of a platform for parallel software called CodeTime[?][?][?][?][?][?][?][?][?][?][?]. This computation model uses declarative scheduling constraints[?]. Declarative constraints allow the run-time's scheduler to dynamically choose the size of data for each unit of scheduled work during a run.

The choice-of-data-size per scheduling event during a run allows the same software, distributed in an intermediate format, to run efficiently on widely varying parallel hardware. Each hardware platform has a custom run-time that chooses sizes such that the computation-to-communication ratio is balanced to overlap communication while maintaining high processor utilization and low scheduler overhead.

Results are given for a run-time written, in Java, for a collection of workstations connected by a local-area network. A test problem was executed on a variety of configurations, and results indicate that scheduling overhead remained a small percentage of total computation time, control overhead increased as the log of the number of processors, and that utilization remained high across configurations.

1 Introduction.

1.1 What is the contribution of this paper?

This paper introduces a run-time system that is scalable and efficient. It is a peer-to-peer system with no centralized functions. The coordination overhead (processing plus number of control messages) grows only logarithmically with the number of processing nodes. It load-balances by dynamically re-sizing data It also lends itself to fault-tolerant implementations.

The run-time system implements the computation model of a platform for parallel software called CodeTime. This computation model uses declarative scheduling constraints, which provides the run-time implementation with the information it needs to adjust parallism granularity to fit the machine and the current load.

1.2 What is the problem we are trying to solve? Writing software for High End Computing requires parallel programming. However, this type of programming has proven difficult. It has been especially difficult to write both correct and high performance programs. Thus, programmer productivity is low, time-to-solution is long, and the number of proficient programmers is small. All of these factors combine to make parallel code expensive to write. Further work must be done to make existing code run on a new machine. The complexity of the code makes this and other maintenance updates also expensive.

1.3 Why does this problem matter? The generation currently under development for all four major lines of microprocessor are either multi-core, multi-threaded, or both. The days of single-threaded processors appear to be over. The number of threads on a chip is expected to double every 18 months⟨cite | ⟩. Therefore, future applications must be flexible in the number of hardware threads they efficiently make use of.

This requires commercial and custom software developers to write parallel programs.

The economic history of the computer industry has shown that the ability to run old software with reasonable efficiency on new machines is a powerful force⟨cite | ⟩. The continued existence of the x86 instruction set is testament to this. Internally, the Pentium processors implement a custom RISC-style instruction set, yet, this inner core is still surrounded by a superfluous shell that translates x86 instructions into this internal RISC format. Processing could possibly be improved by 50% or more by eliminating this shell⟨cite | ⟩. Yet it persists. The reason lies in the economics of software creation and distribution. Customers do not want to buy new software each time they upgrade their machine. Corporate customers do not want to re-write their million-line custom code each time they upgrade hardware.

For parallel software, an equivalent of the x86 instruction set has not been found. Abstract machines have been suggested, but have proven too inefficient in practice.

No viable solution has yet been found which attains these three goals important to industry and science:

1. Hardware independence with high performance
2. High programmer productivity
3. Wide acceptance (enabled by conformance to a standard)

1.4 What makes this problem hard?

Space is limited to do this question justice, but we will attempt to summarize our beliefs. First, we present a framework that we use for thinking about software and computing. Then we use the language of this framework to say why we believe it has been so difficult to attain the 3 goals.

1.4.1 The framework we use to think about software and computing

We view programs, languages and processors as all the same thing. They each consist of a set of commands, a set of constraints on scheduling those commands, and an animator. This view is described in depth in a pair of papers available on the website⟨cite|alg⟩⟨cite|framework⟩.

Commands are names. To apply (execute) a command, the animator looks up the name in the processor's name-space (a formal operational-semantic animator performs lookup by pattern-matching to a rule). The result of lookup is a collection of commands plus constraints. These are commands and constraints of a different processor. They are passed on to that other processor's animator.

An animator chooses data sets, chooses commands to apply to the data sets, looks up the command-name and schedules the lookup-results onto lower-level animators. The lower-level animator(s) then repeats the process of choosing data and scheduling the applications of commands to data-sets, then handing off to yet lower-level animators (basic physics is the base-case animator).

A compiler, in this framework, performs lookup then translates the lookup, from being in terms of one processor's commands and constraints to being in terms of a different processor's commands and constraints. A compiler also checks the input to be sure all the commands are elements of the "from" processor's name-space, and all the stated constraints follow the grammar rules of the "from" processor.

Programs, languages, and processors are distinguished from each other only by practical considerations:

- Languages tend to have simple commands whose lookup-results are easy to translate from one processor to another. They also have constraints which tend to be easy to check automatically.
- Physical processors are themselves languages (ISA: Instruction Set Architecture). Physical processor languages tend to be easy to translate into Register Transfer Language and/or boolean logic.
- Programs tend to have complex commands (each procedure is a command) and define multiple levels of processor (a library is a separate processor).

1.4.2 Our beliefs of why the goals have been difficult to achieve for Parallel Software

We believe that a lack of declarative scheduling constraints to be the heart of why it has proven so difficult to attain the 3 goals of "write once, compile once, run high performance anywhere", high programmer productivity, and wide acceptance.

In order to run high-performance on different hardware platforms, several abilities are needed:

1. the ability to change the computational-complexity of a unit of scheduled work
2. the ability to change the size of data in a unit of scheduled work
3. the ability to predict slack⟨cite|⟩ of a work-unit, in order to stay near critical path
4. the ability to keep scheduling overhead to an acceptable percentage of computation-time

Adjusting computational-complexity and data-size allow tuning the computation-to-communication ratio to fit the network parameters. Choosing computation-to-communication ratio and predicting slack allows both adjusting the frequency of scheduling decisions, and minimizing the idle-time of processors. Knowing the slack allows choosing the work-unit which will, essentially, free the most future-work which is expected soonest (the effect is to stay near the critical path). This minimizes the idle-time of processors. The knee of the mis-prediction-effect curve is when scheduling decisions are made at a rate on the order of the latency of the network. Thus, total run-time of schedulable work-units should be on the order of the latency of the network. This rate of making scheduling decisions also tends to keep scheduling overhead acceptable.

The upshot is that the language must provide the compiler and run-time system the information they need gain the above four abilities.

The information needed is the declarative-form of the scheduling constraints. Without this, only the ability to keep scheduling overhead acceptable is possible.

When the programmer is the only entity possessing the constraints, the programmer is the only entity which can change granularity.

To gain these abilities in an automated tool, the declarative form of the scheduling constraints must be gained by the tool. The constraints may be gained either from the language's semantics, or from performing a search of which hypothesized constraints produce the same results from program-runs.

Various approaches have been taken to parallel software development, with each handling scheduling constraints in a different way:

1. The application programmer implements a custom scheduler. The programmer chooses the complexity of schedulable work-units by implementing the application code in terms of schedulable work-units (which are demarcated by synchronization or equivalent operations). The programmer writes a scheduler which can divide the data-structures of the application and re-assemble the results. The program must be re-compiled for each hardware platform. It must be re-worked to account for OS differences and physical details. The entire scheduler must be re-written when making large changes in size of machine or when significant machine details change, such as network topology or speed. This approach is likely the most difficult of all to program. It requires getting synchronization operations correct, which has proven difficult so far. It also forces the application programmer to understand the hardware well. The programmer knows all of the scheduling constraints, and implements the scheduler to be consistent with them. However, the constraints are not in a form that is available to automated tools, thus only the programmer-chosen complexity and granularity are available. This is the approach of MPI⟨cite | ⟩, actors⟨cite | ⟩, Large Grain Dataflow⟨cite|⟩, and many custom tools from hardware vendors⟨cite|⟩.
2. Add special parallel constructs to a language. The special constructs have known scheduling constraints which allow the language's compiler to choose size of data. This approach is difficult to program because the programmer must try and figure out a way to fit the problem onto the special constructs. It also tends to give limited efficiency on problems that are not highly regular. The programmer has to manipulate the data-structures in order to map the scheduling constraints onto the constraints provided by the language. The language has the constraints available to it for those few data-structures. This is the approach of HPF⟨cite|⟩, parallel pascal⟨cite|⟩, and Sisal⟨cite| ⟩.

3. Make a domain-specific language. The language-designer identifies scheduling constraints that are common to all problems in the domain. They implement the language for each hardware platform, with a custom scheduler. This separates concerns. The hardware expert places their knowledge into the language's custom scheduler. Only a single domain expert is needed to work with the language designer, in order to uncover the scheduling constraints the language designer can use. This approach has a narrow range of problems that each language is useful for. However, it does allow some portability from one type of machine to another. OS issues interfere, as do hardware details, so the application must often be modified and re-compiled, but requires less work than when the custom scheduler is in the application itself. The scheduling constraints are available for use by the tools. This is the approach of CCA⟨cite | ⟩, ⟨cite | ⟩, and skeletons⟨cite | ⟩ which combine this approach and the programmer-implemented custom scheduler approach.
4. Use heuristics and pattern matching to "discover" freer scheduling constraints in the program. Once known, treat them the same way the special constructs are treated in augmented languages. Parallelising compilers do this, with limited success. In general, a full search must be made of all the ways individual constraints can be modified, each way then checked whether the same results are computed by the transformed program. This approach has not yeilded efficient results. It also requires re-compiling the source for each hardware platform. This approach makes the constraints available to the tools by first deducing them. This is the approach of ⟨cite|⟩, ⟨cite|⟩, and ⟨cite|⟩.

1.5 How does our approach overcome these difficulties?

CodeTime languages and computation model (intermediate format) state the scheduling constraints declaratively. Applications declare directly what other approaches either have to deduce or have a limited selection of canned patterns they must map onto.

Constraint languages, evolving from Linda⟨cite | ⟩, appear to allow directly stating scheduling constraints also, but upon closer inspection, they actually only provide a convenient mechanism for implementing a custom scheduler. The scheduling decisions are still implemented via client code as part of the application. The constraints are declared imperatively, via an implementation. A search still must be performed to deduce the underlying constraints.

The CodeTime platform gains all four abilities from declarative scheduling constraints. This allows automated The compiler, which produces executable modules, and the run-time, which implements the dynamic portion of the scheduler, both have knowledge of the constraints themselves. The compiler may use the declared constraints to make static scheduling decisions. The run-time may use the constraints for dynamic scheduling decisions.

This allows the compiler and run-time to be implemented by hardware experts who design a custom scheduler tuned to individual hardware. The custom scheduler is implemented in part in the compiler and in part in the run-time.

The new semantics also appear to help programmer productivity because declared constraints appear to be straight-forward for programmers. Informal discussions with others indicate this; future work is planned to measure the learning curve more objectively.

The rest of the platform is built to fill in the gaps such that the 3 goals can be attained. The new semantics require a new high-level language. Hardware independence requires an intermediate format and a HW-independent OS. Wide-spread acceptance requires a means to ensure conformance to a reference. Each element of the platform has been added for one of these reasons.

A detailed accounting is available on the web-site⟨cite | ⟩ of why each element of the platform was added. In short, the goals create a web of dependencies which force adoption of new semantics that declare the scheduling constraints, which then adds to the force for each of the other platform elements to be adopted.

2 Context and Scope of This Paper

This paper covers only a small segment of a larger platform.

The reader can appreciate that an entire platform, which covers every aspect of software's lifetime, is a large undertaking. In the space allotted for this paper we cannot even begin to address the myriad questions and criticisms which readers new to the CodeTime platform have asked over the years.

We have two choices: either don't share any of these ideas, or else share them in small, targeted parcels, each parcel being too small to answer all but a segment of the questions and criticisms.

We beg the reader's indulgence, putting aside these questions and critical judgements, in order to share with the reader the ideas that are contained in this one, small, paper.

The interested reader can find a list on the sourceforge website of more than 40 questions and criticisms which have been brought up over the years of developing the platform and sharing it with others⟨cite | ⟩. Each has been addressed with a short (1 to 2 pages) paper that is hyperlinked to other papers that delve deeper.

In this paper, the details of a run-time system are presented. The run-time was written to layer over a network of workstations. The same design concepts should work for any hardware consisting of independent processors interconnected by a network.

The interested reader can find other papers on the sourceforge website, each of which covers one aspect of the platform:

1. motivation for defining an entire platform rather than just one piece, such as a language⟨cite|⟩
2. a description of the elements of the platform⟨cite|⟩
3. a framework for understanding the novel semantics in the computation model (alg defn⟨cite | ⟩, lang as processor⟨cite|⟩, circuits⟨cite|⟩)
4. the formal operational semantics of the computation model⟨cite|⟩
5. details of the virtual server and strategies for implementing one⟨cite|⟩
6. details of the OS interface portion of the virtual server⟨cite|⟩
7. a discussion of scheduling strategies in back-end compilers and run-time systems⟨cite|⟩
8. details of how conformance to the reference platform will be maintained⟨cite|⟩
9. a high level language used to write code for the platform⟨cite|⟩

In the first paper we argue that attaining the three goals requres an integrated platform which covers all aspects of the software life-cycle.

The CodeTime platform was designed as such an integrated platform. It has three top-level elements: a virtual server, a family of source languages, and a development environment. The virtual server implements the computation model via a back-end compiler and a run-time system. The back-end compiler takes in a program that has previously been source-compiled down to the computation model. That back-end compiler produces a set of binaries for specific hardware. These binaries are handed to the run-time system. The run-time system forms a software layer directly above the hardware. It implements the portions of the computation model which are dynamic, such as scheduling.

****Section 3 discusses related work, section 4 gives an overview of the run-time system and communication protocol, section 5 discusses the details of this implementation, section 6 gives screen shots produced by this implementation correctly completing one divide-compute-undivide sequence, section 7 suggests improvements to the run-time system, and section 8 restates some important points from the paper.

3 Background on the CodeTime Platform

In order to understand the run-time system presented in this paper, the reader needs an understanding of the CodeTime platform in general, and the Virtual Server in particular.

Here, we, the authors, face a conundrum. The platform is too large, too complex, and too novel to convey an understanding in a single paper suitable for a conference or journal. Our choice is either don't share these ideas, or else share them in small, targeted nuggets. Yet, these small nuggets will not make sense without understanding the larger platform.

The only solution we have thought of is to encourage the reader to iterate. Attempt this paper, then look at the website, then come back, and so on, until the background for this paper starts to come clear.

We have included a brief high-level summary of the platform as the first step in this iteration. This summary will by no means convey any understanding of the platform. However, it will introduce terms and begin to suggest how elements of the platform are related.

****** The CodeTime platform has 3 goals: 1) "write once, compile once, run high performance anywhere" for parallel software, 2) high programmer productivity, and 3) wide acceptance. The platform is built upon a circuit-based computation model whose novel semantics enable attaining the goals. The computation model is implemented by a back-end compiler plus a run-time system which are written for a specific machine. The compiler produces a collection of binaries. When the program is invoked, the binaries are handed to a run-time which schedules them onto physical processors. ******

The CodeTime platform has three top-level elements: a virtual server, a family of source languages, and a development environment.

The virtual server presents the appearance to programs and users of a single machine (a single namespace). The "processor" of this single machine animates mutable circuits. It takes a circuit description, creates an "activation" of that circuit and animates it. The circuit activation is treated as a separate, autonomous system. It interacts by connecting to other systems via a "pin" mechanism. Other systems may be individual files, devices such as the keyboard and display, or external machines such as webservers. Underneath the single-machine abstraction may be any number of physical processors.

The virtual server is implemented separately on each hardware platform. It is intended that the implementation have three main components: an OS interface, a back-end compiler, and a run-time system.

The Virtual Server includes a built-in system for each service a program needs from an OS. These services are meant to be implemented by translating into calls in some underlying OS, such as Linux.

The back-end compiler is written in conjunction with the run-time system. The implementors choose which scheduling decisions they want to implement statically in the compiler, and which they want to implement dynamically in the run-time system.

The back-end compiler then translates the program into a number of executable modules. These modules are used by the run-time system.

To create an activation of the program-circuit, the run-time system loads and initializes the executable modules. The run-time then proceeds to animate that activation. It does this by repeatedly choosing sets of data as inputs to the module-instances then scheduling execution on a particular processor. When a module completes, its results are handed to the run-time. The run-time repeats the process by placing the results in new input-sets and scheduling those.

To run efficiently on different machines with varying numbers and types of physical processors, the platform must provide a means to change the granularity of a computation. The computation-to-communication ratio must be balanced for the given program, processor, and network combination. In addition, the work with the least slack⟨cite | the_critical_path_paper⟩ must be scheduled first, and the scheduling overhead must be kept to a small percentage.

It is intended that the back-end compiler

*****One way to adapt is to give the run-time system a way to dynamically divide data into sizes which fit best on the hardware. One way to accomplish this is by using a divide-compute-undivide pattern, and mapping all application code onto this pattern.

The semantics of the source language and the computation model allow the back-end compiler to accomplish this mapping. The output from the compiler includes a machine-code module which divides data into a set of chunks. The output also includes a machine-code module which takes a number of result-chunks and reassembles them.

The semantics also support combining several circuit elements into a single conglomerate element. The circuit elements are called code-units, and the conglomerate elements are called executable-modules. The back-end compiler chooses which circuit elements to combine according to the details of the hardware.

The run-time system thereby receives a number of executable modules, including ones which divide data, ones which "undivide" data, and a number of which perform steps in the computation. At the end of the execution of each module, control passes back to the run-time system.

The platform supports profile information for the executable modules. This enables the run-time to estimate how long a given size data-chunk will execute in a given module. This is useful when deciding how large each data-chunk should be and which processor it should run on.

This paper gives the details of the implementation of one kind of run-time system. It layers over hardware made up of a number of processors interconnected by a network. High-end computers, networks of workstations, and grid computing are common examples of this hardware model. Multicore processors will progressively fit this model better over the coming years.

4 Overview of the run-time system

**********The control protocol is based upon the principle of speculative execution to increase efficiency. Each child of a node independently calculates the task of every other child in its group. In this way, both sender and receiver are expecting communication without exchanging any control messages, in particular no synchronization protocol is used in this system. Each peer in a group maintains identical state by sending a notice, when it completes a task, to each other member of the group. Thus, actions may be taken by a peer optimistically, in the expectation that its state for the other peers is correct. If a peer receives an unexpected request to receive data, it knows that a difference exists between its state and the sender's, and initiates a resolution protocol.

We discuss the implementation of this run-time system and give results showing its correct operation.
****

The run-time system is a symmetric peer-to-peer system, by which is meant that no services are centralized. All functions of the system may be performed by any of the processing nodes at a given moment. The term peer has meaning, potentially confusingly, in two different senses. In the hardware sense, a peer is a processing-node. In the software sense, a peer is a virtual entity. One or multiple of these may be assigned to a single processing-node. In this paper, processing-nodes are always referred to as processing-nodes, and the term peer always means the virtual-software-entity.

Peers are organized into an overlay network. The overlay network is a tree-graph. Each node in the tree-graph is a peer, and each child of a node is fully connected to each other child of that node. The peers are mapped onto processing-nodes. A one-to-one relationship exists between each processing-node and a leaf-peer. A given processing-node may also have a peer which is a member of any level(s) of the hierarchy above the leaves. If a processing-node has multiple peers, no particular relationship in the tree must exist among that processing-node's peers.

The peers each keep the status of every other peer in its peer-group (the peers it is fully-connected to). Each time a peer completes some unit of work, it sends a message to every other member of its peer-group. When a peer receives a completion message, it updates its status table and decides what action, if any, to take as a result.

This way, all the peers may calculate the action which each other peer should take as a result of every completion message. When a peer receives a request for data-transfer from another, it calculates whether it agrees that this transfer should take place. If it does not agree, it initiates a discrepency-resolution process.

A computation begins when the root peer is handed a set of data to compute the result of. The root chooses one of its children to perform the un-division process. The root broadcasts this choice, along with meta-information about the data, to each child-peer.

All the peers in the group then calculate how to divide up the data, and which piece they should take. Each peer then requests their piece from the data-owner, who could be anywhere in the network. The data-owner responds with that piece of data. The peer then applies the appropriate code-unit to its piece of data. When done, that peer tells all the others in the group, and the parent. All the peers then decide whether this piece allows one of them to collect an input set, and which peer should get it.

If it is possible to gather a complete input set, then each peer which has a piece sends it to the peer that should receive it. When the receiver has them all, it applies the appropriate code-unit to that input-set. When done, it notifies the other peers, which all then repeat the process of deciding whether an input set can be formed and which peer should get it.

When a result comes out of the last code-unit, the peer which produced it sends out a completion message, then sends the result to the decided-upon un-divider peer. The un-divider peer applies the un-divide-unit to the result. When done, it sends out a completion message, then checks whether all the results have been un-divided.

When all the results have been gathered into the un-divider, the un-divider peer sends the combined-result to the parent-peer. The other peers in the group will see the completion message and deduce that the parent has been given the result.

That parent then informs its own peer-group that a result is complete, and things proceed on this level in the same fashion as they did on the leaf-level. The one difference is that the peers above the leaf-level may only delegate computation of executable-modules which contain their own divide-compute-undivide pattern. Not all possible executable-modules will have this. Thus, the above-peers have fewer control-parallelism choices available to them. For this reason, the executable-modules which the compiler makes for the higher-level peers will usually be different than the executable-modules it makes for the leaf-level peers. This maximizes the available control-parallelism at the leaf level.

When the root receives the result from the un-divider it delegated, it in turn hands that result to whatever is attached to the appropriate output pin of the program-package. The details of interaction between the root of the run-time system and the rest of the virtual-server are available on the SourceForge site: http://codetime.sourceforge.net *****

5 Implementation

This section describes the details of the peers which were coded and run. The next section shows output from a test run.

5.1 Composition of a peer

Each peer is composed of four pieces:

- A Server, which accepts all messages to the peer
- A Client, which sends all messages from the peer
- A Scheduler, which tracks the status of all the peers in the group and decides what action each of them, including itself, sould take in response to each incoming message.
- A Worker, which applies executable-modules to sets of data. The scheduler collects each set of data, then hands that set to the worker, along with which executable-module to apply.

The worker has no persistent state. What the scheduler gives the worker contains everything needed to perform the computation. This is an important property. It is one of the reasons the run-time is able to divide data into smaller pieces so easily.

A peer also contains three blocking queues to connect the pieces. The queues are used one-directionally. One goes from the server to the scheduler, one from the scheduler to the worker, the first is re-used to go back from the worker to the scheduler, and finally one from scheduler to client.

5.1.1 The Server

The server is a standard TCP/IP server, which listens on a port for clients attempting to connect. This implementation has all the peers on the same processing-node, so they all have "localhost" as their host name. Different peers are identified by the port they listen on.

The server and client communicate serialized RTMsg objects. When one is received, the server places it on the queue to the scheduler.

5.1.2 The Client

The client takes message objects off the queue from the scheduler, takes the "to" port out of it, establishes a connection to the server listening on that port, and sends the object.

5.1.3 The Scheduler

The scheduler takes messages off the queue going to it and decides what action to take as a result. It implements all of the communication protocol logic, and all of the decision making about which peer should perform which operation. This decision making which assigns actions to peers is a scheduling algorithm.

The communication protocol implements what's unique to the CodeTime computation model.

The scheduling algorithm is independently choosable. A wide variety of different algorithms may be plugged in. This implementation uses an "dummy" algorithm which simply divides the incoming data evenly among all the peers in the group. When a peer acts as a parent, it randomly assigns one of the peers below it to be the undivider.

Thus, this scheduler does not take into account the status of the other peers in the group, and so the discrepency resolution process is not yet implemented.

The scheduler picks which of the executable modules should be applied to a piece of data. In the version implemented, only three executable-modules exist: the divider, the "main" work-unit, and the undivider. The scheduler picks the divider in response to a "here's a chunk to divide up" (chainReqDataTakeMsgFromAbove) message from the parent. It picks the main work unit in response to a "here's the piece you requested"(respDataTakeMsgFromAny) from the owner of the chunk. It picks the undivider in response to a "here's a result to be undivided"(respDataTakeMsgFromSame) from one of the peers in the group.

5.1.4 The Worker

The worker models the compiler-generated executable modules by having a method for each of the three units. The scheduler puts into the message it gives to the worker which of the units should be applied. The worker then executes the appropriate method, giving it the data information from the incoming message (reqWorkMsgFromSelf). The data is modelled as simply an integer, stating the size of the data. When the method completes, it hands back the result-size, which the worker puts into a response message (respWorkMsgFromSelf) that it then places into the queue going back to the scheduler.

5.2 The communication protocol

The communication between run-times is also one-directional. Clients inside the peer objects only send messages, servers only receive. This organisation mirrors the asynchronous nature of the computation model. This asynchrony of the computation model is also an important property. It is a reason that no synchronization operations are needed among peers, making it profitable to take actions speculatively.

5.2.1 The sequence of messages

The sequence goes:

- → Parent sends: ReqChainDataTakeMsgFromAbove To: all peers in the group below it
  - Each peer independently calculates how to divide up the data. The msg contains meta-info about the data, but not the actual data itself. In this impl, the data size is just divided by the number of peers.
- → In response, Peers each send: ReqChainDataGiveMsgFromAny To: the owner of the data
  - In this impl, no data actually moves. In a real impl, all requests would go back through the chain. One of the higher-level peers in the chain would track the requests, to ensure that no requests overlap, and all the data in the original piece is requested.
- → In response, data owner (the parent in this case) sends: RespDataTakeMsgFromAny To: requestor
  - The data-owner just makes a new message with the same size. In the real system, it would place the actual data into this response message.

→ In resp, each peer sends: ReqWorkMsgFromSelf To: worker

- The worker applies the executable-module chosen by the scheduler to the data specified in the message.

→ In resp, worker sends: RespWorkMsgFromSelf To: scheduler

- In this impl, it just sends back the size of the data.

→ In resp, peer sends: InfoComplMsgFromSame To: each other peer in group

- When a peer receives this message, it only updates its copy of the status of the peers in the group (including its own status). The status will be used in real implementations to independently decide how to divide up data, and which peer should receive each completed input-set.

→ The peer also sends: InfoComplMsgFromBelow To: the parent

- The parent also keeps the status of every peer in its child-group. It uses this to pick an undivider. (Other method might be letting the group pick the undivider). It also keeps the parent informed about the progress of the computation. In this impl, no status is kept, the parent picks an undivider at random.

→ The peer also sends: RespDataTakeMsgFromSame To: the undivider (if peer is not undiv)

- When a peer produces a result from the last executable-module, it sends that result to the un-divider. It first checks whether it is itself the un-divider.

→ The peers which are not the undivider are now done sending messages in response to the original from the parent.

→ Meanwhile, each time the undivider receives a piece it sends: ReqWorkMsgFromSelf To: worker

- The worker applies the un-divider-unit to the result.

→ In resp, worker sends: RespWorkMsgFromSelf To: scheduler

- This is the same sequence as when one of the executable-modules completes, except that the worker checks whether this is the last piece to be undivided (based upon info in the message)

→ In resp, peer sends: InfoComplMsgFromSame To: each other peer (says an undivider done)

→ The peer also sends: InfoComplMsgFromBelow To: the parent (says an undivider done)

→ When the last piece is undivided, the peer sends the same info messages as before, then also sends: RespDataTakeMsgFromBelow, which completes the sequence of steps.

- The parent gets this message, then treats it as if the parent itself had completed applying a executable-module to the data. Higher-level peers may only apply executable-modules which contain a divde-compute-undivide pattern inside them. Leaf-peers, however, may apply any arbitrary executable unit to data, because leaf peers do not sub-divide the data.

6 Results

Results were gathered on a system with four peers, all running on the same machine. Each peer listens on a different port. One peer is a parent, three are in a peer-group below that parent. The parent generates a canned "divide up this data" message (ReqChainDataTakeMsgFromAbove). It picks a random member of the peer-group to be the undivider, and sends all the peers the same message.

The two figures show the output from two of the peers: the parent and the chosen undivider. It shows the initial message going out from the parent, the activity of the messages between the group members, and the final "here's the result" coming back to the parent. (messages 2 and 3 in the peer at 9901 are debug messages)


Sched of Peer at Port: 9901
Message: 1
Got Msg: [DestPort:9901] [Request Chain | data | take | from Level above]
[ID: 1 | whoFrom: 9903]] [Undivider Port: 9901] [data size: 633] [null]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
Message: 2
Handling Msg: [DestPort:9901] [Request Chain | data | take | from Level above]
[ID: 1 | whoFrom: 9903]] [Undivider Port: 9901] [data size: 633] [null]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
Message: 3
To Client Msg: [DestPort:9903] [Request Chain | data | give | from any Level]
[ID: 0 | whoFrom: 9901]] [Undivider Port: 9901] [data size: 211] [null]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
Message: 4
Got Msg: [DestPort:9901] [Response | data | take | from any Level]
[ID: 5 | whoFrom: 9903]] [Undivider Port: 9901] [data size: 211] [null]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
Message: 5
Got Msg: [DestPort:0] [Response | work | take | from own worker]
[ID: 1 | whoFrom: 9901]] [Undivider Port: 9901] [data size: 211] [Main Work Unit]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
main dashed unit was completed
Message: 6
Got Msg: [DestPort:9901] [Info | completion | take | from same Level]
[ID: 2 | whoFrom: 9900]] [Undivider Port: 9901] [data size: 211] [Main Work Unit]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
Message: 7
Got Msg: [DestPort:9901] [Response | data | take | from same Level]
[ID: 5 | whoFrom: 9900]] [Undivider Port: 9901] [data size: 211] [Main Work Unit]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
Message: 8
Got Msg: [DestPort:9901] [Info | completion | take | from same Level]
[ID: 3 | whoFrom: 9902]] [Undivider Port: 9901] [data size: 211] [Main Work Unit]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
Message: 9
Got Msg: [DestPort:9901] [Response | data | take | from same Level]
[ID: 5 | whoFrom: 9902]] [Undivider Port: 9901] [data size: 211] [Main Work Unit]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
Message: 10
Got Msg: [DestPort:0] [Response | work | take | from own worker]
[ID: 5 | whoFrom: 9901]] [Undivider Port: 9901] [data size: 211] [Un divider Unit]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
an undivider unit was completed
Message: 11
Got Msg: [DestPort:0] [Response | work | take | from own worker]
[ID: 6 | whoFrom: 9901]] [Undivider Port: 9901] [data size: 422] [Un divider Unit]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
an undivider unit was completed
Message: 12
Got Msg: [DestPort:0] [Response | work | take | from own worker]
[ID: 7 | whoFrom: 9901]] [Undivider Port: 9901] [data size: 633] [Un divider Unit]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
an undivider unit was completed
done undividing
last chunk, package up data

Figure 1.


Sched of Parent at Port: 9903
Message: 1
Sending to all: Msg: [DestPort:0] [Request Chain | data | take | from Level above]
[ID: 0 | whoFrom: 9903]] [Undivider Port: 9901] [data size: 633] [null]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
Message: 2
Got Msg: [DestPort:9903] [Request Chain | data | give | from any Level]
[ID: 0 | whoFrom: 9902]] [Undivider Port: 9901] [data size: 211] [null]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
Message: 3
Got Msg: [DestPort:9903] [Request Chain | data | give | from any Level]
[ID: 0 | whoFrom: 9900]] [Undivider Port: 9901] [data size: 211] [null]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
Message: 4
Got Msg: [DestPort:9903] [Request Chain | data | give | from any Level]
[ID: 0 | whoFrom: 9901]] [Undivider Port: 9901] [data size: 211] [null]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
Message: 5
Got Msg: [DestPort:9903] [Info | completion | take | from Level below]
[ID: 4 | whoFrom: 9900]] [Undivider Port: 9901] [data size: 211] [Main Work Unit]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
Message: 6
Got Msg: [DestPort:9903] [Info | completion | take | from Level below]
[ID: 4 | whoFrom: 9901]] [Undivider Port: 9901] [data size: 211] [Main Work Unit]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
Message: 7
Got Msg: [DestPort:9903] [Info | completion | take | from Level below]
[ID: 4 | whoFrom: 9902]] [Undivider Port: 9901] [data size: 211] [Main Work Unit]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
Message: 8
Got Msg: [DestPort:9903] [Info | completion | take | from Level below]
[ID: 10 | whoFrom: 9901]] [Undivider Port: 9901] [data size: 211] [Un divider Unit]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
Message: 9
Got Msg: [DestPort:9903] [Info | completion | take | from Level below]
[ID: 13 | whoFrom: 9901]] [Undivider Port: 9901] [data size: 422] [Un divider Unit]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
Message: 10
Got Msg: [DestPort:9903] [Info | completion | take | from Level below]
[ID: 16 | whoFrom: 9901]] [Undivider Port: 9901] [data size: 633] [Un divider Unit]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
Message: 11
Got Msg: [DestPort:9903] [Response | data | take | from Level below]
[ID: 17 | whoFrom: 9901]] [Undivider Port: 9901] [data size: 633] [Un divider Unit]
[data owner's port: 9903] [original Msg's ID: 0 | orig who from: 9903]
Handling response from undivider
Which means the test succeeded.

Figure 2.

7 Improvements

7.1 Performance Improvements

The astute reader will have noticed that no comparison has been made between this run-time and any competing technology. The authors estimate that this run-time would take 5 to 10 times longer than an MPI implementation of the same test application. Work is in progress to implement the execution modules as compiled C code which are called through Java's native interface. The report on this work will include head-to-head performance comparisons.

7.2 Leaving pieces of larger chunks distributed across processing nodes

One improvement is to allow the data-chunks to remain divided, when convenient. In this scheme, the data-chunks passed among higher-level peers would consist of lists of smaller chunks. The division of these chunks would involve dividing up the lists. When such a chunk reaches a leaf-peer, that peer requests one or a few of the elements of the list from peers holding the actual data. For example, if all leaf-peer chunks were the same size, this may be simple enough to easily implement.

However, this does complicate the division and undivision process. In some cases, when nested loops exist in the undivision-unit, for example, it may be advantageous to collect the pieces onto the undivider peer and construct a single result-piece. In this way, multiple sizes of data pieces may be included in a single chunk.

Peers asking for the wrong piece must still be detected. This may be accomplished, for example, by requiring that any division of a given list of data-pieces must have at least one higher-level peer which all requests for the data-pieces pass through. The division algorithm would then decide and specify which higher-level peer all piece-requests must pass through (in deterministic replicatable way).

This higher-level peer then gets the original list of pieces and tracks each request, to ensure that all peers in a group came to the same conclusion about how to divide up the data.

Allowing the data-pieces, which make up a larger chunk, to stay on the leaves in this way may improve performance. The performance gains would come from reducing redundant transfers of data. If the larger chunks are physically constructed, then the pieces must be sent to the undivider machine. Now, if this chunk is used in another input-set, it has to be divided up again. Each leaf-level peer getting a piece will then request it from the undivider. Thus the actual data moves twice, once to the undivider, and once again to the new leaf-level peer. On the other hand, with undividers which don't require the contents of the data, only meta-information about it, only a single transfer of the actual data takes place, at the time the new leaf-level peer requests it.

7.3 Data retention for reliability, correctness, and failure detection

To prevent the loss of partial results due to the failure of a processing node, some scheme may be used which retains "ancestors" of results until suitable downstream results are completed on distinct processing-nodes. The idea is to have at least two nodes with a "recoverable" form of the data at all times. Thus, a processing-node keeps data until it is confirmed that two other processing nodes now have some form that the most recent results, on the third node, can be reproduced from the results on the second node.

The communication involved in this protocol would detect lost messages eventually, and allow recovering from machines going down.

Recovery is possible if an invariant is maintained. The invariant says that if a descendant, call it A, shares a processing-node with any ancestors of a second descendant, call it B, then A does not contribute towards the reproducibility of B. Only when a set can be formed of descendants which have the reprodicibility property for all lower-descedants of a given ancestor, can that ancestor be safely deleted. This invariant is easier to calculate than might be guessed, because the flow-graph is fixed. The back-end compiler may generate, statically, a function. At run-time, this function takes in a list of descendant result-pieces and processing-nodes they reside on and returns a yes or no answer for a given ancestor (more analysis of this idea is needed).

A simple way of saying this is that its a sort of worm-hole routing, where the tail of the worm dissapears as the head makes progress. The complication is that the heads of many worms combine together. To show the correctness of this method, it must be shown that the combined-head may be reproduced from the middle-links. As long as enough middle links are on different processing-nodes from the com-

bined-head, the combined head may be lost. The middle-links are then used to reproduce it on another processing node.

7.4 Delaying actions to increase their prediction accuracy

The utility of each action can be optimized by estimating prediction accuracy and modelling the cost of misprediction. Tracking the statistics of flight times between peers can be used to estimate prediction accuracy. A given peer can then simply act as though a particular message took longer to get there before taking action implied by that message. In the meantime, any update messages with previous time-stamps have a chance to arrive. The amount of time to wait between receiving a message and taking action is determined by optimizing the utility of waiting.

The chance of misprediction decreases the longer a peer waits, and can be estimated using the flight-time statistics.

The cost of misprediction also changes with time, and depends upon the type of message and the called-for action. The cost of mispredicting a data-request action decreases with time because less network capacity is used when the transfer time decreases, assuming the data transfer is still in progress when the discrepancy is detected, also the collision rate decreases when fewer non-useful packets flow, which increases the effective bandwidth for the correct transfers. The cost of mispredicting a computation action, however, is zero if it does not interfere with other, higher utility computation actions.

Making the computation queue in the worker a priority queue, which chooses the highest utility computation next implements this. Older messages naturally gain higher utility due to their improved prediction rate, which prevents starvation.

7.5 Redundancy of higher-level peers

For the sake of reliability, possibly performance and somewhat security, each higher-level peer may be implemented with redundancy. In this case, it would be duplicated on several processing nodes, and the duplicates participate in a voting scheme. The degree of replication would increase as the level in the hierarchy increased, with root being the most widely replicated.

The increased reliability is self-evident.

Performance improvement would acrue in a large system, with hundreds of thousands of nodes, from selecting a quorum of processing nodes close, in a network latency measure, to each other and to the data under consideration. The replicated peer would also have higher throughput, due to multiple quorums operating simultaneously on different processing-nodes.

The increased security would come from the ability to detect the processing nodes whose peers consistenly give wrong answers. For example, this would catch virally infected nodes which were not involved in a coordinated attack. The ability to detect misbehaving processing nodes may also serve as an element in a more sophisticated security scheme.

8 Conclusion

This paper has shown the implementation details of a run-time system for the CodeTime platform. Three properties of the CodeTime computation model have been used: self-contained chunks of work, stateless workers, and an asynchronous computation model.

Self-contained chunks of work, which carry their scheduling bookeeping with them, enabled generic scheduler plug-ins to be written by the programmer. The plug-ins take data in a generic carrier along with how many pieces to divide the data into. The plug-ins then extract the information they need from the data itself to perform the division. This divide-compute-undivide pattern let the size of data-chunk be changed dynamically.

This advances the goal of “write once, compile once, run high performance anywhere” by allowing one program to be written with generic plug-ins. That same plug-in is used on all hardware un-changed. On given hardware, the scheduler written for that hardware decides during a run the most efficient size for that hardware, and invokes the generic plug-in to perform that division.

This paper also showed how the asynchronous computation model lets the run-time system produce correct results without synchronization operations. This was seen to enhance performance on hardware on which the synchronization operation is expensive, such as memory barriers on large machines with tens or hundreds of thousands of processors, and networked machines which must synchronize with control messages.

Finally, this paper showed how the self-contained data, plus stateless workers plus asynchronous model made the speculative method shown here attractive. The speculative method allowed scaling to very large systems, due to the small number of control messages, and it allowed optimizing the computation by choosing the highest utility computation at each moment, which this paper has shown should outperform, on average, a scheme which waits for certainty (an ack or a lock), as well as enabling self-healing machines.