# Dataflow Java: Implicitly Parallel Java

Gareth Lee and John Morris

Centre for Intelligent Information Processing Systems,
Department of Electrical and Electronic Engineering,
The University of Western Australia,
Nedlands WA 6907, Australia
E-mail: `[gareth,morris]@ee.uwa.edu.au`

## Abstract

*Dataflow computation models enable simpler and more efficient management of the memory hierarchy - a key barrier to the performance of many parallel programs.*

*This paper describes a dataflow language based on Java. Use of the dataflow model enables a programmer to generate parallel programs without explicit directions for message passing, work allocation and synchronisation. A small handful of additional syntactic constructs are required. A pre-processor is used to convert Dataflow Java programs to standard portable Java.*

*The underlying run-time system was easy to implement using Java's object modelling and communications primitives. Although raw performance lags behind an equivalent C-based system, we were able to demonstrate useful speedups in a heterogeneous environment, thus amply illustrating the potential power of the Dataflow Java approach to use all machines - of whatever type - that might be available on a network .. when Java JIT compiler technology matures.*

## 1 Introduction

Dataflow computation models have two major advantages: (a) they are conceptually simple and (b) they expose *all* the parallelism available in a problem with ease. Since Dennis' early work[1], interest in dataflow systems has been generated machines capable of executing dataflow programmes directly[2, 3, 4, 5] and a number of languages, both textual[6, 7] and visual [8]. The dataflow model is often described as *data-driven* to distinguish it from the *control-driven* model of the majority of languages in use today. The key rule for a dataflow computation is the *firing rule*:

> a computation executes (fires) when all of its data becomes available.

### 1.1 A Dataflow Java?

The Internet now links vast numbers of machines with a wide variety of architectures and capabilities. Java's portability makes it extremely suitable as a common language in a heterogeneous computing environment[9]. Thus, Java is an ideal base for a parallel language that can exploit the computing power of large numbers of different, often idle, machines accessible on networks. The Java designers have also prescribed extensive security features for Java run time systems, thus making it suitable for running large distributed programs on networks of systems with varying ownership expectations or security concerns[10].

Java already provides multi-threaded capabilities and the ability to invoke methods on remote systems and parallel processing systems can be built using these capabilities[1], but programmers must explicitly generate threads and invoke methods on remote machines. Dataflow Java is designed to provide a simple language in which as much of the detail of the management of the parallel processing is handled by the run-time system (RTS). The programmer uses a small set of extensions to Java to indicate portions of a computation which are suitable for parallel processing: a pre-processor transforms this into standard Java which is then compiled and run on any suitable system. A key aim of this work was to design a language which was implicitly parallel by exploiting the dataflow firing rule. Programmers should not need to explicitly program communication, barriers and other synchronisation primitives or direct where a computation is to be carried out: the RTS is much better equipped to do that - especially in a heterogeneous environ-

[1]In fact, Dataflow Java uses many of them - but such use is transparent to the programmer.

ment of multi-user machines, where the capacity of any particular machine may change as unrelated workloads come and go. By providing a simple model in which the Dataflow Java RTS always knows whether a thread is ready to run or not and, furthermore, can choose to run a ready thread on any suitable machine, we not only relieve the programmer of much housekeeping, but potentially achieve significant speed-up through optimal use of resources.

Dataflow Java's design has been strongly influenced by the original Cilk[7, 11]. The ease of porting Cilk to a new system was a key factor. We found that setting up Dataflow Java - in which all of the runtime system was written from scratch in Java 2 (JDK 1.2) - was particularly easy; we were able to exploit a number of Java features to simplify the task. In particular, Cilk's closures mapped almost trivially to Java objects and no special code for memory management was required. Java's memory management facilitates handling of temporary and shared objects, so that although, in general, memory management in parallel systems tends to be complex and error-prone, using Java eases the programmer's task also.

In the NoW environment, the ability to build fault-tolerance into the runtime system[12] without burdening the programmer was seen as a particular advantage.

Section 2 of this paper describes the general structure and semantics of a Dataflow Java program. The Dataflow Java RTS and its management of parallelism and load balancing are discussed in section 3. Each Dataflow Java extension to Java is described in the following section. The fault tolerance capabilities - essential for efficient operation in a dispersed heterogeneous environment are discussed in section 5. The results of a simple experiment, which provide a simple demonstration of Dataflow Java's ability to operate efficiently in a heterogeneous environment, are set out in section 6. Some related work is reviewed in 7 and section 8 summarises the benefits of Dataflow Java and identifies some further capabilities that we are planning to add to the system to further enhance its heterogeneous multi-processor performance.

## 2 Dataflow Java

Dataflow Java is an extension of Java which is pre-processed into standard Java and then compiled and run by any suitable Java system. To design a Dataflow Java program, a programmer sketches out a *side-effect free* dataflow graph identifying suitable computation threads, data required by each and the source of that data.

Dataflow Java threads are Java classes which extend the `DJFrame` class, although the programmer doesn't need to worry about this. He or she simply writes a thread to perform each computation in the dataflow graph. Threads are declared with the `thread` keyword: a prototype for a Dataflow Java thread to evaluate function `f` looks like this:

```
thread f {
  // data
  <data for thread f>
  // methods
  // main computation of the thread
  <type> f() {
    <type> x;
    ....
    return x;
    }
  }
```

Threads must provide a method with the same name as the thread itself: this method contains the main computation of the thread - it will be invoked by the Dataflow Java RTS when a thread is *full*, *i.e.* has all its data and is ready to run. Note that Dataflow Java threads are quite different from standard Java threads: they are run from the Dataflow Java RTS - not the Java runtime system. Java threads also run in a shared memory model, whereas our threads carry within each thread *frame* (*cf.* section 3) all the context for a thread. Thus our threads are entirely compatible with - and easy to implement in - a distributed memory environment.

Figure 1 shows a simple[2] program illustrating the features of Dataflow Java.

Note that the data required for the thread is declared at the head of the thread (just as the attributes of a Java class - in fact the Dataflow Java pre-processor converts data items declared here to class attributes). The key difference between an ordinary Java class and a Dataflow Java thread with respect to the attributes is that, following the dataflow semantics, a Dataflow Java thread must have values for all data items declared at the head of the thread before the thread is able to be run.

Thread data can be supplied in one of two ways:

- when the thread is created: a list of named associations is used to copy data from the creation thread to the thread being created,
- after the thread has been created: by 'posting' data to the thread.

In both cases, the dataflow firing rule applies, if *all* the data that the thread requires has been supplied, then the thread is ready to run and will be posted for execution on some processor by the Dataflow Java RTS.

Examples of the two methods are:

*Supplying data at thread creation*

[2]We apologise for inflicting this program - that no graduate of an elementary algorithms course would own - on our readers yet again, but it seems to provide the simplest illustration and enables direct comparison with other work[13].

```
thread fib {
  // datum on which execu-
tion is conditional
  int n;

  int fib( ) {
    int x, y; // thread locals
    if (n < 2) {
      x = 1;
    }
    else {
      // Create two thread and sup-
ply their data
      x = fib( n => n-1 );
      y = fib( n => n-2 );
      // When both threads have com-
pleted,
      // sum their return values
      x = x + y;
    }
    return x;
  }
}
```

**Figure 1. Dataflow Java Fibonacci numbers**

```
  x = fib( n => n-1 );
```

causes a new `fib` thread to be created, setting its datum `n` to the value `n`-1. Since this particular thread has only one datum, `n`, it will be posted for execution immediately. Values for thread data are supplied by a list of named associations - as in Ada[14].

*Supplying data at some later stage*

A thread may post data to any thread at some later stage. This will commonly occur when a 'final' thread is created to gather and combine results from a number of computations which can run in parallel, *e.g.* the sorting of partitions of a data set which are then merged in a final phase.

This may be illustrated by writing the Fibonacci program in Figure 1 in a slightly different way:

```
(1)  x = fib() in g;
(2)  y = fib() in h;
     // .. some additional computa-
tion could
     //    be inserted here
(3)  g( n => n-1 );
(4)  h( n => n-2 );
(5)  x = x + y;
```

In lines 1 and 2, threads are created without the datum `n` and so are not ready to run. Since data will need to be posted to the thread when it is finally computed, these threads must be named: the `in g` and `in h` directives associate the names `g` and `h` with the two threads. At lines 3 and 4, values for `n` are posted to `g` and `h`, which, following the dataflow semantics, become ready to run and are posted for execution by the Dataflow Java RTS. The Dataflow Java pre-processor detects the dependency of line 5 on the results returned from threads `g` and `h` and suspends execution of this thread until the results are available. When the thread is created with all its data - and thus ready to run immediately - the programmer does not need to refer to it again and it need not be given a name.

A more elaborate version of the simple Fibonacci program in figure 1 illustrating more Dataflow Java features is shown later in figure 2.

The pre-processor converts `thread`'s to *frames* - specialisations of the `DJFrame` class. Each thread is an instance of one of these specialisations: when full, it contains all the data which the thread needs to run and thus can be migrated to any suitable processing element (PE) - subject to any constraints specified in an `at pe(..)` clause (*cf.* section 4.3) - for execution. The pre-processor also creates *continuation* objects which enable a thread to post data back to its parent or siblings.

## 3 Dataflow Java RTS

After it has passed through the pre-processor - which flags Dataflow Java syntax errors and some Java errors, a Dataflow Java program is compiled with the classes which comprise the Dataflow Java RTS. The program is run by invoking the RTS on each PE. On one PE - designated the *master* - an additional argument specifying the entry frame is supplied. The entry frame has a static `main` method (much as Java applications) which creates an instance of the frame and posts it for execution by the RTS. The thread in the entry frame is thus executed first. The remaining PEs are termed *slaves*, although once the entry thread has been started, all PEs are essentially identical. However, for convenience, most programs will post their final results back to the entry thread or some other thread executing on the master PE.

The RTS manages the system startup, frame queues, the running of threads, the transmission of results to frames on other PEs and the orderly termination of all processing when the final frame has been executed. It consists of several communicating Java threads which

- invokes the handlers for incoming messages,
- examines the ready queue for full frames,
- runs threads and

- initiates work stealing from another PE when there is nothing else to do.

### 3.1 Program distribution

A copy of the basic Dataflow Java RTS is started on all PEs. One of the PEs becomes the master by being given the entry thread to run. Slaves simply activate their scheduler threads and start work-stealing.

To minimise load on the networks, Java byte codes are class loaded from the master dynamically, *i.e.* only Dataflow Java threads that are needed by a remote processor are loaded over the network. However, the master processor broadcasts requested classes to all participating slaves to avoid multiple loads of the same class.

### 3.2 Parallelism and Work distribution

Dataflow Java programs achieve parallelism without the need for explicit directions to distribute work because, once all its data has been posted, a thread has all the context that it needs for execution and can be run on any PE.

The Dataflow Java RTS uses ‘idle-initiated’ work-stealing[15]: an idle processor (the *thief*) will send a work-stealing request to another PE (the *target*). If the target has work (threads which have all their data and are ready to run), then it will serialise the frame representing the ready thread and send it back to the thief. The thief runs the stolen thread and sends its result back to the thread’s parent on the target.

If this thread was passed references to other threads waiting for data, then it may also send results to those threads: messages are sent to the original PE, causing *setter* methods (similar to those defined for Java Beans) to be invoked in threads waiting for data. If an invocation of a `setter` method causes a thread to become full, it is posted to a ready queue. The Dataflow Java RTS will select threads from the ready queue for execution when any currently running thread completes. When multiple threads are ready to run, the RTS selects the deepest one first in order to minimise memory requirements.

Since the cost of migrating a thread is significant, a target will send the largest possible block of work to a thief. To determine the most suitable block of work: we use the simple method which has been shown to work effectively in the Cilk RTS. Threads are assigned an increasing level as they are spawned: a work-stealing request will receive the thread with the lowest possible level - as it is very likely to spawn additional threads. These threads will generally be run on the thief and, in a typical program, result in large amounts of local (and inexpensive) communication between threads on the thief - avoiding more expensive network communication.

### 3.3 ‘Full’ and ‘Empty’ threads

A thread maintains a *join counter* - a count of the number of items of data required before the thread becomes ready to execute under the dataflow firing rule. A join counter value of 0 indicates a ready thread. When created, threads may be either full (all data supplied by named associations in the thread creation statement; join counter set to 0) or empty (an empty or partial list of data supplied with the thread creation statement; join counter $> 0$). Empty threads acquire data through post operations. The RTS uses *continuations* (created transparently by the Dataflow Java pre-processor) to address empty data items in threads. Thus the statement

```
f( x => 100.0 );
```

uses a continuation pointing to the empty ‘slot’ for `x` in thread `f` to post the value `100.0` to `x`, decrementing the join counter and - possibly - making the thread ready to run, which will cause the RTS to move it to a ready queue.

### 3.4 Explicit Load Balancing

In many problems, optimum or near-optimum load-balancing strategies are easily discovered. Dataflow Java allows a programmer to specify the processor on which a thread will be run:

```
z = t( a=>a_val, b=>b_val ) in g at pe(k);
```

which creates a new thread, `g`, of type `t` which will be run on a PE identified by evaluating the function `pe`. The present implementation of the `pe` function takes a single integer argument - the index of the target processor. In line with our aim to provide a language well matched to a heterogeneous processing environment, this function will be extended to take an arbitrary list of desired processor attributes (*cf.* section 4.4).

Values `a_val` and `b_val` are supplied for data items `a` and `b` respectively. Any additional data required to fire `g` will be provided by `g(`*`x`*` => `*`val_x`*`);` statements. The RTS will keep track of the fact that `g` was posted to PE `k` and despatch the data accordingly. The Dataflow Java RTS will provide the target PE with a *continuation* pointing back to the current PE.

### 3.5 Communication

The Dataflow Java RTS manages all communication between PEs - relieving the programmer of the need to explicitly program any communication events.

In most workgroup configurations, network bandwidth limits computational throughput. Thus using knowledge of communication requirements to minimise messages sent is a key factor in obtaining performance. Since the Dataflow Java RTS knows the reliability requirements of each message, it uses a light-weight communications protocol (the standard datagram protocol, UDP) for all messages. We plan to study further reduction of network load, *e.g.* by compressing messages.

Every message sent to a remote processor invokes a handler on that processor: in our implementation, a handler identifier is sent in the head of the message. The remote receiver extracts the message from the I/O buffer and jumps directly to the handler code.

## 4 The Dataflow Java Language

In this section, we provide an abbreviated reference for Dataflow Java's extensions to Java. Symbols which are not elaborated in this discussion use standard Java syntax, *e.g.* $\langle$statements$\rangle$ is a list of Java statements (but including Dataflow Java statements) and $\langle$exp$\rangle$ is a valid Java expression. Brackets (`[]`) indicate optional items.

### 4.1 Threads

A thread has the form:

```
thread <name> {
  <thread data>
  <constants>
  // computational thread
  <type> <name>( [ <arg list> ] ) {
     <local data>
     <statements>
     return <exp>;
     }

[<private methods> ]
}
```

A thread is converted to a Java class (extending the `DJFrame` class): the pre-processor supplies a constructor. The method with the same name as the class is actually the main computation thread (the pre-processor re-names it `init` so that the default `DJFrame` method is over-ridden. We adopted this convention to make dataflow programs easier to read: the programmer uses only one name to refer to a thread.

When the pre-processor detects a dependence on a result returned by a thread spawned within another thread, it creates a `succ` thread containing the dependent statement and the remainder of the thread code. Spawned threads posts their results to the `succ` thread, firing it when all required data has been posted.

The definition of the computation method may include a list of formal parameters: these are essentially the same as the parameters for any Java method - actual parameter must be specified at thread creation time. These parameters should be distinguished from thread data, which is supplied - partially or fully - at thread creation time via a list of named associations.

Threads may have as many private methods as required, but they will marked `private` by the pre-processor so that attempts to invoke them from outside the class - possibly violating the dataflow semantics - are trapped by compilers. Additionally, we require that statements within a thread may not block (*i.e.* by calling `wait` or `sleep`): a future pre-processor will enforce this!

### 4.2 Thread data

The thread data is a list of standard Java variable declarations with optional initial values. Values for each of these data items must be supplied - either at initiation or by subsequent posting operations - before the thread is ready to run. Data with initial values may be overridden at thread initiation, but if a value is not supplied at initiation, the default value will be used and a subsequent post to this value is considered an error.

A thread's data or formal parameter list may contain a reference to another thread - enabling one thread to post data back to an ancestor or sibling.

### 4.3 Creating Threads

Creating a thread uses (deliberately) a syntax similar to function invocation:

```
  x = [ call ] t( <args> [<data>] )
          [ in <name> ] [ at pe( <exp> ) ];
```

`t` is the name of a thread known to the pre-processor through `import` statements. The optional `call` keyword directs the preprocessor to invoke the thread code sequentially, *i.e.* without creating a separate thread. It is not essential, but judicious use will avoid the creation of large numbers of fine-grain threads (*cf.* the Fibonacci example in figure 2) and result in a considerable performance improvement in, for example, recursive divide-and-conquer algorithms.

`<args>` is a list of actual parameters corresponding to formal parameters in the method definiton. Normal positional association is used.

`<data>` is a list of named associations between thread data items and values, *e.g.*

```
  x = t( a => 10.0, b => p*q );
```

Named association allows any subset of the thread data to be supplied - and in any order - at thread creation time. The

data list may be empty, in which case all the thread's data will be supplied by later post operations.
The `in` keyword is followed by a name for this thread. `in` is needed if the data list is incomplete, *i.e.* later computation - perhaps in other threads - will supply the data. It is optional if the thread is created 'full'.

### 4.4 Explicit Work Distribution

The `at pe(<exp>)` directive is not needed if the programmer is willing to allow the work-stealing mechanisms to determine work distribution. However, if work-distribution patterns are known, then it is slightly more efficient to direct work to individual PEs immediately rather than let the work-stealing mechanisms work out what you already know! In the current implementation, the `pe(..)` function takes a single integer argument - the index of a target PE. However, we are planning to implement a version in which the expression supplied to the `pe` function gives some suggestions to the RTS about the capabilities of an ideal PE to run this thread. For example, for a particularly long computation, one will write:

```
z = t( a=>a_val, b=>b_val ) in g
                    at pe( tcyc<3 );
```

to direct the Dataflow Java RTS to attempt to find a processor with a cycle time of 3ns or less.
A user will be able to supply his or her own `pe()` function which takes a string argument. Thus a user will be able to formulate advice appropriate to the local environment. The Dataflow Java pre-processor will pass the expression supplied by the programmer to the `pe()` function which will parse the string and determine, using local rules, the best PE to run this thread. This advice may be used to request a particular capability (*e.g.* a particular graphics display) or indicate that the thread is computationally intense and should be run on a fast, lightly loaded PE.

### 4.5 Posting data

When a thread is created 'empty', it will wait until the missing data is supplied by post operations:

```
t( <data> );
```

In this example, `t` is the name of the thread (defined by an `in t` clause when the thread was created) and `<data>` is a list of named associations. As with thread creation, this list may be complete or partial, *i.e.* there may be multiple post operations before a thread becomes ready to execute. A thread is a first class data type in Dataflow Java (the pre-processor converts it to a class), so a thread identifier may be passed to another thread which may compute the required values and post them.

```
// create a thread without data
x = f() in g;
// create a thread to compute one of
// the data values
y = h( c_thread => g );
// compute another value
z = ...
g( arg1 => z );
// use the result from f
return x;
```

In this case, thread `h` will contain a statement:

```
w = ...
c_thread( arg2 => w );
```

Note that the pre-processor will detect the use of the value (`x`) produced by thread `g` in the `return x;` statement and suspend the main thread until

- `h` has computed `w` and posted it to `arg2` of thread `g`,
- the main thread has computed `z` and posted it to `g`,
- both of these posts have been completed, 'firing' `g` and enabling it to run and post a value back to `x` in the main thread.

The pre-processor actually creates a *successor thread* (`succ`) containing the `return` statement and fires this thread when `x` is posted to it.

## 5 Fault Tolerance

Any system which aims to exploit the idle cycles typically available on desktop machines by using a set of geographically distributed processors with owners who have different priorities from those running parallel programs needs to have some tolerance to 'faults' such as disconnection, reboot, re-configuration, physical relocation, *etc.*. Many of these factors are not true faults, but consequences of the physical and ownership distribution. Systems will usually have independent mechanisms (error correcting memory, CRC words on messages, *etc.*) which will enable corrupted signals to be detected. Thus an effective system - one which will continue to proceed *correctly* to a final result - can be built if it is able to tolerate 'fail-stop' faults, *i.e.* those that arise from, for example, complete removal of a machine from the system (its owner decided to relocate it) or complete loss of a message (a network link was temporarily broken). Loss of messages includes decisions to simply drop messages which are corrupted in transit.
The dataflow semantics of Dataflow Java permit the same fault-tolerant mechanisms used in FT-Cilk[12] to be used here. For example, whenever a frame migrates to another

processor, the original PE keeps a *ghost* - a copy of the migrated frame. If the remote processor successfully executes the thread, it will return a result to the original PE: the ghost will then be freed. If however,

- the migration message is lost or corrupted
- the remote PE fails while executing the thread or
- the result is lost or corrupted,

the original processor will eventually become idle and start examining its ghost lists. If it discovers that the remote processor is still executing a migrated frame for which a ghost is being kept, it will check further ghosts or attempt to steal work from another processor. If however, the remote processor doesn't respond to a query, it will be assumed to have failed, and the ghost will be executed on the local PE. There's a possibility that the remote PE actually ran the thread, and posted results that it produced - it was the result message that was lost. This will mean that the thread contained in the migrated frame will be run twice. Although this will result in some duplicated and unnecessary computation, the functional nature of Dataflow Java programs ensures that the second execution of the thread produces the same results as the first, so that the final result is still correct. Other scenarios which fail-stop faults generate are discussed by Morris and LeFevre[12].

## 6 Results

We have deferred extensive performance evaluation of Dataflow Java because current Java compiler technology leaves Dataflow Java programs in the shadow of our C-based Cilk system - at least as far as performance is concerned. Our assessments of available 'Just-in-time' (JIT) Java compilers which are able to optimise remotely loaded Java byte-code programs shows a distinct gap between their performance and that of optiimised native C compilers, but we see this gap shrinking fairly rapidly as analysis techniques mature.
For the present, we have concentrated on demonstrating that Dataflow Java will operate in a heterogeneous environment. All current Cilk systems use absolute addresses within a program's memory space as handler addresses and thus assume clusters of machines which can run the same binary image. However, Dataflow Java remotely loads Java byte-code when it is needed, so that Dataflow Java programs are oblivious of the architecture of the processor running any particular thread.
Table 1 shows results obtained from running the program of figure 2 on a small network of workstations each running a different implementation of the Java 2 virtual machine under a different operating system. Other than running the experiments at a 'quiet' time, no attempt was made to isolate the system used for the experiments from the twenty or so other machines attached to the same sub-net. Measurements of the performance of the Java run-time systems on the individual machines produced raw performance figures (shown as executed Dataflow Java closures per second in table 1) that varied by a factor of slightly more than 2 due to variations in the performance of the underlying virtual machine, so the final speed-up figures (final column of table 1) were adjusted to reflect this.
The raw results show useful decreases in the total processing time as different processors were added to the cluster. Furthermore, the adjusted speedup factors are within 20% of the ideal - a good result with fast processors on a relatively slow (10Mbit ethernet) network.

## 7 Related work

The side-effect free nature of most functional languages means that they have the implicit parallelism that we have exploited in Dataflow Java. Dataflow languages from Dennis' original diagrams[1] to more complex visual languages such as NL[8] and the text-based ones such as Id[16], SISAL and pH[17] belong to the class of functional languages. All of these languages were designed to enable a programmer to extract the implicit parallelism present in a problem without requiring the programmer to be concerned with messages, synchronisations and data allocation. Dataflow Java is an implementation of these ideas using Java's excellent portability to permit operation in a heterogeneous environment.
Baldeschwieler *et al.*[13] have also combined ideas from Cilk and Java in their Atlas proposal, but their proposed syntax exposes the mechanisms by which Atlas works to a far greater degree than is proposed here. In particular, they burden the programmer with a `SyncVar` type, retain the `spawn` directive and add a further `spawnSuccessor` function.
Numerous projects to design parallel processing capabilities into Java have been spawned by Java's instant success: these range from simply using the in-built remote method invocation capabilities to using optimising compilers to identify instruction-level parallelism[18]. These systems make no or negligible changes to the basic language. Several parallel extensions to Java have been proposed - including interfaces to PVM and other parallel communication libraries[19]. At least one has been based on functional language ideas[20].

## 8 Conclusion

Dataflow Java provides some simple extensions to Java that enable a programmer to simply and effectively write programs that can execute on large, heterogeneous collections

| Machine Name | CPU | OS | Raw Speed (closures/s) | Scale Factor | Time seconds | Speedup (Raw) | Speedup (Adj) |
|---|---|---|---|---|---|---|---|
| asp | Pentium MMX 166 | NT 4.0 | 288 | 1.00 | 201 | 1.00 | 1.00 |
| mulga | Pentium Pro 200 | Windows 98 | 302 | 0.98 | 102 | 1.97 | 1.92 |
| bardick | Pentium Pro 200 | Solaris 2.6 | 184 | 1.11 | 80 | 2.51 | 2.80 |
| gwardir | Pentium Pro 200 | Linux | 145 | 1.25 | 76 | 2.66 | 3.33 |

**Table 1. Performance of a heterogeneous cluster running the `fib` program as processors were added.**

Processors were added in the order shown: figures in the Time column are for systems with 1, 2, 3 and 4 processors going down the column.

of work-stations. Our programmers only need to understand the *dataflow firing rule* to write programs: they are not burdened with the need to worry about messages, synchronisations, barriers and similar trappings of other parallel programming systems. We believe that the system presented here is an effective, yet easy-to-use, framework for distributed heterogeneous processing. We have deferred attempts to measure performance as any such measurements would currently be overshadowed by our C-based Cilk system. However, Java compilers are maturing rapidly and will soon be within acceptable factors of their older C counterparts. Then the ability to use *all* our processors with no special effort will make performance experiments worthwhile! The extensible `pe( .. )` function, which guides the Dataflow Java RTS in the selection of suitable target processors for a particular thread, will enable users to tailor the system to resources available on their networks. This will provide a simple, yet effective method for Dataflow Java users to efficiently use a variety of resources in a heterogeneous processor environment.

## References

[1] J B Dennis, "First version of a data flow procedure language," in *Programming Symposium: Proceedings, Colloque sur la Programmation (LNCS, vol 19)*, 1974, pp. 362–376.

[2] G. M. Papadopoulos and D. E. Culler, "Monsoon: An Explicit Token Store Architecture," in *Proceedings of the 17th International Symposium on Computer Architecture, Seattle, WA*, May 1990.

[3] K. R. Traub, G. M. Papadopoulos, M. J. Beckerle, J. E. Hicks, and J. Young, "Overview of the Monsoon Project," in *Proceedings of the 1991 IEEE International Conference on Computer Design*, Oct. 1991.

[4] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba, "An Architecture of a Dataflow Single Chip Processor," *Proceedings of the 16th Annual International Symposium on Computer Architecture, Jerusalem, Israel*, pp. 46–53, May 1989.

[5] Mitsuhisa Sato, Yuetsu Kodama, Shuichi Sakai, Yoshinori Yamaguchi, and Yasuhito Koumura, "Thread-based Programming for the EM-4 Hybrid Dataflow Machine," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, *Gold Coast*, *Australia*, May 1992, pp. 146–155.

[6] John Feo, "Sisal," *??*, 1700.

[7] Michael Halbherr, Yuli Zhou, and Chris Joerg, "MIMD-Style Parallel Programming based on Continuation-Passing Threads," Tech. Rep. 387, MIT Laboratory for Computer Science, 1994.

[8] Nicole Harvey and John Morris, "Nl: A parallel programming visual language," *Australian Computer Journal*, vol. 28, pp. 2–12, 1996.

[9] James Gosling, Bill Joy, and Guy Steele, *The Java[tm] Language Specification*, Sun Microsystems, 1996.

[10] Li Gong, *The Java[tm] Security Model and Architecture*, Sun Microsystems, 1998.

[11] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: an efficient multithreaded runtime system," *ACM SIGPLAN Notices*, vol. 30, no. 8, pp. 207–215, 1995.

[12] John Morris and Neil LeFevre, "Natural Fault Tolerance in Functional Languages," in *Proceedings of the Australian Computer Science Conference, Sydney*, Feb 1997, pp. 364–372.

[13] J. Eric Baldeschwieler, Robert D. Blumofe, and Eric A. Brewer, "Atlas: An infrastructure for global computing," in *Proceedings of the Seventh ACM SIGOPS European Workshop: Systems Support for Worldwide Applications*, Connemara, Ireland, Sep 1996.

[14] US Department of Defense, *Reference Manual for the Ada Programming Language: ANSI/MIL-STD-1815A-1983*, ANSI, 1983.

[15] Robert D. Blumofe and Charles E. Leiserson, "Space-efficient scheduling of multithreaded computations," in *Proceedings of the Twenty-Fifth Annual ACM Symposium on the Theory of Computing (STOC '93)*, San Diego, CA, USA, May 1993, pp. 362–371, Also submitted to SIAM Journal on Computing.

[16] Rishiyur S. Nikhil, "Id Reference Manual, Version 90.1," Tech. Rep. 284-2, Sept. 1990.

[17] Arvind, R.S Nikhil, and Lennart Augustsson, "pH," Tech. Rep., 1993.

[18] Aart J. C. Bik, J. E. Villacis, and Dennis B. Gannon, "javar: a prototype java restructuring compiler," *Concurrency: Practice and Experience*, vol. 9, pp. 1181–1191, 1997.

[19] Ian Foster and S Tuecke, "Enabling technologies for web-based ubiquitous supercomputing," in *Proceedings of the 5th Intl Symp on High Perf Dist Computing*, Syracuse, New York, 1996.

[20] Gary Meehan, "Compiling functional programs to java byte-code," Research Report CS-RR-334, Department of Computer Science, University of Warwick, Coventry, UK, Sept. 1997.

```
package ciips.dataflow;

thread fib {
  // user defined constant
  final int CONST = 24;

  // datum on which execu-
tion is conditional
  int n;

  int fib( int limit ) {
    int x, y; // thread locals

    if (n < limit) {
      x = fibr(n-1) + fibr(n-2);
    }
    else {
      // Create a thread and sup-
ply its data
      x = fib(limit, n => n-1 );
      // Create a thread without data
      y = fib(limit) in g;
      ...
      // Supply the data later
      g( n => n-2 );
      // When both threads have com-
pleted,
      // sum their return values
      x = x + y;
    }
    return x;
  }

  // a user defined local method
  int fibr(int n) {
    if (n < 2)
      return n;
    else
      return fibr(n-1) + fibr(n-2);
  }
}
```

**Figure 2. A more elaborate Fibonacci number program**

To manage thread granularity, a local function, `fibr`, is called sequentially when the `limit` is reached in the recursion.