CL-MW

A Master-Slave Library for Distributed Programming in Common Lisp
Version 0.1

Peter Keller
psilord@cs.wisc.edu

November 2, 2010

Contents

License

The source code to CL-MW, the example programs, and the documentation source and contents are under the Apache 2 license:

Chapter 1

Overview

1.1 Background

Master-Slave is a distributed computing paradigm where a master process partitions work to one or more slave processes, collects the results, and produces the required end solution from those partial results. The work is usually of a fine to medium grained nature. The work request, called a task, and the slave's response, called a result, do not individually require large amounts of disk space, memory space, or network resources. The tasks may be ordered or unordered. Ordered tasks must be assigned to a slave which may additionally hold persistent information. Unordered tasks run on any available slave and are often the only kind of tasks supported by Master-Slave systems. Examples of problems applicable to the Master-Slave paradigm are: numerical optimization, exploring game search trees, web-crawling, and Monte-Carlo simulations.

In many Master-Slave execution environments, it is an unavoidable reality that execute nodes asynchronously connect and unexpectedly disconnect from the computation. This is called slave churn. Many Master-Slave systems accept this reality and provide task scheduling algorithms such that when a slave disconnects, times out, or otherwise is deemed unusable, tasks assigned to that slave are recycled back into the task pool to be distributed out to a different slave at a later time. If the master process exits prematurely, then the slaves all die as soon as they notice, as to not consume computing resources.

Master-Slave systems may be coupled with an external batch scheduling system. The batch system can manage resource acquisitions for the slave processes and provide an environment to restart the computation in the face of machine or other environmental failure.

1.2 CL-MW

CL-MW is a Master-Slave implementation written in, and for, Common Lisp. The design of the library's API was designed for ease of use and rapid prototyping of Master-Slave applications. The library decouples management of the task/result flow through the slaves from the act of spawning slaves to simplify interaction with pre-existing batch systems. CL-MW has three main parts to be specified by the application author: one or more *task algorithm*s, a *master algorithm*, and optionally a *slave algorithm*. CL-MW implements a single binary executable containing both the master and slave code.

The *master algorithm* produces tasks and inserts them into CL-MW. Tasks are data packets destined for a specific *task algorithm* (and potentially a specific slave as well) and which are stored in the master process. *Task algorithm*s are pieces of code in the slave which process the tasks into results which are sent back to the master process for consumption by the *master algorithm*. CL-MW maintains a pool of tasks running and waiting to be run. While all of the tasks can be created during the *master algorithm* initialization, tasks can also be added dynamically. Tasks may be dynamically added based on results from earlier tasks, or they may be dynamically added to limit memory used to store them. Tasks have meta-data associated with them that can dictate where or in what manner the task should be processed. This is known as a *task policy*.

An optional *slave algorithm* allows arbitrary computation to happen in the slave in between processing one or more task. An example of such a slave computation is downloading a database upon slave startup which is used by the *task algorithm*s and then removing it when the slave shuts down.

1.2.1 Task Algorithms

A *task algorithm* is a piece of code written by the application author which converts tasks into results in the slave. The macro `define-mw-algorithm` defines a *task algorithm*. The parameter list of this macro is similar to `defun`. You specify a name and an argument list. Unlike `defun`, `&key`, `&rest`, and `&optional` arguments are currently not supported. The supplied code cannot return a set of values with `values`. There must only be one value that is returned. These limitations will be removed in future versions of CL-MW.

Here is an example definition of a *task algorithm* which echos back its argument unchanged. Note: The arguments accepted and the result returned are that which you would have done had the *task algorithm* been defined with `defun`.

Listing 1.1: The `echo` Task Algorithm

```
(define-mw-algorithm echo (val)
  val)
```

The result of expanding `define-mw-algorithm` is a collection of functions and a macro as defined in table 1.1.

Symbol	**Kind**
`echo`	Function
`mw-funcall-echo`	Macro
`mw-set-target-number-for-echo`	Function
`mw-get-target-number-for-echo`	Function
`mw-pending-tasks-for-echo`	Function
`mw-upto-target-number-echo`	Function

Table 1.1: Expansion of `define-mw-algorithm` for the `echo` *task algorithm*.

The functions and macro created in expansion of the *echo task algorithm* are grouped into three sets: the *echo* function itself–which is the body of the `define-mw-algorithm` macro, the task creation macro `mw-funcall-echo`, and a set of functions which allow one to manage how many pending tasks for this *task algorithm* are queued.

The `mw-funcall-echo` Macro

The macro `mw-funcall-echo` creates a new task and separates the arguments to the *echo* function–ultimately called with those arguments in the slave process, from the *task policy* associated with the task.

The signature of `mw-funcall-echo` is:

```
(mw-funcall-echo (str)
                  (&key sid tag do-it-anyway (retry t)))
```

This example call of `mw-funcall-echo` shows the creation of a new *echo* task with an argument of `"Hello World"`. All task arguments to the *task algorithm* must occur within the first set of parentheses and in the order specified for the specific *task algorithm*'s parameter list. Processing this task will result in a result structure given back to the *master algorithm* which contains the echoed string `"Hello World"`.

```
(mw-funcall-echo ("Hello World"))
```

Task Policy

The *task policy* associated with a task describes *how* and *where* a task should be executed. The *task policy* for a task is defined when a task is created via the `my-funcall-*` macro. Part of the default *task policy* for a task to be considered unordered and to run on any available slave. Another part of the default is that if the task running on an arbitrary slave—which then disconnects without providing an answer, the task is reassigned to a different slave for processing and this can happen many times. The full default *task policy* is specified on page 36 as well as what each portion of the policy means.

Through the *task policy*, one may assign tasks to run on previously acquired ordered slaves. These ordered tasks will be run in the order inserted by the *master algorithm*. The *task policy* is directly responsible for a ordered task possibly becoming *unrunnable*. This happens when the ordered slave which is processing the task disconnects. The default *task policy* for ordered tasks is that any tasks being processed on the disconnected slave *or queued waiting to run on the slave* become unrunnable and the task structures are given back to the *master algorithm*.

This example call of `mw-funcall-echo` is the same as above with respect to the task generated and the result expected. However, when the result structure associated with this task is presented to the *master algorithm*, the result will have in it the associated tag of `1234`. The tag of a task may be any Lisp form.

```
(mw-funcall-echo ("Hello World") :tag 1234)
```

Task Algorithm Target Numbers

Target numbers are values recorded by CL-MW and set by the *master algorithm* which represent the number of pending to run tasks for a *task algorithm* that should be kept in memory at all times. It is useful when a task generator in the *master algorithm* can produce many more tasks than can fit into the master process's memory or disk on the resource where the master process is running.

An analogy to the CL-MW concept of target numbers is the temperature setting of a thermostat. If one sets a thermostat to 70°F then when the temperature falls below that, the furnace kicks in and injects heat into the room until the target number is reached. As the furnace heats up the room it may overheat it, but it generally shouldn't since the goal is to keep the temperature stable at the thermostat's setting.

In the same manner as the furnace, the *master algorithm* can use the target number for a *task algorithm* to create the required number of tasks (which could be zero if no new tasks are needed) into CL-MW until the target number is reached. The *master algorithm* can run millions or billions of tasks through without having to have all of them in existence at once. The tasks may be lazily generated as needed.

The target numbers themselves have no behavior on how CL-MW processes the tasks or enforces restricting the number of created tasks. All target numbers do are provide a means so that the *master algorithm* can police itself when creating tasks. The master is free to create more tasks than specified in a target number without restriction (other than running out of memory or other resources in the process).

Continuing the example of the *echo task algorithm*, here is a description of the signatures and meaning of this set of generated functions:

`(mw-set-target-number-for-echo` *`value`*`)` `Function`

> A target number of *echo* tasks the *master algorithm* would like to keep in CL-MW. Initially 0.

`(mw-get-target-number-for-echo)` `Function`

> Return the number of in memory tasks that exist for this *task algorithm*.

`(mw-pending-tasks-for-echo)` `Function`

Return the number of tasks (both currently running and pending to run) for this *task algorithm* that are currently known about by CL-MW.

`(mw-upto-target-number-echo)` Function

A number of tasks which must be created by the *master algorithm* in order to reach the target number for this *task algorithm*. This function could return zero if the return value of `mw-get-target-number-for-echo` is equal to or less than the return value of `mw-pending-tasks-for-echo`.

The General Target Number

If an application author wishes to manage only the total number of created tasks in memory independent of which *task algorithm* they represent, then they can use the general target number API as defined in section 5.2 on page 37. The number of tasks *up to* the general target number is the general target number minus all pending tasks from any *task algorithm*.

1.2.2 The Master Algorithm

The macro `define-mw-master` defines the *master algorithm* for a CL-MW application. There is only be one *master algorithm* per application. The *master algorithm* is responsible for:

- Parsing non-CL-MW command line arguments passed to the application process
- Partitioning the main problem into sub-problems and creating tasks
- Acquiring and managing ordered slaves
- Calling the *master algorithm*'s event loop function
- Processing the results returned by the slaves
- Determining what to do when some tasks become unrunnable
- Computing the final answer of the application from all results

The parameter list of this macro is:

```
(define-mw-master (argv) &body body)
```

where `argv` is a variable–arbitrarily named and available in the `body`, which will be bound to the command line argument list. Any arguments destined to the CL-MW library will have been removed from the list before the *master algorithm* is invoked. The arguments not stripped out are left in the order specified on the command line. The return value of the *master algorithm* must be an integer in the range of 0 and 255 (inclusive) and this value becomes the Unix return value for the master process. If the return value is any other Lisp form other than an integer between 0 and 255, the returned result will be 255.

Driving the CL-MW master event loop is one of the main functions of the *master algorithm*. The *master algorithm* accomplishes this by calling the function `mw-master-loop` (or a variant of this function–see page 38). This function blocks and performs network I/O to all slaves or any other background work in the CL-MW library. This function returns with a set of values that specify what is available for the *master algorithm* to process (such as new results, arrival or disconnection of ordered slaves, etc) only when there is some event for the *master algorithm* to process.

Slave Categorization

The *master algorithm* can use the function `mw-allocate-slaves` to categorize connecting slaves into the groups: `:ordered`, `:intermingle`, and `:unordered`. A connecting slave is first used to satisfy the needs of the group `:ordered`, then `:intermingle`. If enough slaves connect as to satisfy the needs for both the `:ordered` and `:intermingle` groups, then they are placed into the `:unordered` group. Initially the `:ordered` and `:intermingle` groups need 0 slaves and all slaves will default to being placed in the `:unordered` group.

The `:ordered` group means that the slave will *only* run ordered tasks dedicated to that slave. The `:intermingle` group means that a slave may run unordered tasks in addition to ordered tasks dedicated to that slave but the ordered tasks are given priority over any unordered tasks which could run on that slave. Slaves in the `:unordered` group only run unordered tasks.

When `:intermingle` or `:ordered` slaves are needed and new slaves placed into those groups, the `mw-master-loop` notifies the *master algorithm* (or variant–see page 38) that there are ordered slaves ready for use. This notification happens with one of the values returned by `mw-master-loop`. The CL-MW application author can use the function `mw-get-connected-ordered-slaves` to retrieve the list of connected slaves.

The function `mw-get-connected-ordered-slaves` returns a list of SLAVE-IDs which can be used as the `:sid` field with the macro `mw-funcall-*`. If no ready ordered slaves are available, then `NIL` is returned. Connected ordered slaves accumulate in that list until the *master algorithm* uses `mw-get-connected-ordered-slave` to retrieve them. If at any time ordered slaves disconnect, the function `mw-master-loop` will notify the *master algorithm* of the change via one of its returned values. The *master algorithm* can use the function `mw-get-disconnected-ordered-slaves` to learn the SLAVE-IDs of the disconnected ordered slaves. A slave may be in both lists at once if it connected and then disconnected before the *master algorithm* was able to retrieve either of the lists. There is no notification when an `:unordered` slave connects or disconnects.

The *master algorithm* can free a slave from the `:ordered` or `:intermingle` groups be using `mw-deallocate-slaves` and `mw-free-slave`. These will move ordered slaves to the `:unordered` group but only after they have completed processing any assigned `:ordered` tasks. <u>`Note`</u>`:` When a freed ordered slave finishes processing the tasks assigned, it will move to the `:unordered` group even though there may actually be more tasks destined for that slave. In this case, the tasks will follow the task policy dictated by the *master algorithm* when it created the task.

Membership

The membership token–an arbitrary string, is a token known between the master and the slave. It must match for the master to accept the slave and have it perform work. This is not a security measure. This keeps the master and slave processes synchronized in heavy churn situations where many masters and slaves from different computations could be going up and down quickly. The major risk in high master churn situations is port reuse of the master process. A port may have master 1 bind to it, write a resource file, die, then later master 2 from a different computation binds to the port, meanwhile a slave using the original master 1 resource file tries to connect to master 1 but actually connects to master 2. Membership tokens must be unique across CL-MW application master processes running concurrently on one machine.

Unless otherwise specified with the `--mw-resource-file` *filename* command line option, the membership token will default to `"default-member-id"`.

1.2.3 The Slave Algorithm

The *slave algorithm* is defined with `define-mw-slave`. The parameter list of this macro is similar to `define-mw-master`. This macro must return an integer from 0 to 255 inclusive. This portion of a CL-MW application is optional and may be left out entirely in an application.

```
(define-mw-slave (argv) &body body)
```

The body of a *slave algorithm* is usually a simple call to `mw-slave-loop-simple`. `mw-slave-loop-simple` will wait for tasks to arrive from the master, process them, send the results back, and will repeat until the master sends a shutdown command. `mw-slave-loop-simple` returns 0 if the shutdown was explicitly requested by the master and happened normally or 255 otherwise.

There are other slave looping function variants which allow the slave loop function to return after a single, or group, of tasks is finished. These variants are used when the slave needs to set up or tear down some files while it is working or otherwise manipulate the environment around it in-between processing tasks. Please see page 41 for these other variants.

If no slave algorithm is specified in a CL-MW application, then this default *slave algorithm* is automatically defined and used.

Listing 1.2: Default Slave Algorithm

```
(define-mw-slave (argv)
  (mw-slave-loop-simple))
```

1.2.4 Running a CL-MW Application

A CL-MW application can be executed in two ways: interactively at a REPL or as a dumped binary from the command line. When running in the REPL, there should be a REPL for the master process and one each for the slave processes. It is not recommended to start different threads where one is the master and the rest are slaves. From the REPL, the CL-MW entry point is the function `mw-initialize` which takes a list of strings that represent the command line arguments of the application. Running in the REPL is useful for debugging or incremental development.

The recommended means of doing production runs with a CL-MW application is via a dumped binary created with the CL-MW function `mw-dump-exec`. When this function is called, the entire Lisp image will be written into a binary and the Lisp image will exit and any shared libraries which `mw-dump-exec` finds as being local to the installation to SBCL or any used CL libraries will be written to the current working directory. The entry point will be an implementation specific function which does some bookkeeping and then invokes `mw-initialize` with the command line arguments supplied. In this form, the binary acts like any other client/server application and you can easily run as many as you need. If for some reason the process gets an uncaught signal or other terminating error, a stack trace created by SBCL's runtime will be emitted from the program to facilitate debugging.

CL-MW has a collection of settings which are adjusted via command line options as described on page 33.

1.2.5 Network I/O and Task/Result Size

The underlying network implementation of CL-MW is nonblocking and fully asynchronous. A connection to a client is handled by a packet buffer that is split into two pieces: a read buffer and a write buffer. The initial size of each buffer is controllable. The read buffer can grow to a specified maximum size before the connection is cut to the other side on account of the packet being too big. The write buffer size is advisory at this time and only limit how much can be written at one time instead of how large the write buffer actually can be. This will be addressed in a future revision of CL-MW.

The master process will internally group tasks into a whole network packet and subsequently tell the slave how many results to group into the network packet back to the master. The grouping of tasks and results amortizes the cost of sending data over TCP and increases network utilization efficiency–at the cost of memory, of network communication. It is up to the application author to understand enough of their task and result size requirements to pick good groupings so grouped tasks or results don't overflow the packet buffer sizes. Understanding the scale of how many slaves will be connecting to the master process will determine how big to make the initial network buffer sizes and to what they should be capped as they grow.

Chapter 2

Downloading and Installing

2.1 Compatibility and Versioning

CL-MW will be considered in beta until it reaches the 1.0 version number. During this phase, the APIs or feature sets of CL-MW may change in a non-compatible ways with previous versions of CL-MW. Such compatibility or feature changes will be detailed in the version history section of this document located in appendix B on page 49.

2.2 Supported Implementations

CL-MW is currently supported on/with:

- SBCL 1.0.39.16 or later.
- Stable releases of IOLib such as 0.7.0, 0.7.1, 0.7.2, 0.7.3 or later.

2.3 Official Release Tarballs

This web page contains all official releases of CL-MW in addition to a live git repository of the HEAD of CL-MW.

`http://pages.cs.wisc.edu/~psilord/lisp-public/index.html`

Manuals in PDF and HTML forms corresponding to each release exist next to the release tarball.

2.4 Installation Using a Tarball

1. Unpack the tarball, e.g., `tar zxf cl-mw-0.1.tar.gz`.
2. Configure SBCL to know about the `*.asd` files. This may entail making symlinks in your `$HOME/.sbcl/systems` (or appropriate) directory to the `*.asd` files in the CL-MW directory.
3. In the unpacked CL-MW directory:
   - `make` will make all of the examples.
   - `make clean` will remove all generated files.
   - `make docs` will make the PDF and html manual output in `doc/`.
   - `make SBCL=/path/to/sbcl` *`target`* will use a specific sbcl installation instead of the one in your path. By using this mechanism, you can specify things like `clbuild lisp` if that is how you start SBCL. If you specify SBCL for the toplevel Makefile, it will propagate to the Makefiles in the `examples/*` directory

Chapter 3

Writing Applications

A CL-MW application uses the `:CL-MW` package and exists in its own arbitrarily named package determined by the application author. There exist three parts to a CL-MW application: one or more *task algorithm*s, a single *master algorithm*, and a single *slave algorithm*.

3.1 Example: Hello World

The purpose of this minimal example is to show how to create a *task algorithm*, a *master algorithm*, and a *slave algorithm*. The *master algorithm* will create tasks and process the results from one or more slaves which connect to the master process. The *task algorithm* we describe simply concatenates the string arguments with another string and returns it. Both the master and the slave processes are assumed to be on the same machine with both binding to the localhost interface.

We start with the unsurprising ASDF file for the hello-world CL-MW application.

Listing 3.1: `cl-mw.examples.hello-world.asd`

```
(asdf:defsystem #:cl-mw.examples.hello-world
  :depends-on (#:alexandria #:cl-mw)
  :components ( (:file "package")
                (:file "hello-world"
                       :depends-on ("package"))))
```

For this next listing we see that `mw-master` and `mw-slave` are functions which are used for testing or debugging in the REPL. Notice we re-export the `:CL-MW` package symbol `mw-dump-exec` from our application package which helps us easily save the lisp image into a binary at a later time.

Listing 3.2: `package.lisp`

```
(defpackage #:cl-mw.examples.hello-world
  (:use #:cl #:alexandria #:cl-mw)
  (:export #:mw-master
           #:mw-slave
           #:mw-dump-exec ))

(in-package :cl-mw.examples.hello-world)
```

For documentation purposes, we partition the main single file of the implementation into parts which contain the *task algorithm*, the *master algorithm*, and the *slave algorithm*.

The *task algorithm* accepts a regular Lisp string and also returns one.

Listing 3.3: `hello-world.lisp:  Part 1 of 4`

```
(in-package :cl-mw.examples.hello-world)

(define-mw-algorithm hello (str)
  (concatenate 'string "Hello World: " str))
```

The *master algorithm* creates 10 tasks into CL-MW and then continues to call `mw-master-loop` until 10 results have been processed. When `mw-master-loop` returns, one or more of these CL-MW functions will return meaningful data, depending upon the application: `mw-get-unrunnable-tasks`, `mw-get-results`, `mw-get-connected-ordered-slaves`, `mw-get-disconnected-ordered-slaves`.

Listing 3.4: `hello-world.lisp:  Part 2 of 4`

```
(in-package :cl-mw.examples.hello-world)
(define-mw-master (argv)
    (unwind-protect
         (let ((num-tasks 10)
               (num-results 0))
           (dotimes (x num-tasks)
             (let ((str (format nil "Task ~A" x)))
                (mw-funcall-hello (str))))
            (while (/= num-results num-tasks)
              (mw-master-loop)
              (when-let ((results (mw-get-results)))
                (dolist (result results)
                  (let ((payload (mw-result-packet result)))
                    (incf num-results)
                    (format t "Got result from slave: ~S~%"
                            payload)))))
            0)
       (format t "Master algo cleanup form.~%")))
```

The *slave algorithm* is very simple in our case. The function `mw-slave-loop-simple` simply loops inside of CL-MW processing tasks until the master tells the slave to shut down, at which point `mw-slave-loop-simple` returns 0 and the slave exits with that return code. We could have left off this definition of a slave algorithm altogether and used the default slave algorithm in a CL-MW application. We included it here as demonstration of how to write one.

Listing 3.5: `hello-world.lisp: Part 3 of 4`

```
(define-mw-slave (argv)
  (unwind-protect
       (mw-slave-loop-simple)
    (format t "Slave algo cleanup form.~%")))
```

We additionally specify two helper functions which are not part of CL-MW, nor technically the application, but allow us ease of debugging and testing the application in the REPL.

Listing 3.6: `hello-world.lisp: Part 4 of 4`

```
(defun mw-master ()
  (mw-initialize
    '("--mw-master" "--mw-slave-task-group" "10"
                    "--mw-master-host" "localhost"
                    "--mw-slave-result-group" "10")))

(defun mw-slave (port)
  (mw-initialize
    `("--mw-slave" "--mw-master-host" "localhost"
                    "--mw-master-port"
                    ,(format nil "~D" port))))
```

3.2 Running Hello-World in the REPL

Now, let's run this example in the REPL so we can see how it works. First, we'll set up and run the master process. We call our master helper function to start the master process. We're packaging together 10 tasks to an idle slave and expecting 10 results back from any particular slave. Otherwise 1 task will be sent and 1 result sent back from the slave. Grouping the tasks or results together makes the network communication more efficient. The master is told to start up on the `localhost` interface. There is no method to start the master bound to all interfaces.

In the log output below, the member id token of the master and slave is **"default-member-id"**. In normal use, this should probably be changed to be unique to the specific master/slave computation. Please see the section on command line arguments on page 33 for how to do this.

```
> sbcl
This is SBCL 1.0.39.16, an implementation of ANSI Common Lisp.
More information about SBCL is available at <http://www.sbcl.org/>.

SBCL is free software, provided as is, with absolutely no warranty.
It is mostly in the public domain; some portions are provided under
BSD-style licenses.  See the CREDITS and COPYING files in the
distribution for more information.
* (require :cl-mw.examples.hello-world)
[ Lots of output on compiling and loading libraries ]

* (use-package :cl-mw.examples.hello-world)

T

* (mw-master)
07/20/2010 23:15:42 [A] INIT MASTER "default-member-id"
07/20/2010 23:15:42 [A] MASTER READY 127.0.0.1:52942
```

At this point, the master process has already created some hello world tasks and is waiting for some slaves to connect. The output lines with `[A]` in them are emitted as an audit trail by the CL-MW library. The **"default-member-id"** is the membership token of the master which the slave must match. Let's start up a slave with our helper slave initialization function and pass in the port number of the master process—because for this example the helper slave function assumes localhost.

```
> sbcl
This is SBCL 1.0.39.16, an implementation of ANSI Common Lisp.
More information about SBCL is available at <http://www.sbcl.org/>.

SBCL is free software, provided as is, with absolutely no warranty.
It is mostly in the public domain; some portions are provided under
BSD-style licenses.  See the CREDITS and COPYING files in the
distribution for more information.
* (require :cl-mw.examples.hello-world)
[ Lots of output on compiling and loading libraries ]

* (use-package :cl-mw.examples.hello-world)

T

* (mw-slave 52942)
07/20/2010 23:23:46 [A] INIT SLAVE "default-member-id"
07/20/2010 23:23:47 [A] MASTER <- CONNECTED TO 127.0.0.1:52942 \
                        FROM 127.0.0.1:47768
07/20/2010 23:23:47 [A] MASTER -> ID SLAVE-0
07/20/2010 23:23:47 [A] MASTER -> 10 tasks (10 grouping)
07/20/2010 23:23:47 [A] MASTER <- 10 results
07/20/2010 23:23:47 [A] MASTER -> SHUTDOWN
Slave algo cleanup form.
07/20/2010 23:23:47 [A] FINI SHUTDOWN "default-member-id"
0
*
```

The last number is the return code of the slave function.

Meanwhile, let's see what the master emitted:

```
07/20/2010 23:23:47 [A] NEW-CLIENT -> 127.0.0.1:47768
07/20/2010 23:23:47 [A] SLAVE-0 127.0.0.1:47768 -> \
                        ["default-member-id"] \
                        :connecting [:unordered]
07/20/2010 23:23:47 [A] SLAVE-0 -> :idle
07/20/2010 23:23:47 [A] SLAVE-0 <- 10 tasks
07/20/2010 23:23:47 [A] SLAVE-0 -> :busy
07/20/2010 23:23:47 [A] SLAVE-0 -> 10 results
07/20/2010 23:23:47 [A] SLAVE-0 -> :idle
Got result from slave: "Hello World: Task 0"
Got result from slave: "Hello World: Task 1"
Got result from slave: "Hello World: Task 2"
Got result from slave: "Hello World: Task 3"
Got result from slave: "Hello World: Task 4"
Got result from slave: "Hello World: Task 5"
Got result from slave: "Hello World: Task 6"
Got result from slave: "Hello World: Task 7"
Got result from slave: "Hello World: Task 8"
Got result from slave: "Hello World: Task 9"
Master algo cleanup form.
07/20/2010 23:23:47 [A] SLAVE-0 <- TRY-SHUTDOWN
07/20/2010 23:23:47 [A] SLAVE-0 -> :shutting-down
07/20/2010 23:23:47 [A] SLAVE-0 -> :disconnected
07/20/2010 23:23:47 [A] EOF -> 127.0.0.1:47768
07/20/2010 23:23:47 [A] FINI SHUTDOWN "default-member-id"
0
*
```

`Note`: The audit lines have been reformatted slightly to fit. They do not have the traditional shell line continuation characters in them.

We see that the master had packaged all ten tasks into one packet and sent it to the slave. After getting the results–also in one packet, back, it printed them out. At this point the results have equaled the tasks in the *master algorithm* and it returns. CL-MW enters the shutdown phase where it actively tried to shut off all known slaves and then exit with the return code the *master algorithm* generated. If a severe problem arose during shutdown, then the return code will be set to 255.

3.3 Producing an Executable

The :`CL-MW` package exports the function `mw-dump-exec` which saves the Lisp image as an executable to the current working directory. We recommend that this function be re-exported from the application package built on top of the `:CL-MW` package as shown previously in the ASDF file for this example. Exporting this function makes it trivial to produce an executable—one just **require** s the package, then **use-package** s it, and then calls `mw-dump-exec` to produce the binary.

`mw-dump-exec` simplifies collecting required libraries that may not be present on the slave system. `mw-dump-exec` will copy any currently loaded libraries with an absolute path into the current working directory. For libraries without any path, it will approximate the search algorithm used by `dlopen()` to find an absolute path for the library and then copy it to the current working directory. `mw-dump-exec`, with the `:ignore-libs` keyword argument, can be told to ignore specific libraries loaded by the lisp image. One would supply a list of strings representing unqualified library names to be ignored. Libraries can also be remapped, with the `:remap-libs` keyword argument, from their unqualified name to a specific path. An association list should be supplied with `:remap-libs` which maps unqualified library names to absolute paths. Ignoring a library overrides a remap of a library, and a remap of a library overrides the auto detection of the library's absolute path. `mu-dump-exec` will update the Lisp image to look for the dumped libraries in the path `./` when the saved executable is started.

How the lisp image is started before the executable is produced is important. We start SBCL up with the `--disable-debugger` option which tells SBCL to dump a stack trace and exit when something has gone wrong in the executable—such as the signaling of an unhandled condition. Otherwise, SBCL will drop into an interactive debugging session and wait for input to arrive. Disabling the debugger prevents the executable from having a problem and then consuming valuable compute time on a resource waiting for input which will never come.

Dropping into the debugger is one of a few things in the execution environment that can be altered with various command line options to SBCL. Another common adjustment to set is how big the heap is in the Lisp image runtime. The default runtime heap size is operating system specific. On the Linux machine upon which I developed CL-MW, it was 512MB and so for each invocation of the master and slave executable, about 512MB of memory will be requested from the operating system—even if it isn't all used by the application. Depending upon your *master algorithm* and *task algorithms*, you may need to tune the runtime heap size to fit the computation requirements. Please see the SBCL manual for more tunable options as needed by your computation.

```
> sbcl --disable-debugger
This is SBCL 1.0.39.16, an implementation of ANSI Common Lisp.
More information about SBCL is available at <http://www.sbcl.org/>.

SBCL is free software, provided as is, with absolutely no warranty.
It is mostly in the public domain; some portions are provided under
BSD-style licenses.  See the CREDITS and COPYING files in the
distribution for more information.
* (require :cl-mw.examples.hello-world)
[ Lots of output on compiling and loading libraries ]

* (use-package :cl-mw.examples.hello-world)

T

* (mw-dump-exec)

######################################
# Processing loaded shared libraries #
######################################
Shared-library: /home/psilord/content/code/lisp/clbuild/source\
                /iolib/src/syscalls/libiolib-syscalls.so...\
                dumping...fixating.
Shared-library: librt.so...looking up...found \
                /usr/lib/librt.so...dumping...fixating.

########################################################
#  Please package these libraries with your executable #
########################################################
./librt.so
./libiolib-syscalls.so

####################################
# Writing Master/Slave executable #
####################################
[undoing binding stack and other enclosing state... done]
[saving current Lisp image into ./a.out:
writing 3512 bytes from the read-only space at 0x01000000
writing 2256 bytes from the static space at 0x01100000
writing 38322176 bytes from the dynamic space at 0x09000000
done]
```

`mw-dump-exec` iterated over all of the shared libraries being used by the Lisp image. `mw-dump-exec` determined that the shared library `libio-syscalls.so` used by IOLib (a package needed by CL-MW) must be included and the actual library file is copied into the current working directory. Then the Lisp image is adjusted to look in the path `./` for `libio-syscalls.so`. `mw-dump-exec` noticed that `librt.so` didn't have an absolute path but had a successful search for an absolute path to the library. This library is also copied to the current working directory and the lisp image adjusted to find it. If `mw-dump-exec` isn't told otherwise, the name of the binary it dumps is `a.out`. You can supply a different executable name to `mw-dump-exec`, see page 44 for details.

If this executable is supplied with the same arguments to `mw-initialize` as defined in the helper function `mw-master` and another invocation started with the same arguments to `mw-initialize` as defined in the helper function `mw-slave` (adjusting for the master's host and port!), then you will see similar output as the slave executes the master's tasks.

Here we see the executable and the shared libraries with which it should be bundled when moved to another machine for execution:

```
> ls a.out *.so
-rwxr-xr-x 1 psilord psilord 38948892 Jul 20 23:44 a.out*
-rw------- 1 psilord psilord     7235 Jul 20 23:44 libiolib-syscalls.so
-rw------- 1 psilord psilord    30684 Jul 20 23:44 librt.so
```

`Important`: Any dumped shared libraries **must** exist in the current working directory when the main binary is invoked for them to be found by the restarting binary. Relative paths and the environment variable `LD_LIBRARY_PATH` do not work properly.

3.4 The Audit File

The master and slave process both can write their audit trail to a specified file. This is done with the `--mw-audit-file` *file-name* command line option. When this option is used, every written line above with an `[A]` in it will be written to the specified audit file. Any other output that the *master algorithm* or *slave algorithm* creates will go to `*standard-output*` or to wherever that is bound.

<u>`Limitation`</u>: The audit files do not rotate and can grow unboundedly. The audit file will be appended to if it exists upon start of the master or slave process.

CL-MW does minimal statistics bookkeeping. The audit files can be used to answer questions about the application's run. For example, how many slaves are connected, what is the slave churn rate, on what subnet are the slaves, or how many tasks were processed for a given time interval.

<u>`Note`</u>: The format of the audit file may change in a future revision of CL-MW.

Chapter 4

Interfacing with Existing Batch Systems

CL-MW is designed to work with existing batch systems. CL-MW has no provision for starting up slaves on remote machines, moving files between machines, detecting and killing run away slaves, managing user identity, storing or transmitting credentials, or enforcing authorization policies for resource use. These, and many other features, are general features which batch systems usually provide. In general, batch systems provide very robust mechanisms for each of these features and are well suited to handle many edge cases which show up in practice. Usually a CL-MW application will be sharing resources with other applications across a common cluster of machines.

What CL-MW does provide is a means for communicating to a batch system called the *resource file*. The master process, when configured to do so, periodically writes information into the *resource file*, including liveness of the master process, if the computation is still in progress, how many slaves the master desires, and how to start up those slaves. The master process adjusts the resource requirement information in the file based upon the outstanding workload and the slaves successfully started by the batch system.

The *resource file* is written when the command line option
`--mw-resource-file` *file-name* is used with the master process. By default the master will rewrite it every 300 seconds (5 minutes) with new information. This can be controlled by the command line option
`--mw-resource-file-update-interval` *integral-seconds*.

The *resource file* contains information in Lisp form and is intended to be read by another process specific to the batch scheduler whose responsibility it is to acquire resources from said batch scheduler. The batch scheduler is directly notified in the *resource file* if the *master algorithm*'s computation is in progress or finished.

The slave process can also read the *resource file* in order to determine the master host, port, and member-id to which it should connect and if the computation is still in progress or finished. Depending upon the batch scheduler, having the slave read the *resource file* directly can be a very helpful because the dynamic connection information needed by the slave is available in the file. There is no requirement to adjust meta-information for the slave (e.g., its command line arguments which may specify a new master ip:port combination) through the batch system itself.

The internals of the *resource file* are described on page 45.

4.1 Condor

Condor is a versatile, robust, and free batch scheduling system available from `http://www.cs.wisc.edu/condor`. It can maintain high job throughput for tens of thousands of jobs on tens of thousands of resources. Condor's built in mechanisms for file transfer, job execution policies, and robustness mechanisms make it a good distributed computing platform on which to run CL-MW applications. We describe a simple application of Condor which provides a reliable execution platform for a CL-MW application.

4.1.1 Interfacing CL-MW with Condor

Condor can be told to transfer the job's input files to the remote execute node just before the job is about to start. Condor will do this each time a job tries to run when it has been sitting idle in the queue. We use this fact to copy over the *resource file* written by the master over to the slave so the slave knows where to connect.

We use a Condor feature that can control when a terminated job is to removed from the queue. We forbid the slave job to be removed from the queue unless the slave gets told to shutdown by the master or otherwise exits with the return code of zero. If the master dies without producing a completed answer, the slaves, having noticed that the master closed the connection without having been told to explicitly shut down, will exit with a non-zero status. The slave jobs will remain in the job queue due to the

Condor enforced job policy. If the slave runs again and tries to connect using a stale resource file (due to the master not running for an extended period of time), the slave will again exit with a non-zero status value, again remaining in the queue.

We also submit the master process itself as a job into the Condor system. The master will always run on the submit node as opposed to being shipped to an execute node. This allows the master access to the needed input files or other resources usually only found on the submission machine and owned by the submitter of the job. We do not perform the same type of job policy for the master as we did for the slaves. If the master exits with any return code, it will be removed from the job queue. However, in the event of machine reboot (or many other types of failures), Condor will, when it starts running again, know that the master and slave processes were present in the job queue. It will restart the master and again find more resources for the slave jobs. When the master restarts it will write a new *resource file*. When a slave runs again, the new *resource file* gets moved to it and it will connect with the currently running master. Finally, if the master has finished and written the final update into the *resource file* stating the computation is finished, then any slaves that start up with that particular *resource file* will immediately exit with a status zero. This allows the slaves to be removed from the queue. If a slave is told to shutdown properly by the master, and does so, then it will exit with an exit code of 0 and also be removed from the queue.

The combination of the slave job policy and the restart robustness of the master makes CL-MW jobs reliable. In the event of a restart of the master process, the entire computation will restart. However, the entire computation will reliably restart and run until it finally completes.

The particular interface described in this section is: simple, doesn't make full use of the information the *resource file*, and wastes compute time on the execution nodes during the time the master process is down. It could be extended with an actual process outside of CL-MW and managed by Condor which watches the *resource file* and actively submits and removes slaves from Condor based on the master's current resource needs.

A Simple Interface to Condor

This example describes the Condor interface for the `ping` example supplied with CL-MW.

There are two Condor submit files, one for a single master process, and one for a static cluster of slave processes. The master will write a *resource file* and the slaves, using file transfer, will read it and know where to connect. No dynamic adjustment of resource acquisition is done in this example.

The Master Process

The following submit file details how the master process is to be run. This file is given to Condor's `condor_submit` which submits the job into Condor's job queue on the local machine.

```
# Begin master.sub
universe = scheduler
executable = ./ping
arguments = --mw-master \
--mw-slave-task-group 100 \
--mw-slave-result-group 100 \
--mw-resource-file resource-file.rsc \
--mw-slave-executable ping \
--mw-audit-file master.$(CLUSTER).$(PROCESS).audit \
--mw-member-id $(CLUSTER).$(PROCESS)

output = master.$(CLUSTER).$(PROCESS).out
error = master.$(CLUSTER).$(PROCESS).err
log = master.$(CLUSTER).$(PROCESS).log

notification = NEVER

queue 1
# End master.sub
```

Each line will be described as to its effect on the job submission.

```
universe = scheduler
```

The job is marked to be a scheduler universe job. It will start immediately and only on the machine where this job is submitted.

```
executable = ./ping
```

The executable Condor will use when executing this job. Condor and the `ping` executable will assume the needed libraries, if any, associated with the `ping` program are present in the same directory as the executable.

```
arguments = --mw-master \
   --mw-slave-task-group 100 \
   --mw-slave-result-group 100 \
   --mw-resource-file resource-file.rsc \
   --mw-slave-executable ping \
   --mw-audit-file master.$(CLUSTER).$(PROCESS).audit \
   --mw-member-id $(CLUSTER).$(PROCESS)
```

This is the complete list of command line arguments supplied to the master process when it is executed by Condor.

The meaning of the `--mw-*` arguments in numerical order are:

1. Must be first and specifies that this invocation of the `ping` binary is the master process.
2. Number of tasks that the master will pack into one network packet to a slave.
3. Number of results that the slave will pack into one network packet to the master.
4. Specify the resource file. The *resource file* path must be unique to each CL-MW computation submitted to the same scheduler daemon in Condor. The path must match between the master's submit file and the slave's submit file.
5. We explicitly specify the slave executable name. Otherwise, the master would try to determine the name of itself when it is running in order to find its own executable to use as the slave executable in the *resource file*. The explicit specification of the slave executable is necessary because Condor specifies a different name for the executable when executing it.
6. An audit file is specified based upon the cluster and process id of the job. It will be filled with information about who and how the slaves connect and what work is given to them. We use Condor's $(CLUSTER) and $(PROCESS) macros,

which are unique per job submitted, to assign a unique identifier to this file.

7. Using Condor's $(CLUSTER) and $(PROCESS) mechanism, we assign a unique identifier to the master. This identifier will be written into the *resource file* so that the slaves can authenticate themselves to the master.

```
output = master.$(CLUSTER).$(PROCESS).out
```

Any standard output written by the *master algorithm* will be written here.

```
output = master.$(CLUSTER).$(PROCESS).out
```

Any standard error output written by the *master algorithm* will be written here.

```
log = master.$(CLUSTER).$(PROCESS).log
```

This file is written by Condor and is a sequential record of a job's lifetime in Condor. A sample of the events which can happen to a job are: submission, execution, termination, held, released, etc. This file is a very useful debugging and tracking tool to find out the state in which a job may be.

```
notification = NEVER
```

No matter how this job completes, do not send an email to the account which submitted this job. Valid options are **ALWAYS**, **COMPLETE** (the default if `notification` is not specified), **ERROR**, and **NEVER**.

```
queue 1
```

This will submit one cluster of jobs into Condor with only one job in the cluster.

When this job is submitted, it should start immediately and create the *resource file* `resource.rsc`. After this file is in existence, the slaves can be submitted.

The Slave Processes

The following submit file submits a static cluster of vanilla jobs which are the slaves.

```
# Begin slaves.sub
universe = vanilla
executable = ./ping
arguments = --mw-slave \
--mw-resource-file resource-file.rsc

output = slaves.$(PROCESS).out
error = slaves.$(PROCESS).err
# All slaves will share a log file.
log = slaves.log

should_transfer_files = YES
when_to_transfer_output = ON_EXIT
transfer_input_files = libiolib-syscalls.so,resource-file.rsc

notification = NEVER

on_exit_remove = (ExitBySignal == False) && (ExitCode == 0)

queue 1000
# End slaves.sub
```

We describe the interesting lines in this submit file.

```
universe = vanilla
```

This fixates a set of features for this job in the Condor system which state that the job can run on any suitable execute machine in the pool.

```
arguments = --mw-slave \
   --mw-resource-file resource-file.rsc
```

The meaning of the `--mw-*` arguments in numerical order are:

1. Must be first and specifies that this invocation of the `ping` binary is a slave process.
2. Specifies a resource file the slave will use to contact the master process and identify itself.

`should_transfer_files = YES`

This states that Condor is responsible for moving any files the job needs or produces during its execution. If not specified it means there is a common file system between the submit machine and the execute machine that the job can access.

`when_to_transfer_output = ON_EXIT`

This specifies that Condor only cares about the output a job produces when a job completes.

`transfer_input_files = libiolib-syscalls.so,resource-file.rsc`

Any input files needed by the job are specified here. We specify the shared libraries the slave executable needs in addition to the resource file. Condor will transfer the most recent versions of these input files each time the job starts.

`on_exit_remove = (ExitBySignal == False) && (ExitCode == 0)`

This implements the job policy previously mentioned. Only allow the job be removed from the queue when it hasn't exited by a signal and the exit code of the job is zero. If the job exits for any other reason, it will remain in the job queue and be eventually restarted.

`queue 1000`

Submit one cluster with 1000 jobs in it into Condor.

In summary, one thousand jobs will be submitted into this cluster. The `:slaves-needed` setting in the *resource file* is ignored, as there is no overseer watching the *resource file* and managing resource acquisition on behalf of the *master algorithm*. The *resource file* is transmitted as an input file and under Condor this means to transmit it anew every time the job is rerun. Condor also send over any shared libraries or other files the executable needs. These slaves will stay in the queue until they either connect to a master and the master tells it to shut down, or because the master wrote `:computation-status :finished` into the *resource file* and the slave reads the file. In addition, we specify the `on_exit_remove` policy for the job; the slave will only be removed from the queue if the slave had not exited with a signal and the exit code was zero as requested by the master. The output of the slaves will be the standard out of the *slave algorithm*. Since no audit file is specified, the auditing information will go to the standard out of the process.

4.1.2 Environmental Requirements

Running binaries across a host of machines which differ in OS revisions, physical capabilities, and network will bring to the forefront scalability and binary compatibility problems. Here we describe a few common problems and their solutions.

Memory Requirements

Depending upon the memory capabilities of the resource slots in the pool (suppose they only have 384MB each), a slave may run once, consume 512MB which Condor records as the memory usage for the job, be preempted for some reason, and then never run again because no slot in the pool will accept a 512MB job anymore.

This is fixed by either adding `Requirements = Memory > 0` (or whatever fits your needs) to the slave's submit file or adjusting the fixed runtime heap size with command line arguments to SBCL when you create the executable. The latter choice is safer since it models the true resource requirements your application needs and does a better job of preventing thrashing. The former is more useful for testing purposes.

Network or Disk Bandwidth

Another environmental concern is the network or disk bandwidth of the submit machine as potentially thousands of slaves simultaneously have their executables, shared libraries, and other files transferred from the submit machine and onto the execute slots. In practice this often isn't a problem, but it is good to know what to do if it becomes one.

The *condor_schedd* config file entries `JOB_START_DELAY` and `JOB_START_COUNT` can be used to limit the job start rate to restrict network and disk bandwidth when bursts of jobs begin running.

Dynamic Linking

Since a CL-MW application is a dynamically linked binary, it will need to find and load its required libraries at run time. When the binary is moved from the submit host to the execute host, the execute host may not have a required dynamic library available, or more rarely, a required kernel syscall interface the job needs. In this event, the job (in our case, the slave) will (often) die with a SIGKILL and go back to idle (due to our `on_exit_remove` policy). It could be possible for a slave to continuously match to a machine upon which it cannot run. In this situation the slave will accumulate runtime but make no forward progress. The preferred solution is to package your libraries with your job. In the rare cases where this will not be sufficient, you may have to restrict the set of machine upon which your job runs. Please read section 2.5 in the Condor manual for how to specify this kind of a requirement for your job.

Chapter 5

Technical Specification

5.1 Command Line Arguments

These are the command line arguments the CL-MW library accepts. These command line arguments are stripped from the argv before the argv is handed to the *master algorithm*.

`--mw-help` Emit the usage and exit.

`--mw-version-string` Emit the version string and exit.

`--mw-master` Run the executable in Master Mode. Required if `--mw-slave` is not set and must be first on the command line.

`--mw-slave` Run the executable in Slave Mode. Required if `--mw-master` is not set and must be first on the command line.

`--mw-master-host` ***ip-address-or-hostname*** When in Master Mode, it is the interface (either the hostname or the ip address) to which the master should bind and is emitted to the resource file if any such file is written. When in Slave Mode, it is the hostname, or ip address, to which the slave process should connect and get work.

`--mw-master-port` ***port*** To which port should the slave connect for work.

`--mw-max-write-buffer-size` ***size-in-bytes*** How big the network writing buffer should be before rejecting the write.

`--mw-max-read-buffer-size` ***size-in-bytes*** How big the network reading buffer should be before rejecting the read.

`--mw-client-timeout` ***seconds*** How many seconds should the master wait for a client to respond when the master is expecting a response.

`--mw-audit-file` ***filename*** A file in which the audit trail of the process is stored.

`--mw-resource-file` ***filename*** Describes the resources needed by the master for a higher level batch system to honor.
When in master mode this file contains information concerning:

- The time stamp of when the file was written.
- The member id of the master group.
- The update interval of when this file will be written again.
- How many slave processes are needed by the master.
- The full path to the slave executable.
- The complete arguments to the slave in order for it to connect to the currently running master process which produced this file.

When in slave mode:

> Determine the master-host, master-port, and member-id to which the slave should connect by reading it from the resource file. The ordering of this command line option in relation to `--mw-master-host`, `--mw-master-port`, and `--mw-member-id` is important. If `--mw-master-host`, `--mw-master-port`, and/or `--mw-member-id` are specified before this argument then the resource file will overwrite the command line specification, and vice versa. If the resource file does not exist, then this knowledge is ignored (but warned about) if `--mw-master-host` and `--mw-master-port` are present.

`--mw-resource-file-update-interval` ***seconds*** How many seconds between updating the resource file with current information.

`--mw-slave-task-group` ***positive-integer*** How many tasks are grouped into a network packet being sent to a slave process. If the packet is larger than the maximum size of the read buffer of the slave, the slave will abort the read. Defaults to 1.

`--mw-slave-result-group` ***positive-integer*** How many completed results should be grouped into a network packet being sent from the slave to the master. If the packet is larger than the maximum size of the read buffer for the master, then the master will abort the connection to the slave. Defaults to 1.

`--mw-member-id` ***string*** This is a token which must match between the slave and the master. It is used to insecurely identify a working group of masters and slave. In a harsh environment with many masters and slaves going up and down, this acts as a simple sanity check that the correct slaves are connected to the correct master process. Default is the string "default-member-id".

`--mw-slave-executable` ***path-to-executable*** This specifies the absolute path to a slave executable. It is used when writing the resource file only.

5.2 The API

The CL-MW library is in the `:CL-MW` package and it is used by the application package built on top of CL-MW. The exported symbols in the `:CL-MW` package are:

The Task Algorithm

`(define-mw-algorithm` *`name`* `(`*`parameters*`*`) &body` *`body`*`)` Macro

Defines a *task algorithm* with name *name*. The arguments passed to this call are exactly those which were passed into the `mw-funcall-`***`name`*** form for the *task algorithm*.

`Limitation`: The *parameters* list is restricted to being *required* parameters only.

`Limitation`: A *task algorithm* may not return multiple values or a function or closure. The latter restriction is due to the inability to serialize a closure from the slave to the master.

Task Computation Function Generated by `define-mw-algorithm`

`(`*`name parameters*`*`)` Function

This is the function which actually performs the work of the *task algorithm*. It accepts the parameters specified and returns the last expression in the body supplied to the *task algorithm* macro.

Task Submission Macro Generated by `define-mw-algorithm`

```
(mw-funcall-name (parameters) &key
   sid tag do-it-anyway (retry t))                                    Macro
```

This is a destructuring macro which will insert a single new task of the *task algorithm* named by *name* into CL-MW. The *parameters* are in the same order as the parameter list for the defined *task algorithm* and are evaluated before being packed into the task structure. The other parameters describe a behavior which together constitute the *task policy* for a submitted task.

sid ***SLAVE-ID*** Send the task to a specific slave denoted by *SLAVE-ID*. If `NIL`, this task is considered `:unordered`, otherwise it is a `:ordered` task.

tag ***FORM*** A form which will appear unchanged in the result structure associated with the computed task. The default is `NIL`.

do-it-anyway [**T** ***or*** **NIL**] If the task was a `:ordered` task and the slave disconnected, then should this task be moved into the `:unordered` group (yes if T), or become unrunnable (yes if `NIL`)? By default `:ordered` tasks become unrunnable if the associated slave is disconnected.

retry [**T** ***or*** **NIL**] If an unordered task was assigned to a slave and the slave went away, then this controls if we should retry on a different slave or if the task becomes unrunnable. If this test passes then :do-it-anyway is consulted in the case of `:ordered` tasks.

Specific Target Number API Generated by `define-mw-algorithm` `(mw-set-target-number-for-`*`name value`*`)` Function

Sets the target number for the *task algorithm* specific to *name* to *value*, which is clamped to zero or greater. This represents the maximum number of pending tasks for this *task algorithm* that the *master algorithm* would like to keep in memory at once. This target number is advisory and the *master algorithm* can insert more tasks than indicated by the target number. The default target number for any specific *task algorithm* is 0.

`(mw-get-target-number-for-`*`name`*`)` Function

Returns the target number for the number of desired tasks to keep in memory for the *task algorithm* specific to *name*.

`(mw-pending-tasks-for-`*`name`*`)` Function

Return how many tasks are in memory (and not running on any slaves) specific to the *task algorithm name*.

`(mw-upto-target-number-`*`name`*`)` Function

Returns the number of tasks the *master algorithm* would have to create in order to reach the desired target number for *task algorithm name*.

The General Target Number API

`(mw-set-target-number `*`value`*`)` Function

Sets the general target number for all tasks regardless of *task algorithm*. This is only advisory and more tasks could be created into CL-MW by the *master algorithm*.

`(mw-get-target-number)` Function

Return the current value of the general task target number. The default value for the general target number is 0.

`(mw-pending-tasks)` Function

Return how many tasks of any kind are waiting to be scheduled to slaves.

`(mw-upto-target-number)` **Function**

Return how many tasks of any kind should be created by the master in order to reach the general target number for all tasks.

<u>The Master Algorithm</u>

`(define-mw-master (`*`argv`*`) &body `*`body`*`)` **Macro**

Defines the *master algorithm* for the application of which there may only be one. When the *master algorithm* has finished computation, it must return an integer from 0 to 255 which will become the return code of the process. If this doesn't happen, the return integer will be 255.

<u>`Note`</u>: If no master algorithm is specified in a CL-MW application. An audit line will be emitted stating this fact and the master computation will shut down immediately. A return code of 255 will happen in this case.

Parameter *argv* will be the command line arguments passed to the executable or to `mw-initialize` with the CL-MW specific arguments stripped out.

`(mw-master-loop &key (`*`timeout`*` .05))` **Function**

Enter the CL-MW system loop processing I/O and other library tasks until one or more of these events happen:

- Some tasks become unrunnable.
- There are pending results from slaves.
- Sequential slaves have connected to the computation.
- Sequential slaves have disconnected from the computation.

When one or more of these events happen the function will return the 4 values:

1. Number of unrunnable tasks
2. Number of ready results
3. Number of newly connected and unprocessed ordered slaves
4. Number of newly disconnected and unprocessed ordered slaves

Parameter *timeout* is a time unit in real seconds which should be waited in the Network IO multiplexing library before timing out due to inactivity. In the case of this function, it means we perform bookkeeping work inside of the CL-MW library and enter back into the loop if no meaningful events occurred. Setting this value too low will result in excessive CPU usage by the master process.

`(mw-master-loop-iterate &key (`*`timeout`*` .05))` Function

Enter the CL-MW system loop processing a *single* pass of network I/O and other library tasks. After this call one or more of the same events as described in `mw-master-loop` *may* have happened.

Parameter *timeout* is a time unit in real seconds which should be waited in the Network IO multiplexing library before timing out due to inactivity. In the case of this function, it means we return the 4 values as described in `mw-master-loop`. Setting this value too low could result in excessive CPU utilization.

`(mw-get-unrunnable-tasks)` Function

Return all currently unrunnable task structures in a list or `NIL` if none.

`(mw-get-results)` Function

Return all currently finished result structures in a list or `NIL` if none.

`(mw-get-connected-ordered-slaves)` Function

If there are any connected ordered slaves ready for use, this will retrieve the list of slave ids or `NIL` if none. In practice each slave id is a string, but generally they are an opaque data structure used to uniquely identify a slave. You should use `equal` to check quality between slave ids.

`(mw-get-disconnected-ordered-slaves)` Function

If any ordered slaves have become disconnected, return a list of their slave ids. You may use `equal` to compare against other slave ids.

`(mw-allocate-slaves &key (`*`amount`*` 1000) (`*`kind :unordered`*`))` Function

There are three kinds of groups for which slaves can be allocated: `:ordered`, `:intermingle`, `:unordered`. When a slave initially connects for work, it is placed into one of the three groups. The order of group fulfillment is `:ordered`, `:intermingle`, `:unordered`. If both `:ordered` and `:intermingle` are full, then any connecting slaves go over to the `:unordered` group. The total number of desired slaves for all groups is written into the resource file as the number of needed slaves. This function can cause slaves in the `:unordered` group to move to the groups desired.

It is valid for the `:unordered` group to contain more than the allocation for it. The default allocation for all groups is 0.

`(mw-deallocate-slaves &key (`*`amount`*` 0) (`*`kind`*` :unordered))` Function

This does not stop any slaves from processing any tasks, but it does lower the number of slaves desired, clamped to zero, of any of the of group `:unordered`, `:intermingle`, or `:ordered` as specified. This relates to what is written in the resource file by the master process.

`(mw-free-slave `*`slave-id`*`)` Function

Move the slave specified by *slave-id* into the `:unordered` group after it completes whatever tasks it may be running and adjust the desired slave amounts for the group the slave was in. This does not evict or otherwise stop currently allocated tasks from running on that slave. The slave's group is only changed once all of the tasks it is currently running are computed.

`(mw-num-runnable-tasks)` Function

Returns the number of runnable tasks which includes tasks that were sent out and currently executing on slaves.

`(mw-num-unrunnable-tasks)` Function

Returns the number of unrunnable tasks in waiting to be consumed out of CL-MW with `mw-get-unrunnable-tasks`.

The Slave Algorithm

`(define-mw-slave (argv) &body body)` Macro

Defines the *slave algorithm* for the application of which there may only be one. When the *slave algorithm* has finished computation, it must return an integer from 0 to 255 which will become the return code of the process. If this doesn't happen, the return integer will be 255.

Parameter *argv* will be the command line arguments passed to the executable or to `mw-initialize` with the CL-MW specific arguments stripped out.

Note: If no *slave algorithm* is specified in a CL-MW application, then the default *slave algorithm* defined in listing 5.1 is used. An audit line entry will occur stating that the CL-MW default *slave algorithm* is being used.

Listing 5.1: Default Slave Algorithm

```
(define-mw-slave (argv)
  (mw-slave-loop-simple))
```

`(mw-slave-loop &key (timeout .05))` Function

Process all pending tasks and return control to the *slave algorithm*.

This function will return 6 values in this order:

master-disconnect Did the master close the connection to the client (or under some conditions CL-MW wanted to immediately exit due to some problem in the environment). T if the master cut the connection or the library wanted to exit, `NIL` otherwise.

explicit-shutdown Did the master send a shutdown command to the slave according to the master/slave protocol? T if it did and `NIL` if it didn't.

total-results-completed The number of total results which have been completely processed by the slave.

num-tasks A number which is how many tasks are yet to be processed.

num-results The number of results that are currently waiting to be sent back. This is affected by the master process with the command line parameter `--mw-slave-result-group`.

result-grouping The number of results which must be grouped together before being sent back (or if there are no more tasks to compute whatever results are pending to go back get sent back).

Parameter *timeout* is a time unit in real seconds which should be waited in the Network IO multiplexing library before timing out due to inactivity. In the case of this function, it means we perform bookkeeping work inside of the CL-MW library and then return into the *slave algorithm*. Setting this value too low will result in excessive CPU usage by the master process.

`(mw-slave-loop-iterate &key (timeout .0001))` Function

Process a *single* pending task, inspect the network buffers for more work to do, and return control to the *slave algorithm*. This will generally be extremely slow and hence has a short timeout. It returns the same values as `mw-slave-loop` and there may or may not have been any new tasks sent by the master in that time.

Parameter *timeout* is a time unit in real seconds which should be waited in the Network IO multiplexing library before timing out due to inactivity.

`(mw-slave-loop-simple &key (timeout .05))` Function

Process all pending tasks form the master and wait for more. Only return when the master says to shutdown or there was a bad error and return 0 or 255 respectively.

Parameter *timeout* is a time unit in real seconds which should be waited in the Network IO multiplexing library before timing out. In the case of this function, it means we perform bookkeeping work inside of the CL-MW library and begin waiting again for more tasks from the master, or a shutdown command. Setting this value too low will result in excessive CPU usage by the slave process.

The Task Structure

`(mw-task-sid` ***`task-structure`*** `)` `Function`

Returns the slave-id for which the *task-structure* was destined. If the task is `:unordered`, then `NIL` is returned.

`(mw-task-tag` ***`task-structure`*** `)` `Function`

Return the associated tag object for this *task-structure*, or `NIL` if not set.

`(mw-task-packet` ***`task-structure`*** `)` `Function`

Retrieve, as a list, the arguments specific to the algorithm for which this *task-structure* was created.

The Result Structure

`(mw-result-algorithm` ***`result-structure`*** `)` `Function`

Return an uppercase string which is the name of the *task algorithm* that produced this *result-structure.*

`(mw-result-sid` ***`result-structure`*** `)` `Function`

Return the slave id of the slave which produced this *result-structure.*

`(mw-result-tag` ***`result-structure`*** `)` `Function`

Retrieve the unmodified tag associated with the original *task-structure* for this *result-structure.*

`(mw-result-compute-time` ***`result-structure`*** `)` `Function`

Return the length of time in seconds which represents how long it took to compute this *result-structure.*

`(mw-result-packet` ***`result-structure`*** `)` `Function`

Retrieve the actual returned form of the *task algorithm* which produced this *result-structure.*

Miscellaneous API

`(mw-initialize (`*`argv`*
  `&key (`*`system-argv sb-ext:*posix-argv*`*`)))` Function

The entry point into CL-MW. The parameter *argv* is a list of strings which represent the argument list to the library. Anything not a CL-MW specific argument will be passed to the *master algorithm* or the *slave algorithm* in the same order as it was on the command line.

`(mw-version-string)` Function

Return a string which represents the version number for this library.

Note: The format and meaning of this string may change in the future.

`(mw-zero-clamp `*`value`*`)` Function

If the *value* is less than zero, then return 0, otherwise return the *value*.

`(mw-dump-exec &key (`*`exec-name`*` "a.out")`
  `ignore-libs remap-libs``)` Function

Produce an executable named *exec-name*, which is `a.out` by default, and copy any shared libraries needed by the application into the current working directory.

Any shared libraries loaded in the lisp image which are already an absolute path will be copied verbatim to the current working directory. Any unqualified libraries will be transformed by an algorithm approximating the search algorithm of `dlopen()` into absolute paths and then copied to the current working directory. The dumped shared libraries must be shipped with the executable to the target machine.

The parameter *ignore-libs* is a list of strings where each string is an unqualified library name. These libraries will be ignored by `mw-dump-exec`. If this parameter is `NIL`, the default, then no libraries are ignored.

The parameter *remap-libs* is an association list of strings where the first string is an unqualified library name and the second an absolute path to a library that will be copied to the current working directory in place of what is found in the lisp image. If this parameter is `NIL`, the default, then no libraries are remapped.

This interface may change in the future.

Limitation: The dumped libraries must exist in the current working directory when the executable is run.

`(while` *`test-expr`* `&body` *`body`*`)` Macro

A ubiquitous macro which implements the usual "while" loop control flow.

5.3 Resource File

Each form in the *resource file* is a two item list where the first item is the attribute name as a keyword, and the second an arbitrary Lisp form whose schema depends upon the specific attribute. They take the form of:

```
(keyword form)
```

The current attributes for the *resource file* in this version of CL-MW are:

:computation-status The value is either the keyword `:in-progress` or the keyword `:finished`. It represents if the *master algorithm* thinks the computation is finished or not. If a slave reads a *resource file* with `:computation-status` being `:finished`, it will exit immediately with a status of zero.

:timestamp The value is an integer which represents the universal time when the file was written.

:member-id The value is a string which must match in the master and slave.

:update-interval The value is an integer which represents the number of seconds since the timestamp after which the resource file will be rewritten. The default is 300 seconds.

:slaves-needed This value represents the raw number of slaves the *master algorithm* has requested in order to complete its task.

:slave-executable This value is a list where the first element is a string representing the full path to the executable which is the slave executable, and the second element is a list of strings representing full paths to any shared libraries that have to be moved along with the executable.

:slave-arguments This value is a list of strings which are the command line arguments, in order, with which the slave is to be spawned.

An example file:

Listing 5.2: Contents of a sample *resource file*

```
;; Status of the computation
(:computation-status :in-progress)
;; Time Stamp of Resource File
(:timestamp 3488766071)
;; Member ID
(:member-id "default-member-id")
;; Update Interval (sec) of Resource File
(:update-interval 300)
;; Slaves Needed
(:slaves-needed 1000)
;; Slave Executable and Shared Libs
(:slave-executable
   ("/home/psilord/bin/a.out"
      ("/home/psilord/bin/libiolib-syscalls.so")))
;; Slave Arguments
(:slave-arguments ("--mw-slave" "--mw-master-host" "black"
                   "--mw-master-port" "47416"))
```

Appendix A

Example Application Descriptions

A.1 Hello-World

The canonical example detailed in this manual.

A.2 Ping

The *task algorithm* for this example returns `:ping-ok` if presented with a `:ping` argument, or otherwise `:ping-not-ok`. The interesting aspect of this example is the use of the *general target number* API for the in memory tasks. Billions of tasks can be run through this application, but only a small number are kept in memory at any give time to prevent memory exhaustion.

A.3 Monte-Carlo-Pi

This example implements the Monte Carlo algorithm to compute pi. Each task runs N trials and returns N and the number of trials in the circle. The master keeps a running sum of the total trials and the total number of trials in the circle. At the end of the maximum number of iterations, the approximation algorithm is performed with the computed ratio and the approximation to pi is produced. You may specify `----max-trial-sets` *integer* to the master process to state the total number of trial sets that must be performed.

The number of trials performed by each trial-set is non-configurable.

A.4 Higher-Order

This example shows that *task algorithm*s can be quite versatile. Here the *horder task algorithm* compiles a function presented to it as an argument and applies it to the data also presented to it returning the result of the application. The *master algorithm* creates a unique function for each task and associates it with the data for that task. All results are printed out when gotten back from the slaves. While this example shows the fundamental sketch of producing higher order *task algorithm*s, more work would be needed to handle signaled errors or other problems that could show up in the *task algorithm*.

Appendix B

Version History

Version 0.1
(Released 11/02/2010)

- **Info:** Initial release of CL-MW.

Appendix C

Acknowledgements

I would like to graciously thank: my wife Stephanie–who often put up with me vanishing for hours on end to write and test CL-MW, Greg Thain, Mick Beaver, and Alan De Smet, whom acted as sounding boards for the implementation and gave great feedback in the design of the system, manual, and how a user other than me would want to interact with it. In addition, I would like to thank the various denizens at `comp.lang.lisp` and `#lisp` for answering my many questions about Lisp.

CL-MW is not an official product from the Condor Project. It is written by me in my free time. If you would like to use a C++ version of the Master-Slave paradigm then check out `Condor-MW` from Condor's website.