User’s Guide to MW

A guide to using MW

The MW team

User's Guide to MW : A guide to using MW

by The MW team

Published 2005

Table of Contents

List of Tables

List of Examples

Chapter 1. Preface

MW is a software library that enables user's to quickly and easily create master-worker type parallel applications on Computational Grids. This user's guide is intended to help you get started using MW quickly. MW is an object-oriented C++ library providing abstractions to the master, worker, and task entities thta define a master-worker type application.

Here I will describe the chapters. For example, we begin in Chapter 2 with a quick getting started guide.

Prerequisites

This manual assumes you have a working knowledge of C++ and of Condor.

Notation and Naming Conventions

Throughout this manual we will use the following typographic rules:

- keywords, methods, functions, objects, and classes appear in a special `typeface`,
- larger program listings are specified in the text in special sections such as:

  ```
  this is code
  ```

  and
- important ideas are emphasized using *italics*.

Tested Platforms

MW is shipped as a set of source files. We have tried to keep this source code portable, so it should compile on any reasonable C++ compiler. However, as the MW-File module needs to link with the condor C++ userlog class, the compiler used for MW must be compatible with the compiler that built the libcondorapi.a library which ships with condor. None of othe other RMComm modules have this restriction. These are the following platforms MW has been tested on, all against Condor version 6.7.

Table 1-1. Tested Platforms

OS	CPU	Compiler	Notes
RedHat Linux 9	Intel x86	G++ 3.4.3	
Solaris 9	SPARC	G++ 2.95	
MAC OSX	PowerPC	G++ 1256	
Linux SLES	IA-64	G++ 3.2.2	

OS	CPU	Compiler	Notes
Linux RHAS 3	x86_64	G++ 3.2.3	32 or 64 bit mode, latter with restrictions
Windows XP	Intel x86	G++ 3.3.4	via cywgin

The License

Here we should include a copy of the MW license agreement.

Acknowledgements

MW was developed under the metaNEOS project (grant) and is suported by grants number ...

Chapter 2. A Quick-start guide to MW

This chapter explains how to get started using MW. It describes:

- MW theory of operation
- Selecting an RMComm implementation
- Verify MW prerequisites
- Download MW
- Configure MW
- Run a MW application in independent mode
- Run the same application in parallel under Condor
- Build your own application from a skeleton

MW Theory of operation

MW is conceptually simple. You break up a large computation into a number of idempotent tasks, to be executed in parallel. The tasks cannot communicate with each other directly, rather, they are given arguments by a single Driver process, and report their results back to the Driver when the task finishes. Each task runs within a shared context, set up once, allowing efficient sharing for large initial data sets. This style of computing is often called master-worker. In some ways, if normal Condor jobs are analogous to processes in a single operating system, these tasks are analogous to threads. The MW framework code works with Condor to find machine resources for these tasks, handle communication between the nodes, to re-assign tasks if their current machine fails, and generally manage the large parallel computation. MW provides hooks for you to save the state of the driver, so that if the driver, or its machine crashes, the computation can make progress upon driver restart.

The MW framework is a C++ library. To use it, you must subclass three major classes: MWTask, MWDriver, and MWWorker. As the names imply, the MWTask represents the basic unit of work with inputs and outputs to be marshaled to each distributed node. Your subclass of MWWorker provides the context for the task to run in. The MWframework finds machines to run your MWWorker on, and passes MWTasks for it to run. Your MWDriver subclass manages the whole process -- it is responsible for creating tasks, initially, statically, or also dynamically, in response to results from earlier tasks. It also collates the results of the various MWTasks, and decides when the computation is complete. Section XXX describes the interface you use to these classes in detail. MW provides a simple data marshaling interface independent of specific network implementation. This allows you to send data between the master and the workers without worrying about the specific communication implementation in use. And, you can change the communication paths without changing any of your user code.

Choosing an RMComm implementation

MW can run with one of several RMComm (Resource Management and Communication) implementations. This layer implements communication between the master and the workers, and the management of the worker machine resources -- e.g. allocation of CPUs, handling loss of a worker's CPU, etc. You should choose the RMComm implementation that works best for your system. The choices are: Independent, CondorPvm, Files, Files with Chirp, and Socket.

The Independent RMComm

The Independent RMComm is different from the rest. It is used for debugging user's MW code. Independent RMComm runs the master and the worker in a single executable in a single process on one machine. You should almost always start out with this implementation, as it is trivial to debug with a debugger, run profiling and memory checking tools, etc. Once your code is known to work in independent mode, you can select one of the other RMComm implementations to run in parallel on your distributed cluster.

To build your application in independent mode, you must first recompile all your MW C++ code with the **-DINDEPENDENT** flag. Then, link all of the resultant object files with all of the libraries in the MW `lib/` which end in _indp.a. The Makefiles in the `example` directories show you exactly how to do this. You can then run this executable like any other.

The CondorPVM RMComm

CondorPVM is a good default RMComm to choose. However, not all Condor platforms support CondorPVM. To check if your Condor installation supports CondorPVM run the command **condor_status -l**, and check look for the **StarterAbilityList** in the ClassAd for your machines. It should look something like: StarterAbilityList = "HasFileTransfer,HasCheckpointing,HasPVM" . If the **HasPVM** attribute is listed, then CondorPVM is supported. If it isn't, you must chose another RMComm implementation. CondorPVM supports relatively good communication performance, and, with user-level checkpointing, is robust in the face of master crashes. CondorPVM works a little differently than the other RMComm layers, in that the one user-written submit file runs in the PVM universe, and allocates all the machines. The first machine allocated becomes the master, the others, workers. In the other RMComm implementations, the user-written submit file runs in the scheduler universe on the submit machine, and it is the master. This distinction is important to remember when submitting to run on multiple architectures. (See the Section called *Running with Multiple Architectures* in Chapter 3). This master the explicitly runs **condor_submit** and **condor_rm** to allocate and free worker machines.

To use the CondorPVM RMComm, compile your user MW classes without any special flags. You must then create a master executable and and worker executable. Your master executable should be linked with the versions of the MW libraries in the `lib/` directory that end in condorpvmmaster.a, and your worker executable linked with those that end in condorpvmworker.a. To run your application, you must submit the master into the Condor pvm universe. That is, you must create a condor submit file, and run **condor_submit**, passing your submit file as an argument.

The Sockets RMComm

The Socket RMComm offers the lowest latency communication between the master and the workers. If your application is bound by network latency, that is, you have a large number of short-lived tasks, the Socket implementation may be the best choice. The Socket implementation runs in the vanilla condor universe, so every

Condor platform should be able to run it. However, the implementation assumes that the worker can initiate a TCP connection on a well known port (8997) to the master. If there is a firewall between the workers and the master, you will either need to open up this port in the firewall, or chose another RMComm implementation. If need be, it is easy enough to change this hard-coded value in the source code to another value, which might be more friendly to your firewall. As the Socket RMComm uses one socket for each worker to talk to the master, this may lead to a scalability problem on the master side, if your OS is not configured to support a large number of file descriptors per process.

To use the socket RMComm, compile your user MW classes without any special flags. You must then create a master executable and and worker executable. Your master executable should be linked with the versions of the MW libraries in the `lib/` directory that end in socketmaster.a, and your worker executable linked with those that end in socketworker.a. The socket master is a condor scheduler universe job. This means that it runs directly on the submit host. The socket master then explicitly launches the worker executables into the condor pool by running the condor_submit command itself, passing it a submit file it has created. Again, the example applications show sample submit files. Sometimes, for debugging purposes, it is easier to manually launch the socket master executable outside of condor. Because it is a scheduler universe job, there is nothing special that needs to be done -- it can simply be run from the shell, or a debugger, or within a profiling tool.

The Files RMComm

The MWFiles RMComm implementation is the most robust, but offers the slowest communication. To communicate, it uses one of two remote-i/o methods provided by Condor. For resource management, it parses the Condor userlog files. MWFiles relies on the Condor standard universe, which is not available on all Condor platforms. In particular, it not available on so-called "clipped" Condor ports. Also, running in the standard universe introduces some restrictions on system calls the worker program can use. Most notably, the worker cannot fork, or make long-lived socket connections. The condor manual has the full list of restrictions when running in the vanilla universe. However, because the usual condor communication mechanisms are used, all the condor communication benefits apply to the communication between master and worker. So, if Condor is configured to use encryption, authentication, or firewall-traversal, MW-Files gets this all, too. This is not the case for any of the other RMComm implementations. Also, the Files implementation is more robust than others. Because the workers are loosely-coupled, via files, to the worker, if the master crashes, the workers may be able to continue their computations, write them to the filesystem, and wait for the master to restart. In Sockets, or pvm, if the master dies, the workers quickly die too.

To use the Files RMComm, compile your user MW classes without any special flags. You must then create a master executable and and worker executable. Your master executable should be linked with the versions of the MW libraries in the `lib/` directory that end in filemaster.a, and your worker executable linked with those that end in fileworker.a. The file master is a condor scheduler universe job, just like the socket master is. As before, the example applications show sample submit files.

The Static-MPI RMComm

The Static-MPI RMComm is for users that would like to run an MW-enabled code in an MPI environment. The MPI (v1.0) specification did not allow for dynamic addition or removal of processors in the course of a computation, so the running environment in this case is static.

Verify MW Prerequisites

Before getting started with MW, you should have Condor installed, and have some basic understanding of how it works: how to submit jobs, monitor the queue and manage jobs. You do not need root. The Condor (http://www.cs.wisc.edu/condor) web page is the definitive resource for this. You will also need a basic understanding of writing and compiling C++ applications on Unix systems.

Download MW

The latest release of MW exists at MW web site. (http://www.cs.wisc.edu/condor/mw) MW is packaged at a ".tgz" file, so you can unpack it with the shell command **tar xvzf mw-version.tgz**. This will create a number of directories under the main directory `mw/`

`src/`

The code consisting of the main API code. The base classes you need to extend live here (and other classes as well). There are a number of subdirectories here too.

`doc/`

The documentation directory.

`examples/`

Example MW applications.

`examples/newskel`

A perl script and skeleton code you can use to create your own MW applications

Configuring MW

MW relies on the a configure script in order to create the Makefiles necessary to build on your system. **./configure --help** from the `mw` directory lists all the configuration options. Note that if you change any configuration options, you should first make clean, then make install, and finally re-link your MW application. In addition to the usual configuration options, here are the most often used options with MW

--prefix=`/some/absolute/path`

The most important option, and one which you almost always should use. The default, `/usr/local`, is almost certainly wrong, especially if you are not root, or want to install more than one version of MW on your system. The value is the directory which **make install** will create and populate `lib` and `include` directories under.

--with-condor=`/path/to/condor/release_dir`

If this option is not specified, configure will try to infer the correct value. If condor is not in your path, or if you have multiple versions of condor installed (perhaps a test and production condor), you should use this option to explicitly use one particular version.

--without-condor

Mutually exclusive with the above option. If condor is not installed, you may use this option, and a version of MW which only supports Independent mode will be built.

--with-pvm=`/path/to/condorPVM`

If you want the CondorPVM RMComm implementation, you should specify the path to your condorPVM libraries. The default value for this option is the value of the environment variable PVM_ROOT. You will need to set the environment variable PVM_ARCH to the correct value before running configure.

--without-pvm

If your Condor installation does not support CondorPVM, you should configure MW with this option. Mutually exclusive with the above.

--without-mwfile

Remove support for the MW-File RMComm from this configuration of MW

--with--mpi

Add support for building executables that will run with MPI. In order to (currently) use this, your environment must be set up to compile mpi programs via the **mpicxx** command.

Building MW

After you run **./configure** with the appropriate options, build MW with the make command from the top level. This will build the MW framework and the several example applications. There should be no compiler errors or warnings for either the examples, or the libraries. If there are errors, perhaps different **./configure** options will fix the problems. Once make builds clean, run make install, and make will copy the libraries and includes into the install location specified by the --prefix option to configure.

Running MW

In this section we describe the steps necessary to run a sample MW application using the different `MWRMComm` layers. The sample application that we will use is newmatmul.

Running with `MWIndependent` for debugging on one machine.

Running the independent version is very easy. First, make sure the build succeeded without errors as described above. Then, simply cd into the `newmatmul` directory under examples, and run the command

./masternewmatmul_indp < in_master . Note there is nothing inherent in MW that requires it to read from **stdin**, but this makes it easy to change certain parameters without recompilation.

The first think you'll notice about MW is that it is very verbose, and, by default, produces a lot of output about what it is doing. The program should complete very quickly, and near the end of the output, the driver prints the resulting matrix, after the line The resulting Matrix is as follows Note that you can run this under the debugger or profiler easily. This particular application reads from standard input. The file format of this input consists of two integers, which should always be "1", followed the name of the worker executable, which is also ignored for independent mode. This is followed by the number of rows in each matrix, followed by a partitioning factor, which sizes the amount of work each task does. The shipped input file uses "10" as the partition factor, which results in one task. You can lower this to "5" or "2", to see how multiple tasks are sent to the worker, and results returned to the driver. As independent mode only has one worker, these tasks are processed sequentially, but would be done in parallel for the other RMComm modes.

Running in parallel on many machines

Now that you've run MW in independent mode on one machine, and verified that the basic algorithms and mechanisms work, you can run MW and exploit available parallelism in your pool. Select those RMComm algorithms that are available at your site, based on the information described above. The default settings for the newmatmul example only create one task, so only one worker will be needed. Try lowering the partition factor to "1" to make the example have more tasks, and more available parallelism. You can use the **condor_q** command to see the various workers, and how condor assigns them to machines, as the computation progresses.

Running with MW-File RMComm

If your condor installation supports the Standard Universe, that is, the Condor platform is not a so-called "clipped" port, you can use the MW-File RMComm. For the newmatmul example, the master executable is called masternewmatmul_mwfile, and the worker is called workernewmatmul_mwfile. Submit the master executable to the condor scheduler universe by submitting the example file called submit_mwfile. You can do this with the condor command **condor_submit submit_mwfile**. You may want to change the email address specified in the **notify_user**, or remove it altogether.

Running with MW-Sockets RMComm

The Sockets RMComm implementation should work with every condor installation, barring any difficulties created by firewalls. The newmatmual example master executable is named masternewmatmul_socket, and the worker is workernewmatmul_socket. The condor submit file is named submit_socket, so you can submit this example to condor with the command **condor_submit submit_socket**.

Running with Condor-PVM RMComm

The Condor-PVM RMComm is a good starting choice, if it is supported by your Condor installation. Condor-PVM, unlike the others, runs in the PVM universe. The submit file specifies one machine, which is the master. The master then uses the Condor-PVM mechanisms to add machines to the computation dynamically. To run this example, simply **condor_submit submit_pvm**.

Running with StaticMPI RMComm

To run with an MW executable that has been enabled to run with MPI, you simply follow the standard instructions for running MPI programs at your site. For example, to run the example in Section the Section called *Branch-and-Bound: Solving `0-1` Knapsack Problem* in Chapter 4, you would run the command **mpirun -np 8 knap-mpi data/cir20.txt params.txt**

Using new_skel to create your own MW program

There is a fair amount of boilerplate code you need to write in order to get a trivial MW application running. In order to ease this, there is the **new_app** script in the directory `examples/newskel`. Simply **cd** to this directory, and run the command with the name of your new MW application. **TEST** might be a good first choice. Newskel will create a directory under examples called TEST, fill in all the requisite MW subclasses, and try to compile and build a simple application. You may need to re-run **./configure** by hand in this directory with appropriate options before you successfully **make** the skeleton, but other than that, the skeleton should be compilable. This trivial application is easy for you to extend to create your own MW computation. The following documentation assumes that you are using TEST as the name of your app, so you can replace this with the real name in all applicable places.

new_app creates the following source files for you to edit:

- MasterMain_TEST.C
- WorkerMain_TEST.C
- Driver_TEST.C
- Driver_TEST.h
- Task_TEST.C
- Task_TEST.h
- Worker_TEST.C
- Worker_TEST.h

`MasterMain_TEST.C` and `WorkerMain_TEST.C`

These two files simply provide the main entry point for the master and worker executables, respectively. The only line of code you may want to change is the line which configures the debug MWprintf capability in MW. set_MWprintf_level Generally, setting this to lower levels decreases the amount of debug output, and increasing it raises the threshold of debug messages. see XXX Link here in the reference menu for more details.

`Driver_TEST.C` and `Driver_TEST.h`

These two files subclass of MWDriver. Your Driver class is responsible for managing the various tasks which compose the parallel computation. The minimum set of methods you must implement to do this are:

MWReturn Driver_Test::get_userinfo(int argc, char **argv)

This method is the first that the driver calls on startup, and the command-line arguments from the driver executable are passed into it. get_userinfo should do the following, at minimum, but in this specific order:

- Set up the names and types of the worker executables. Use the following code as a template. The second parameter to set_arch_class_attributes is the Condor ClassAd requirements expression. Consult your condor manual for legal values here for other architectures. Note that the MW code parses this string, and looks for values for OPSYS and ARCH, so they should always be present.

  ```
  RMC->add_executable("worker-executable", "((OPSYS==\"LINUX\") && (ARCH==\"INTEL\"))");
  ```

- Set up the driver checkpoint frequency. To have the driver checkpoint every time 10 tasks complete, write the following line:

  ```
  RMC->set_checkpoint_frequency(10);
  ```

- Read in any driver-specific information from configuration files passed as command-line parameters
- Do all program-specific initialization
- Return OK if the initialization succeeded. Otherwise MW will not start.

MWReturn Driver_TEST::setup_initial_tasks(int *n_init , MWTask ***init_tasks)

This method is responsible for setting up the initial set of tasks to compute. It needs to allocate an appropriately sized (via n_init) array of properly constructed Task_TEST objects. The MW framework is responsible for deleting the tasks and the array when it is done with them.

MWReturn Driver_TEST::pack_worker_init_data(void)

It is common in MW applications that a large set of read-only data needs to be shared between all tasks. Rather than sending this data over to the worker with every task, and increasing network traffic, MW has the ability to send some initial data to the worker once. pack_worker_init_data does this. To do this, it should call the RMC routine (q.v.) pack, to send the data over. There is a corresponding unpack_worker_init_data routine in the worker, which should call unpack. It is vital that the packs on the Driver side match the unpacks on the worker side. As a trivial example, if you wanted to pass a single integer, you'd just write

```
int length = 100;
RMC->pack(&&;length, 1);
```

, and be sure to RMC->unpack it on the other side. As with the other routines that return a MWReturn, be sure to return OK if you want the MW computation to start.

MWReturn Driver_TEST::act_on_completed_task(MWTask *t)

When the worker is finished with the task, this function is called in the driver. What happens here depends on your application -- you can either just print a message here, or perhaps add new tasks, or remove existing tasks to the computation, based on the result of the Task object.

void Driver_TEST::printresults()

When all of your tasks have been completed, this function is called, then the master exits. This is a good place to print out and save your results.

`Task_TEST.C` and `Task_TEST.h`

These two files implement the Task class. This just represents the state of the task -- the Worker class contains the code that executes as task. The primary methods you need to implement in your Task class involve the saving and restoring of Task state to the RMComm and checkpointing streams.

The Big Three

> In C++ jargon, The Big Three methods are the assignment operator, and the copy constructor, and a virtual destructor. MW uses the copy constructor on tasks, so be sure to correctly implement all three, as the compiler-generated default is almost certainly not what you want.

void Task_TEST::pack_work(void)and void Task_TEST::unpack_work(void)

> These two methods are used to stream the contents of a task from the master to the worker, and back again. Note the relationship between these two methods and the Task constructor. Your User Driver class uses the Task constructor to create your Task objects with certain state, and which live in the master's address space. When the master wants to send a Task object to the worker, it calls pack_work on the existing task on the driver side, calls the default constructor on the worker side, and fills in the state of the task by calling unpack_work on the worker side. Thus, it is important that the pack calls on the driver side match up with the unpack calls on the worker side.

void Task_TEST::pack_results(void)and void Task_TEST::unpack_results(void)

> These two methods are very similar to the previous pair. However, these are used when the worker has finished computing the task, and needs to pass the results back to the master.

`Worker_TEST.h` and `Worker_TEST.C`

The final two files implement the Worker class. The main function of the Worker class is to execute tasks.

void Worker_TEST::unpack_init_data()

> This method is the inverse of Driver_Test::pack_init_data(), mentioned above. Use the RMC->unpack methods corresponding to the pack methods used on the master side to unmarshall the data it has sent.

void Worker_TEST::execute_task(MWTask *)

> This last method is the guts of your MW application. It uses the state in the MWTask it is given to compute something. You should store the results somewhere in the Task object, so the results will be streamed back to the master.

Chapter 3. Advanced Features of MW

Checkpointing long-running computations

Long running applications are, at some point, going to fail. Either a program bug, power loss, OS or network failure will cause the program to crash. If this happens on the worker side, it isn't too much of a problem: MW will automtically detect this, and resend that task onto another worker. A crash on the master is much more serious, though. To mitigate this risk, MW provides a mechanism for user-level checkpointing of the master state. Periodically, MW will open up a file named "checkpoint", write out all of its state to this file, and call a method in your Driver and unrun Tasks which should save state. When MW starts up, it checks to see if this file exists, and if it does, restarts from that state. To implement checkpointing, you simply need to set the checkpoint frequency, and to implement the following four methods.

void Driver_TEST::write_master_state(FILE *fp) and void Driver_TEST::read_master_state(FILE *fp)

The Driver class periodically calls write_master_state, which is part of the user-level checkpointing code. You should simply write out all current Driver state to the fp passed into this method. Its inverse function, read_master_state, will be called if the master needs to be restarted from a checkpoint file. You should make sure that the writes in the former all corresponds to reads in the later. Unless you have a tremendous amount of state in the Driver, it is usually best to write out in a simple, human-readable ASCII format. This also enables a weak form of "computational steering", if you kill the driver, and manually edit the checkpoint file,

void Task_TEST::write_ckpt_info(FILE *)and void Task_TEST::read_ckpt_info(FILE *)

These two methods are used to implement master-side checkpointing of tasks. Any data you need to write with pack_work and unpack_work should probably be saved to this file.

The user can control the frequency of the checkpoint in two ways: a checkpoint can be taken upon completion of a fixed number of tasks or after a specified time limit. The methods to use are the following:

int MWDriver::set_checkpoint_frequency(int freq)

Sets the number of task completions (calls to act_on_completed_task()) between checkpoints. The default frequency is zero (no checkpoints). A good place to set this is in get_userinfo().

set_checkpoint_time(int secs)

Set a time-based frequency for checkpoints. The time units are in seconds. A value of 0 "turns off" time-based checkpointing. Time-based checkpointing cannot be "turned on" unless the checkpoint_frequency is set to 0. A good place to do this is in get_userinfo().

Advanced MW users often wish to use the checkpointing as a kind of "computational steering"---stopping the computation in order to change some parameters. This can be done (if done with care), by resetting the appropriate parameters in read_master_state. Tasks in the checkpoint file will be ordered in the master task list according to the task_key function if one exists. (See the Section called *Task management*). To make this easier, the format of the checkpoint file should be in pure ascii. The built-in MW checkpointing functions all write their state this way, but if

the user code also does, steering via editing this file is much easier. Though this is somewhat less effecient in space and time, the time spent writing the checkpoint file is usually very small.

Statistics and Benchmarking with MW

Statistics in MW

During the course of a run, MW records statistics about the characteristics of the workers, the workers' state, and the worker's usage. These statistics are maintained in the `MWStatistics` class, and the user is encouraged to examine the source in the files `MWStats.h` and `MWStats.C`. Suppse that the job took a (total) wall clock time of `T`, and there was a set `W` of workers that participated in the computation. For each worker `j` in `W`, the following statistics are kept:

- u_j: Total (wall) time worker `j` was up
- c_j: Total (CPU) time worker `j` was executing tasks
- s_j: Total (wall) time worker `j` was suspended. *Note: suspended time is also counted as up time*
- b_j: Benchmark factor for worker `j`

At the end of each run, MW prints a summary of the statistics. In Table 3-1, we list these statistics and their meaning. For further discussion of the statistics relating to MW benchmarking, please see the discussion in the Section called *Benchmarking in MW*

Table 3-1. MW Statistics

MW output	Meaning
Number of (different) workers	`|W|`
Wall clock time for this job	`T`
Total time workers were alive (up)	$\sum_{j \in W} u_j$
Total cpu time used by all workers	$\sum_{j \in W} c_j$
Total time workers were suspended	$\sum_{j \in W} s_j$
Average benchmark factor	$\sum_{j \in W} b_j / \|W\|$
Equivalent benchmark factor	$\sum_{j \in W} c_j b_j / \sum_{j \in W} c_j$

MW output	Meaning
Minimum benchmark factor	$\min_{j \in W}\{b_j\}$
Maximum benchmark factor	$\max_{j \in W}\{b_j\}$
Average Number Present Workers	$\sum_{j \in W} u_j / T$
Average Number Nonsuspended Workers	$\sum_{j \in W} s_j / T$
Average Number Active Workers	$\sum_{j \in W} c_j / T$
Equivalent Pool Performance	$\left(\frac{\sum_{j \in W} c_j}{T}\right)\left(\frac{\sum_{j \in W} c_j b_j}{\sum_{j \in W} c_j}\right)$
Equivalent Run Time	$\sum_{j \in W} c_j b_j$
Overall Parallel Performance	$\sum_{j \in W} c_j / \sum_{j \in W} (u_j - s_j)$

MW will also print the "raw" statistics to standard output if the XXX link in here MWprintf_level is set greater than 20.

Benchmarking in MW

The heterogeneous and dynamic nature of a computational grid makes application performance difficult to assess. Standard performance measures such as wall clock time and cumulative CPU time do not separate application code performance from computing platform performance. By normalizing the CPU time spent on a given task with the performance of the corresponding worker, MW aggregates time statistics that are comparable between runs. The normalization factor can be based on vendor information such as MIPS or KFLOPS, if this information is available from the underlying Grid service software. Alternatively, MW allows the user to register an application specific benchmark task that is sent to all workers that join the computational pool. The speed at which the benchmark task is completed is used as the normalization factor.

In order to register a benchmark task with MW the user need only call the method `register_benchmark_task` in the `MWDriver` class. For example, in the `examples/knapsack/KnapMaster.C` code example included in the distribution, a benchmark task is registered with the call

```
register_benchmark_task(new KnapTask(KnapNode(instance_.getNumItems()), 10000));
```

which creates a task that is to evaluate 1000 nodes of the branch and bound search tree for the instance.

Resource Management

MW allows the user to specify the (target) number of workers that should participate in the computation. This is done with the method `set_target_num_workers( int num_workers )` in the `MWRMComm` class. For more control, the user can specify a target number of workers in each "class" of executable through the call `set_target_num_workers(int exec_class, int num_workers )`. The current target number that MW is striving for can be returned to the user with the call `get_target_num_workers(int exec_class = -1)`.

MW typically does not make all the resource requests at once, but rather in increments. The user can set the increment size with a call to the methos `set_worker_increment(int newinc)` in `MWRMComm` and retrieve the current increment by calling `set_worker_increment()`

Task affinity and grouping

Task management

If your tasks are completely independent of each other, there is nothing you need to do to manage them -- MW will run them all in no particular order. However, if your tasks do depend on one another, MW provides interfaces to ensure that they are queued in the correct order. This can improve your runtime, and in some cases, even the correctness of your code. All of these methods can only be called from the master process, and probably should be called from the act_on_completed_task method.

In MW the master class manages a list of uncompleted tasks and a list of workers. The default scheduling mechanism in MW is to simply assign the task at the head of the task list to the first idle worker in the worker list. However, MW gives flexibility to the user in the manner in which each of the lists are ordered.

```
#define MWKey double
MWDriver::set_task_key_function( MWKey (*)(MWTask *));
```

This method sets the function which maps a MWTask to a double, which can be subsequently used to sort or cull the list of outstanding tasks. In MW, tasks with small key value have higher priority than tasks with large key value. You may call set_task_key_function more than once during the master’s run to rearrange the task queue.

```
MWDriver::delete_tasks_worse_than(MWKey threshold);
```

Sometime, in act_on_completed_tasks, the result of a newly-finished task proves that certain pending MWTasks are now irrelevant. For example, assume each task searches to maximize a certain function. If one task returns a high value for that function, you can safely remove all pending tasks that search for values lower than your recently-found maximum. This method can be used to remove those tasks, if the set_task_key_function has been set up properly. Note that this function only removes pending tasks, not those which are already running on a worker.

```
enum MWTaskAdditionMode {
```

```
  /// Tasks will be added at the end of the list
  ADD_AT_END,
  /// Tasks will be added at the beginning
  ADD_AT_BEGIN,
  /// Tasks will be added based on their key (low keys before high keys)
  ADD_BY_KEY
};

enum MWTaskRetrievalMode {
  /// Task at head of list will be returned.
  GET_FROM_BEGIN,
  /// Task with lowest key will be retrieved
  GET_FROM_KEY
};

MWDriver::set_task_add_mode(MWTaskAdditionMode);
MWDriver::set_task_retrieve_mode(MWTaskRetrievalMode);
```

These methods control how the list of pending tasks is sorted. If the tasks have dependencies on each other, it is important to set these values up correctly. Otherwise, MW is free to run the tasks in any random order.

For advanced examples of using the task list management features of MW, please refer to the the Section called *Branch-and-Bound: Solving `0-1` Knapsack Problem* in Chapter 4

Running with Multiple Architectures

With MW, there can be worker exectables of varying operating system and architecture type. All of the executables must be linked to use the same the Section called *Choosing an RMComm implementation* in Chapter 2RMComm layer.

Special Consideration: When using the Condor-PVM RMComm to submit workers of multiple architectures, there must be multiple queue commands in the condor submission file. Example 3-1 is an example of a Condor-PVM submission file for a multi-architecture run. The other RMComms do not have this restriction -- you specify the architectures within the Driver code only.

Example 3-1. A Multi-architecture MW Condor-PVM submission file

```
  # This Condor Submit File will submit multiple architectures
  #  In MW -- the worker executables that will be run
  #  are
  universe = pvm
  executable = knap-master
  arguments = cir50.txt params.txt
  output = master.out
  error = worker.out
  log = condor.log

  # This is machine class "0" - Opteron
  requirements = (( arch == "x86_64" ) && ( opsys == "LINUX" ))
```

```
rank = KFlops
machine_count = 1..1
queue

# This is machine class "1" - Intel-Linux
requirements = (( arch == "INTEL" ) && ( opsys == "LINUX" ))
rank = KFlops
machine_count = 1..1
queue
```

The (worker) executables that will be run as a result of the queue command must be specified within the MW driver code itself. (Usually in `MWDriver::get_userinfo()`. Example 3-2 is an example of how the executables are requirements are registered with MW. *The order of the calls to `add_executable` must be the same as the order in the condor submission file.*

Example 3-2. Calling Sequence for Adding Executables with Different Requirements

```
RMC->add_executable("knap-worker-x86_64", "Arch == \"x86_64\" &&
Opsys == \"LINUX\"");
RMC->add_executable("knap-worker-x86_32", "Arch == \"INTEL\" && Opsys == \"LINUX\"");
```

Tips for higher throughput

MW has been used with thousands of concurrent workers. Careful tuning is often needed to effectively run with this many workers.

Configuring MW for highly parallel work

The first limit to set is the call to RMC->set_target_num_workers(int), which is an upper bound on the number of workers. The second limit is the host increment parameters with RMC->set_worker_increment(int) function. The default value of this is six workers, which mean that MW will only ask for six new workers at a time, and wait for condor to start those before asking for another six. With only six workers starting at once, rampup to hundreds of workers will be very slow. Setting this to 128 will get MW started much more quickly.

Configuring Condor for highly parallel MW work

In order to run with a large number of workers (more than 100), there is often some condor tuning needed.

From an administrative perspective, it can be easier to set up a personal condor just to run MW jobs, and set up that condor "pool" to flock to other pools, or accept glide-ins from elsewhere. This allows a user without Condor administratrive privileges to make changes to the condor pool running mw easily. The Condor manual describes how to set up a personal condor. Also, the personal condor will probably run entirely as a non-root user, which means that if operating system limits are hit (e.g. too many processes per user), it is much easier to recover.

The first setting is the MAX_JOBS_RUNNING setting in the schedd. The value needs to be set above the total number of concurrent workers. Setting it high is generally not harmful, so setting it to twice the expected number of workers is a good value. Note that when the master is submitted as a scheduler universe job, it counts against this limit.

When running with a large number of jobs, the condor schedd can get very busy. Because the schedd is single threaded, it can become unresponsive for minutes in rare situations. This cases often have to do with execute machines which have crashed at inopportune times. The schedd will eventually recover from these cases, but often it has to let multiple timeouts expire before returning to service commands like condor_q. One of these timeouts is specified by the condor_config variable SEC_TCP_SESSION_TIMEOUT. The default value for this is 20. Lowering this to 10 or 5 speeds up the schedd when it needs to handle killed workers.

If the schedd is unresponsive for a configurable amount of time, the condor master daemon assumes that it is "stuck", and will kill the schedd with a signal. The config parameter that controls this is named SCHEDD_NOT_RESPONDING_TIMEOUT. The default value is twenty minutes (the units are seconds). In rare cases, if the condor MasterLog reports that it is killing an unresponsive schedd, upping this parameter to a large value (say hours) will fix the problem.

By default, the condor schedd will only start one job every two seconds. If there are a lot of matches, this can slow startup. The parameter that controls this is JOB_START_DELAY. If the MW hostinc is a moderate number, JOB_START_DELAY can set to zero, which means all shadows are started as fast as possible If MW hostinc is larger, setting this to 1 (units are seconds) is a better value.

Configuring the OS for highly parallel work

Running a large number of workers can require operating system limits to be raised from the default values. For MW-Socket and MW-File, Condor will run one shadow process per worker. If the per-user limit on number of processes is low, this limits the number of workers. Also, the Condor schedd tries to estimate the amount of memory and swap space need, and if it thinks that running more shadows would exhaust either, it will stop spawning new workers.

MW-Socket uses one file descriptor per running worker. The number of file descriptors is a common limit for MW-Socket, and needs to be about 10 more than the peak number of workers, depending on how many file descriptors the user driver code needs.

Chapter 4. Advanced Examples Using MW

Branch-and-Bound: Solving `0-1` Knapsack Problem

In the `0-1` knapsack problem, there is a set of `N = {1, ... , n}` items each with weight a_i and profit c_i, a knapsack capacity `b`, and the objective is to fill the knapsack as profitably as possible. This can be formulated as the following mathematical program:

$$\max \left\{ \sum_{i \in N} c_i x_i \mid \sum_{i \in N} a_i x_i \leq b, x_i \in \{0,1\} \; \forall i \in N \right\}.$$

The knapsack problem is known to be NP-Hard, so it is unlikely that a "good" (polynomial time) algorithm exists for its solution. However, an efficient and popular way to solve this problem is through a procedure known as *branch-and-bound*.

Branch-and-Bound

Without loss of generality, we assume that the items are sorted in decreasing order of profit to weight ratio c_i/a_i. Branch-and-bound computes lower and upper bounds in the following fashion. The lower bound in the branch-and-bound algorithm is computed by greedily inserting items while the knapsack capacity is not exceeded. The upper bound in the algorithm is obtained by additionally inserting the fractional part of the last item that exceeds the knapsack capacity in order to fill the knapsack exactly. Note that in the solution there is at most one fractional item, which we denote as `f`. Therefore, the solution to the LP relaxation of the original knapsack problem is $x_i = 1$ for `i = 1, ... , f-1`,

$$x_f = \{b - \sum_{i=1}^{f-1} a_i\}/a_f,$$

and $x_i = 0$ for `i = f+1, ... , n`. The lower bound on the optimal solution value z_L is given by

$$z_L = \sum_{i=1}^{f-1} c_i,$$

and the upper bound on the optimal solution value z_U is given by

$$z_U = \sum_{i=1}^{f-1} c_i + (b - \sum_{i=1}^{f-1} a_i)\frac{c_f}{a_f}.$$

If all items fit into the knapsack `(f = n)`, then the lower and upper bound are

$$z_L = z_U = \sum_{i=1}^{f} c_i.$$

Task Management

For `MWKnap` we use a dynamic way to manage the task list through *task keys*. Each (derived) `MWTask` may be assigned a key value through the method MWDriver::set_task_key_function (MWKey(*)(MWTask *) key_func), where `key_func` is the address of a function that takes a pointer to a `MWTask` and returns the `MWKey` of the task, which is typed to be a double. The `task_key` may be changed dynamically during the course of the computation by using this method. The task list can be sorted through the method MWDriver::sort_task_list() , and once sorted, tasks

can be added and retrieved from the sorted list by task key value. In MW, the task list is sorted from smallest key value to largest key value.

One final method that can be of particular importance to branch-and-bound applications is a call that can delete all tasks in the task list whose key values are larger than a specified value: MWDriver::delete_tasks_worse_than(MWKey). Example 4-1 demonstrates the implementation of the MW task key features.

Example 4-1. The implementation of `task_key` in `KnapMaster`

```
MWKey upper_bound(MWTask *t)
{
  KnapTask *kt = dynamic_cast< KnapTask * > (t);
  assert(kt);
  return ((MWKey) (kt->getInputNode().getUpperBound()));
}

MWKey neg_upper_bound(MWTask *t)
{
  return (-upper_bound(t));
}

MWKey depth(MWTask *t)
{
  KnapTask *kt = dynamic_cast< KnapTask * > (t);
  assert(kt);
  return ((MWKey) (kt->getInputNode().getDepth()));
}

MWKey neg_depth(MWTask *t)
{
  return (-depth(t));
}

void
KnapMaster::setMasterNodeOrder(NodeHeap::Type type, bool sort_it)
{
  masterNodeOrder_ = type;
  switch(type) {
  case NodeHeap::VALUE:
    set_task_key_function(neg_upper_bound);
    break;
  case NodeHeap::WORST:
    set_task_key_function(upper_bound);
    break;
  case NodeHeap::DEPTH:
    set_task_key_function(neg_depth);
    break;
  default:
    assert(0);
  }

  if(sort_it) sort_task_list();
}
```

The ability to dynamically alter the task key during the course of the search is important for some branch-and-bound computations. For example, many branch-and-bound algorithms search the tree in a best-first manner. For large branch-and-bound trees, this can lead to the number of active nodes becoming very large, exhausting the available memory on the master processor. Instead, by dynamically altering the task list ordering, the user can adopt an approach where the nodes are searched best-first until the number of tasks at the master exceeds a “high-water” level `h`, and then the task list is reordered so that the tasks with the worst bound are searched. The task list can be kept in this order until its size of becomes smaller than a “low-water” level `l`, at which time the list can be reordered in a best-first fashion.

MW Implementation

MWTask

> We wish to parallelize a generic branch-and-bound algorithm for solving `0-1` within the master-worker framework by making the base unit of work a limited subtree. Thus, in our parallel implementation the algorithm becomes a {\em task}, with the exception that the grain size is controlled by specifying the maximum CPU time or maximum number of nodes that a worker is allowed to evaluate before reporting back to the master. In MW, there are two portions of a task, the work portion and the result portion. For our solver `MWKnap`, a `KnapTask` class is derived from the base `MWTask`, and the work portion of the `KnapTask` consists of a single input node. The result portion consists of an improved solution (if one is found), and a list containing the nodes of the input subtree that are not able to be evaluated before reaching the task’s node or time limit.

MWWorker

> In MW, the (pure virtual) MWWorker::execute_task(MWTask *task) method is entirely in the user’s control. Therefore, when implementing the branch-and-bound algorithm for which the task is to evaluate a subtree, the user is responsible for writing code to manage the heap of unevaluated subtree nodes. For `MWKnap`, we implement a heap structure using C++ Standard Template Library to maintain the set of active nodes. The heap can be ordered by either node depth or node upper bound, so we can quantify the effect of different worker node selection techniques on overall parallel efficiency.

MWDriver

> In `MWKnap`, a `KnapMaster` class is derived from the base `MWDriver` class. The MWDriver::act_on_completed_task(MWTask *t)} method is implemented to handle the results passing back from the workers which include updating the improved solution value, removing nodes in the master pool that have their upper bounds less than the current best solution value and adding new tasks.

Parallel Efficiency

In order to use the computational resources with maximum efficiency, the parallelization strategy of the branch-and-bound tree search has been carefully designed. Issue such as the proper ordering of the task list and the selection of the grain size are carefully considered in order to minimize communication overhead and contention at the master process without introducing large parallel search anomalies.

Contention

Since the master-worker paradigm is inherently not scalable. That is, for configurations consisting of a large number of workers, the master processor may be overwhelmed in dealing with requests from the workers and *contention* may occur. Many parallel branch-and-bound methods have a more loosely coupled form of coordinated control that allows for more scalability. It is our goal in this example to show the limits to which branch-and-bound algorithms can be scaled using the master-worker paradigm, with a well-engineered version of the algorithm running on a computational grid.

The lack of scalability of the master-worker paradigm comes from the bottleneck of a single master process serving many worker requests. The contention problem can be quite serious in a grid computing environment, as our goal is to have hundreds or thousands of workers served by a single master. To ease the contention problem, it is useful to think of the master-worker paradigm as a simple G/G/1 queueing model. There are two ways to increase the efficiency of the model:

- Decrease the arrival rate. This can be accomplished by increasing the *grain size* of the computation. In the context of branch-and-bound, the grain size can be increased by making the base unit of work in the parallel branch-and-bound algorithm a *subtree*, not a single node. The grain size can be limited by giving an upper bound on the CPU time (or number of nodes) spent evaluating the subtree.
- Increase the service rate. This can be accomplished by searching the subtrees in a depth-first manner. Searching the subtrees depth-first minimizes the number of nodes that will be left unexplored if the evaluation limit on the subtree is reached. This has two positive effects for increasing the service rate of the master processor. First, the size of the messages passed to the master is reduced, and second, the size of the list of unexplored nodes on the master is kept small.

Clean-up

The unit of work in our parallel branch-and-bound algorithm is a time or node limited subtree in order to ease contention effects at the master. However, a subtle point as regards to this strategy is that even though we may wish a worker to evaluate a subtree for γ seconds, it may take significantly less than γ seconds to completely evaluate and fathom the subtree. Somehow, we would like to ensure that if a node enters the master's task queue, then it is likely that it will require the full time γ (or close to the full time γ to evaluate. This is accomplished with a second (or clean-up) phase in every task. The goal of the clean-up phase is to fathom nodes that are unlikely to lead to full-length tasks. Nodes deeper in the branch-and-bound tree are likely to lead to short tasks, so in the clean-up phase, the focus is on evaluating these nodes.

Ramp-up and Ramp-down

Contention is not the only issue that may cause a lack of efficiency of a parallel branch-and-bound algorithm. Ramp-up and ramp-down, referring to the times at the beginning and the end of the computation when there are more processors available than active nodes of the search tree, can also reduce efficiency. A simple and effective way to deal with these issues is to exploit the fact that the grain size of the branch-and-bound algorithm can be dynamically altered. If the number of tasks in the master's list is less than α, the maximum task time is set to a small number of seconds β. Note that this strategy works to improve the efficiency in both the ramp-up and ramp-down phases.

Chapter 5. Using the MW API

In this chapter, we describe MW's application programming interface.

`MWDriver`

Pure virtual methods... Checkpointint... Task List Management... Other important stuff? It is not necessary to document every method -- that is for the reference manual/doxygen? Though we should be able to produce a reasnable sized reference manual from doxygen as well...