# Design and Implementation of GXP Make —a Workflow System Based on Make

Kenjiro Taura, Takuya Matsuzaki, Makoto Miwa, Yoshikazu Kamoshida,
Daisaku Yokoyama, Nan Dun, Takeshi Shibata, Choi Sung Jun, and Jun'ichi Tsujii
University of Tokyo
7-1-3 Hongo Bunkyo-ku, Tokyo 113-0033, Japan
Contact Email: `tau@logos.t.u-tokyo.ac.jp`

***Abstract*—This paper describes the rational behind designing workflow systems based on the Unix make by showing a number of idioms useful for workflows comprising many tasks. It also demonstrates a specific design and implementation of such a workflow system called GXP make. GXP make supports all the features of GNU make and extends its platforms from single node systems to clusters, clouds, supercomputers, and distributed systems. Interestingly, it is achieved by a very small code base that does not modify GNU make implementation at all. While being not ideal for performance, it achieved a useful performance and scalability of dispatching one million tasks in approximately 16,000 seconds (60 tasks per second, including dependence analysis) on an 8 core Intel Nehalem node. For real applications, recognition and classification of protein-protein interactions from biomedical texts on a supercomputer with more than 8,000 cores are described.**

## I. Introduction

Scientific workflows generally consist of many individually developed components. The primary role of scientific workflow systems is to facilitate composition of such components. A number of workflow systems have been proposed with different focuses, but we feel they are still not as accessible and user-friendly as they should be. More specifically, many systems suffer from one or more of the following problems.

**Setup Cost:** They are often built on a heavy stack of software layers that take a significant amount of effort to set up. For example, many systems require a batch scheduler or a Grid middleware that take a fair amount of installation effort on individual compute nodes and an even larger cost on the master node. Systems based on web services require installing and configuring an application server.

**Development Cost:** They sometimes lack a workflow description language sufficiently expressive and convenient. Some systems employ an XML-based syntax that is full of tags enclosing small pieces of information; some lack dynamic constructs to concisely describe many tasks instantiated from a template, forcing users to manually describe all individual tasks; some require individual components to be wrapped (e.g. by a Java class) to conform to their component models.

**Learning Cost:** Besides learning a workflow language, users often need to learn specifics of the surrounding environment such as batch schedulers.

Such problems are often overlooked for the ambitious project goal of being comprehensive tools. In reality, however, users often like to start developing their workflows in their personal or laboratory machines with a minimum learning/physical investment. They like to have a short turnaround time until they get the first result and a smooth transition path to larger systems when a need arises. Workflow systems thus need to be accordingly designed for such criterion as a minimum installation cost especially on small systems, a small initial learning cost, a straightforward composition of ordinary executables, and a smooth transition path to larger environments.

This paper describes a workflow system called "GXP make," which takes these design goals driving principles rather than afterthoughts. As a workflow description language, it simply uses the Unix make, which is not only popular among programmers but also very "straight to the point," thus easy to start with for inexperienced users too. Moreover, GXP make uses an existing implementation of make (GNU make) *without any modification*. This achieves an almost complete compatibility with the current and future versions of GNU make. GXP make is a part of a software package called GXP and it is internally a thin layer on top of an underlying function called "GXP shell" [1]. As its name suggests, GXP shell exposes a shell-like remote process invocation command, making it uniform across a range of systems (a single multicore node, Beowulf cluster with SSH accesses, batch-scheduled clusters, and distributed environments consisting of multiple sites). Finally, it automatically installs itself to individual compute nodes on demand, without assuming any shared file system between the local node and the compute nodes. The net result is a lightweight workflow system that, on most Unix platforms, only takes a download to a single node for installation, little initial learning at least for those who have some experiences in command-line interfaces of Unix, and no additional work to migrate to clusters with a shared file system. GXP is written in Python, so it needs no compilation. Like the original make, GXP make currently assumes an underlying shared file system. This is typically not an issue within a single cluster or a supercomputer, which usually operates one of mature networked file systems (e.g. NFS, Lustre [2], and GPFS [3]). With multiple clusters, we routinely experiment with SSHFS [4] and Gfarm [5]. We are working on a user-level distributed file system called GMount [6], again focusing on low setup

978-0-7695-4290-4/10 $26.00 © 2010 IEEE
DOI 10.1109/eScience.2010.43

TABLE I
WORKFLOW SYSTEMS

| | workflow desc. | primary component | target env. |
|---|---|---|---|
| GXP make | make | executable | HPC |
| Swift[8] | SwiftScript | executable | HPC |
| Dryad[9] | C++ | executable | HPC |
| Xcrypt[10] | Perl dialect | executable | HPC |
| Hadoop[11] | N/A (fixed) | Java class | HPC |
| Taverna[12] | GUI | Web service | WWW |
| Triana[13] | GUI | Java class | LCS |
| Kepler[14] | GUI | Java class | LCS |
| Pwrake[15] | Rake | executable | HPC |
| makeflow[16] | make-like | executable | LCS |
| SGE qmake[17] | make | executable | HPC |
| DAGMan[18] | static DAG | executable | LCS |
| Pegasus[19] | static DAG | executable | LCS |

cost. Using a system call interceptor such as Parrot [7] could be another interesting approach.

This paper makes the following contributions.

- It demonstrates the suitability of make as a workflow language. It specifically shows there are many features not so commonly known but quite useful to describe large workflows with many tasks.
- It shows how to extend function of GNU make, which originally supports parallel execution within a single node, with parallel execution across nodes.
- It analyzes performance and scalability limits of the resulting system.
- It describes a case study of a real natural language processing application.

## II. Related Work

Workflow systems have different target scenarios and design emphases, but their basic function is common; they maintain a DAG of 'tasks' and execute those whose dependencies are met. They are categorized by at least the following three aspects. Table I gives a brief summary.

1) DAG description interfaces or languages, i.e., how users express tasks and communication thereof.
2) Component models, i.e., how users create a component instantiated to tasks. They may be regular executables, Java classes, or Web services.
3) Main target environments, e.g., High Performance Computing environments (HPC), Loosely Coupled Systems (LCS), World Wide Web (WWW), etc. While there are no strict boundaries between these environments, HPC generally refers to a cluster or a supercomputer often sharing a single file system and LCS a collection of resources distributed in a campus wide scale.

Our work is in spirit closest to Swift scripting language [8] and its dispatcher Falkon [20]. They focus on massively parallel computation out of regular executables and their high performance execution in HPC environments. Swift is a scripting language specifically designed for describing workflows. A similar approach is to embed a DAG description library into an existing programming language, as demonstrated by Dryad [9] and Xcrypt [10]. Dryad is a C++ library to compose DAGs and Xcrypt a perl extension. Like Dryad and Xcrypt, we leverage an existing syntax and tool (make) to minimize learning barriers. Neither did we implement a parser/interpreter of make but only implemented a dispatcher.

MapReduce[21] was shown to be a powerful framework that can expresses many algorithms by customizing just a few elementary components (namely, map, reduce, and a few others). As a model of computation, MapReduce can be viewed as a special type of workflows following a fixed template (i.e. $m$ map tasks each connected to $r$ reduce tasks). Not surprisingly, make is expressive enough to encode MapReduce, as shown in Section III. To be fair, the focus of MapReduce is on high performance execution of data-intensive computation rather than rapid and flexible compositions of existing executables.

Taverna [12], Triana [13], and Kepler [14] have many common characteristics. They all provide GUI for composing workflows with boxes and connectors. They have underlying languages based on XML, but they are very tedious to read/write. Components are primarily either web services or Java classes adhering to specific conventions. Either case, integrating a normal command as a component is not as straightforward as just using it from make. Due to its emphasis on integrating remote web services, implementation is not geared toward high throughput and many-task computing. They invoke either a SOAP-based RPC or a Grid job submission for each task.

Parallel and distributed make is an old idea [22], [23], [24], and it is recently used as the basis of workflow systems [16], [15]. Sun Grid Engine [17] also supports a parallel make in clusters (qmake). GNU make actually has an old undocumented configuration option, turned off by default, for distributed make based on a library called Custom. Among them, SGE qmake [17] and makeflow [16] are the closest to GXP make. For comparison, SGE qmake only supports SGE and since it dispatches individual jobs via SGE, its throughput is limited by that of SGE, which is a few jobs per second. Makeflow is a job description language similar to make but lacks features useful for describing large workflows. They include pattern rules and wildcards, which we show are very important in Section III.

## III. Describing Workflows with Make

### A. Make Basics and Simple Examples

An input file of make, commonly called a *makefile*, describes workflows with a set of *rules*. A rule specifies a *target* file, files it depends on called *prerequisites*, and a command (or a series thereof) to create the target from its prerequisites. In essence, a makefile describes dependencies between files and actions to resolve them. Given a set of rules described in a makefile, files that should be created, and the current file system state (i.e. existence and timestamp of files), make calculates commands to invoke and their order.

GNU make supports *pattern rules* (rule templates) to effectively specify many rules with a single description. This feature is particularly useful for many-task computation. GNU make supports two kinds of pattern rules, one called *static* pattern rules and the other called *implicit* pattern rules (similar to the traditional *suffix* rules). Differences between them are subtle, and this paper only explains the former.

The syntax of static pattern rules is similar to that of ordinary rules, except for the following.

- A static pattern rule specifies a list of files to which the pattern applies,
- the target may contain a single '`%`' character that matches any non-empty string, and
- the prerequisites may also contain `%` character(s) representing the string that matched the `%` character in the target.

A command-line can refer to the string that matched the `%` character by the built-in variable '`$*`'. GNU make has a filename globbing function (similar to that of shells), a function that returns the standard output of arbitrary shell commands, and many functions that transform a list of strings. Combination of these functions and pattern rules makes make particularly convenient for describing workflows of many tasks.

### B. Pattern Rules for Many Tasks

Here is a small yet complete makefile that applies 'bwa' command to all files that have suffix '`.fastq`' in the current directory.

```
fastq:=$(wildcard *.fastq)
sais:=$(fastq:.fastq=.sai)
all : $(sais)
$(sais) : %.sai : %.fastq
    bwa aln hg19.fa $*.fastq -f $*.sai
```

The `$(wildcard *.fastq)` is an invocation of a filename globbing function that expands into a list of files matching the pattern `*.fastq`. The `$(fastq:.fastq=.sai)` in the second line substitutes `.sai` for every occurrence of `.fastq` in the `$(fastq)` list. This is a common template for data parallel workflows.

The almost identical template can be used to describe so-called "parameter-sweep" workflows, with the only difference being that we use a function to generate parameters (most typically by a shell command) rather than a file globbing.

```
x:=$(shell seq 1 10)
results:=$(addsuffix .res,$(x))
all : $(results)
$(results) : %.res :
    simulate --seed $* > $@
```

The right hand side of the first line expands to the output of `seq 1 10`, which is `1 2 ... 10` and that of the second line `1.res 2.res ... 10.res`, which becomes the list of target files.

### C. Encapsulating Parameter Sweep and Data Parallelism

With only pattern rules, describing tasks that take multiple parameters is not straightforward. For example, let's say we have two parameters $x$ and $y$ both taking one

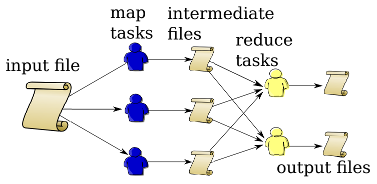

Fig. 1. A DAG representation of MapReduce

of `Cameroon, Denmark, Japan,` and `Netherlands,` and we would like to execute commands '`match` $x$ $y$' for all combinations of $x$ and $y$. That is, we would like to have a concise description of tasks that would be equivalent to the following.

```
Cameroon-Cameroon :
    match Cameroon Cameroon
Cameroon-Denmark :
    match Cameroon Denmark
  ...
Netherlands-Netherland :
    match Netherlands Netherlands
```

A pattern rule cannot apply straightforwardly to this example because the pattern rule can have only one placeholder (i.e. `%`). A more powerful `define` primitive can be used to write a template taking any number of parameters. The template of the above rules can be written as follows.

```
define match_rule
$(1)-$(2) :
    match $(1) $(2)
endef
```

A built-in function `call` instantiates the body of a `define` and `eval` evaluates the argument string as if it is written in the makefile. Finally, `foreach` built-in function is the make's loop construct. Putting them together, our example can be written as follows.

```
teams:=Cameroon Denmark Japan Netherlands
$(foreach x,$(teams),\
  $(foreach y,$(teams),\
    $(eval $(call match_rule,$(x),$(y)))))
```

It is general and conceptually simple, but admittedly not elegant in terms of syntax. Since this is very common in parameter-sweep workflows, GXP make encapsulates this pattern into a convenient library, with which this example can look more straightforward.

```
parameters:=x y
x:=Cameroon Denmark Japan Netherlands
y:=Cameroon Denmark Japan Netherlands
cmd=match $(x) $(y)
include $(GXP_MAKE_PP)
```

All the mechanism is in an include file `$(GXP_MAKE_PP)`. Due to space limitation, we refer the interested readers to GXP distribution [25] for implementation details. .

### D. MapReduce

Given the power of user-defined macros (`define`), its invocation (`call`), and a loop (`foreach`), it is not surprising that MapReduce can also be expressed with them. Fig. 1 represents MapReduce in a DAG. GXP make allows users to specify arbitrary command-lines for map and reduce tasks, giving a framework similar to Hadoop's streaming API. It is encapsulated in a library just like the above parameter-sweep framework. Here is the famous word count example that prints the number of occurrences of each word in the text.

```
input:=big.txt
output:=wc.txt
mapper:=awk -f wc_mapper
reducer:=awk -f wc_reducer
include $(GXP_MAKE_MAPRED)
```

We again refer the interested readers to [25] for its implementation. By default, the mapper is any command-line that reads lines from its standard input and writes key-value pairs to its standard output. Each line of the output should contain a single key-value pair, with its key in the first column, followed by the value. The reducer is any command-line that reads key-value pairs from its standard input in the ascending key order, and writes key-value pairs to its standard output. A fully functioning implementation of `wc_mapper` and `wc_reducer` in AWK is as follows.

**mapper:**

```
 { for (i = 1; i <= NF; i++) print $i,1; }
```

**reducer:**

```
{ if (k != $1) {
      if (k != "") print k,v;
      k = $1; v = 0; }
  v += $2; }
END { if (k != "") print k,v; }
```

While GXP make's MapReduce implementation is experimental and nothing like a replacement of Hadoop, it has advantages of being small and simple. It inherits the GXP's advantage of requiring no installation on compute nodes; it can run in any environment GXP supports (e.g. batch scheduled supercomputer) and in particular independent from the underlying file system. For example, it can use an HPC file system such as Lustre. It also inherits the benefit of being based on make; it allows users to customize other components, including input readers, partitioners, sorters, and mergers, all with a matter of a single assignment prior to the include statement. Customized components can also be arbitrary executables and interaction between them can be completely understood by dry-running the workflow (i.e. by '`make -n`'). This is in contrast to Hadoop streaming in which only mappers and reducers can be arbitrary executables but other components can only be written in Java after learning the structure of its class library.

### *E. Summary: Why Make is so Useful for Workflows?*

We believe make is useful not just because it is already there in many environments, but also because it has many features that fit workflows.

**Data-oriented declarative view:** Most workflow systems are "task-oriented," in the sense that users specify tasks that should be performed. In practice, they depend on file system states such as available input files and intermediate/output files already produced, perhaps by previous runs of the same workflow or other workflows. In most workflow systems, it is the user's responsibility to derive tasks to execute from such file system states. In contrast, make allows users to specify "result they eventually want." The make system calculates tasks that should be run based on file system states. As a result, workflows become more declarative and their reusability enhanced.

**Concise description:** As we have seen, makefile has powerful constructs that make workflow descriptions much more concise and less error-prone than some other systems. Some systems lack mechanisms analogous to pattern rules (or `define`) and many systems adopt XML-based syntax, which are if not impossible very tedious to write manually.

**Fault tolerance:** The execution model of make automatically implies a limited form of fault tolerance. After a job fails for any reason and so does the entire make eventually, simply running the same make again continues the workflow from the point it left off. Similar observations are made in [18] and [8].

**Dry run:** Make supports an option ('`-n`') that shows commands that will be executed without actually running them. Obviously, the feature is quite useful for debugging.

**Flexible parallelism control:** Make supports an option ('`-j`'), which specifies the maximum parallelism (the number of outstanding tasks). Switching between parallel and serial executions is also trivial.

## IV. Design of GXP Make

### *A. Design Features*

As we have seen in the previous section, while the core functions of make looks simple at a first glance, it is a number of apparently minor features described in the previous section that makes make particularly powerful for scientific workflows. The design philosophy of GXP make is to carry *all* functions of GNU make, from single node multicore systems to clusters and distributed environments. Specific features are the following.

**Full Compatibility with GNU Make:** From the beginning, the goal is to make the migration path from GNU make on single node to GXP make on multi node environments as smooth as possible. To this end, GXP make is made fully compatible with GNU make by its construction.

**Ease of Installation:** The setup cost of using GXP make is kept minimum. All that is required is to download GXP tarball on a single node and have GNU make ready on that node. Specifically, GXP make requires *no* prior setup on remote compute nodes, either by the user or the administrator, besides the following two obvious requirements; (1) The user has a means to remotely execute a command on the remote compute nodes. It is usually either a remote shell (e.g. `ssh`) or job submission command of a local batch scheduler (e.g. `qsub`). We collectively call them remote-exec commands hereafter. (2) They have a Python interpreter, since GXP is written in Python.

**Portability and Uniform Interface across Platforms:** GXP make supports a uniform user interface across a range of platforms including multi-core single nodes, Beowulf clusters accessed by SSH, and shared clusters or supercomputers managed by a resource manager (a batch scheduler). When a user migrates from one environment to another, he can specify the difference with a matter of a few lines completely independent from workflow descriptions.

**Fast Dispatching:** Batch schedulers and Grid middlewares have varying dispatching latencies and throughput, and some have poor performance [20]. Workflow systems dispatching individual tasks to the underlying batch scheduler [14], [13], [19], [18] easily hit these limitations. GXP make acquires resources by an underlying remote-exec command only once per a unit of resource (typically a single node) and dispatches tasks of a workflow without invoking it each time. This two level approach was successfully used in Falkon [20] for fast dispatching, and in Condor Gliding In [26] for interfacing with resources managed by different resource managers.

### B. User Interface

This section explains interface of GXP shell and GXP make from a user's perspective. Assume he can access machines either via SSH or TORQUE batch scheduler, but the system does not have GXP installed. The steps until running his first workflow are completely described below.

**Step 1 (Installation):** Installation takes only downloading GXP to a single node where he intends to interact with GXP (hereafter called the *home node*). He needs no work on individual compute nodes. To interact with GXP, he issues `gxpc` command from his favorite shell. All functions of GXP are accessible via a sub-command of `gxpc`.

**Step 2 (Configuration):** He tells GXP how it can reach a particular resource. This is done by the `use` sub-command. Here is an example specifying that resources whose name begin with *tokyo* can be reachable from resources whose name begin with *amsterdam* via SSH.

```
gxpc use ssh amsterdam tokyo
```

Each term (i.e. `amsterdam` and `tokyo` above) is actually interpreted as a regular expression; it can be made more specific and/or reusable.

The following two lines describe another environment in which a cluster's gateway host (`cluster-gw`) can be reached from the current host via SSH, and cluster nodes, which we assume are named like `cluster-node000, cluster-node001, ...,` can be reached from the gateway or another cluster node via TORQUE.

```
gxpc use ssh `hostname` cluster-gw
gxpc use torque cluster cluster-node
```

He can issue any number of such `use` commands to flexibly specify the resource usage conventions/restrictions of the environment. In essence, they specify an edge-labeled graph of resources in which an edge $A \overset{m}{\to} B$ represents $A$ can reach $B$ by issuing $m$ (e.g. `ssh`).

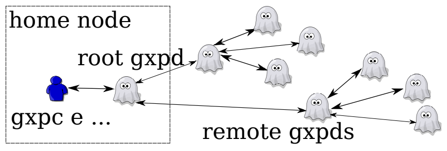

Fig. 2. Process Structure of `gxpc` and `gxpd`

The setting persists only until he quits the GXP "session", which automatically starts when `gxpc` finds no sessions and ends with an explicit `gxpc quit` command. To avoid repeating this setting on every session, they can of course be stored in a shell script file. It is only a few lines of `gxpc use` commands that need to be different from an environment to another. Resource acquisitions, command invocations, and workflow executions that follow can be done identically regardless of the configuration. Besides `ssh` and `torque`, GXP includes supports of `sge` (Sun Grid Engine), `condor` (Condor), `sh` (shell), and systems customized for several supercomputer sites.

**Step 3 (Resource Acquisition):** After telling GXP how it can reach resources, he can actually acquire resources by issuing `explore` sub-command. The following tries to acquire 8 resources named `cluster-node`, as well as a resource named `cluster-gw`.

```
gxpc explore cluster-gw cluster-node 8
```

It is at this point that GXP invokes `ssh` and `qsub` to reach `cluster-gw` and eight `cluster-node`'s, respectively. On each resource successfully acquired via `explore` command, a process we call *GXP daemon*, or `gxpd`, brings up. Users normally consider a single `gxpd` a representation of a node, but it is sometimes useful to bring up multiple daemons on a single physical node. All GXP daemons form a tree rooted at the home node and the tree serves as the message channel to communicate with daemons (Fig. 2).

**Step 4 (Workflow Execution):** Finally, he writes a regular makefile and issues `make` sub-command to run the workflow with resources acquired. The simplest example is this.

```
gxpc make -j
```

All but the last step above is an interface of GXP shell, whose role here is to form a pool of resources and expose a uniform interface to launch remote processes on them. With some resources acquired with `explore`, he can launch processes on them using `e` sub-command of GXP shell. By default, a command is executed by all GXP daemons he acquired in the session, but there are many ways to specify a subset of them, such as those based on host names, unique names given to them, etc. For example,

```
gxpc -t cluster-node e hostname
```

lets all GXP daemons acquired by names matching `cluster-node` run `hostname` command;

```
gxpc -g tokyo-tau-2010-...-3700 \
  e hostname
```

specifies a GXP daemon whose uniquely assigned name matches `tokyo-tau-2010-...-3700` (omitted due to page width limit). This is useful to specify exactly one GXP daemon. GXP make uses it when dispatching a task of a workflow.

## V. Implementation

This section illustrates the main points of GXP implementation, focusing on how it achieves features claimed above. It also reveals its architecture necessary to understand the current performance/scalability limitations.

### A. *Implementing* `e`

Launching a process on a remote node via `e` is a simple and lightweight event. As previously described, it is an invocation of `explore` sub-command that triggers resource acquisition by a remote-exec command. After acquiring some resources, GXP daemons stay there until the user quits the session. A subsequent invocation of `gxpc e` command simply sends a command message to the GXP daemon at the home node (hereafter called *the root daemon*). It then forwards the message to all the designated GXP daemons down in the tree. Thus, `gxpc e` is much lighter than invoking `ssh` or `qsub`. Given a command message, a GXP daemon launches a process executing it and forwards its standard out/err upward to the root daemon, which then forwards them to the `gxpc e` process. It also detects the termination of the process and similarly sends its status to the root daemon. `gxpc e` exits after collecting statuses of all the processes invoked as the result of it.

### B. *Implementing* `explore`

The goal of `explore` command is to bring up GXP daemons on all targets specified by the command-line. It determines which GXP daemons should invoke which underlying remote-exec command to bring up new daemons, consulting the configuration expressed by `use` commands. Thus, an elementary function that needs to be implemented is to bring up a new GXP daemon on a remote host by invoking the specified remote-exec command. An interesting issue is how to do it without requiring the user to install a GXP daemon program on the remote host in advance nor assuming a shared file system between the local and the remote host.

You might imagine it is as simple as invoking an `scp` command or alike to copy the GXP daemon program, but it is much more complex. First, a particular file transfer command may be unavailable or disallowed between some hosts. Let us take `scp` as a particular example. It is a subsystem of SSH, so it is simply unreasonable to assume it gets through to hosts to which `ssh` does not. Letting users specify a file transfer command that should be used is too user-unfriendly. More importantly, it is not a simple file copy that is required, but a more intelligent remote installer that first checks if the GXP daemon program has already been copied, copies it to a unique temporary directory (to avoid race conditions) only when this is not the case, and finally invokes the copy. Note that we need to do it within a *single* invocation of a remote-exec command, to guarantee that they all happen in the same node. We developed a bootstrapping mechanism to achieve it under the minimum assumptions that *GXP already relies on*—availability of a Python interpreter and a means to remotely invoke it.

We invoke a Python interpreter with an option '`-c`', which takes a string as an argument and executes it as a Python program. Since strings that can be written in a shell command-line can be limited in size (among others), we pass a small stub code that reads a number of characters from its standard input and then executes it. That is, we invoke a remote python interpreter by a command-line like this.

```
ssh target python -c \
'import os; exec(os.read(N))'
```

This is an example for SSH and the exact command differs depending on the underlying remote-exec command used. $N$ is replaced with the number of bytes to feed. `exec` is a Python function that executes a string as a Python program. Now the client (the host that issued the above `ssh` command) feeds the real installer program to its standard input. The real installer program performs the actions outlined above and finally calls the '`execv`' system call to invoke the GXP daemon program just installed.

### C. *Implementing* `make`

`gxpc make` invokes GNU make passing all arguments to it. In addition, `gxpc make` invokes a dispatcher, named `xmake`. `xmake` is a simple event-driven program that behaves as follows.

- It listens to a Unix domain socket to which external clients will connect. The socket can only be written by the user.
- Upon accepting a connection from a client, it receives tasks. They originate from GNU make, so their dependencies have been met (i.e. they are ready to go). The exact mechanism in which GNU make sends tasks to it will be described shortly.
- It performs match-makings between tasks and available resources.
- It dispatches tasks to the resource selected, by invoking `gxpc e` command.
- It detects termination of tasks and collects their exit statuses.
- For each task terminated, it sends back the task's status to the client that sent it.

A missing piece is how to let the unmodified GNU make send tasks to `xmake`, rather than directly executing it on the local node. When GNU make decides to launch a command, say `hostname`, it invokes the following command-line.

```
sh -c hostname
```

The good news is GNU make allows the user to customize the name of the shell. Specifically, if it finds a definition

```
SHELL=S
```

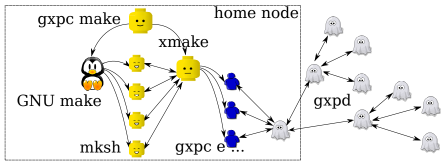

Fig. 3. Process Structure During make

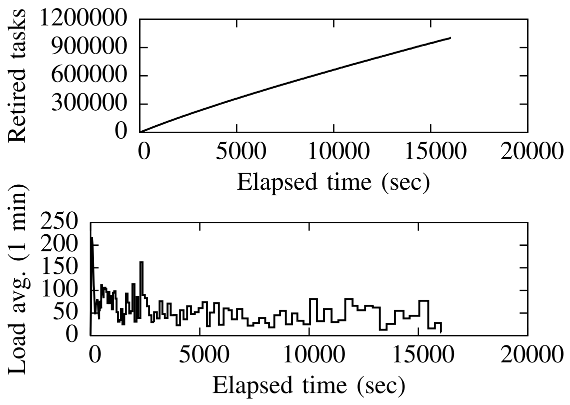

Fig. 4. Retired Tasks (top) and Load Average on the Dispatcher Node (bottom)

in the makefile, it uses $S$ in place of the `sh` above. This way we can effectively 'intercept' process invocations of GNU make. We implemented a small program called `mksh`, which simply sends the argument of `-c` option to `xmake`. Fig. 3 summarizes the process structure while running a workflow with GXP make.

We still need a trick to insert the above definition into the makefile given from the user, for which GNU make provides another useful feature. It is a customization via an environment variable `MAKEFILES`, which lists makefiles that should be read before each makefile as if they are included from it. Before invoking GNU make, `gxpc make` appends to this variable the name of the file whose only content is `SHELL=mksh`.

## VI. Performance Assessment

Scalability and throughput of GXP make are limited by resources imposed on the dispatcher and actions that must happen there. In particular, given the maximum parallelism $P$ (specified by '`-j` $P$'), the dispatcher node must accommodate GNU make, `xmake`, up to $P$ `mksh` processes, and up to $P$ `gxpc e` processes. They clearly limit the scalability at some point. Throughput is limited by how fast the dispatcher node can process events related to task submission and termination. Due to space limitation, our evaluation results presented here are brief. We will publish more thorough performance analysis and optimization elsewhere.

We measured the dispatching throughput of GXP make in the InTrigger environment [27]. It is a cluster of 16 clusters in Japan. It has approximately 400 nodes and 1,600 CPU cores in total, of which 385 nodes were used in the experiment. To emulate a high concurrency environment, GXP make is configured so it sends to each host up to six times as many tasks as its cores. For example, it sends to a dual-core host up to 12 tasks at a time. It resulted in a worker pool running up to 10,032 tasks concurrently. A node is designated as the dispatcher node (home node of GXP). It is DELL R610 server with Xeon 5560 2.8GHz and 24GB DDR3 memory. The operating system is Linux version 2.6.26-2-amd64 with completely fair scheduler (CFS).

In this environment, we ran a workflow of 1,003,200 ($\approx$ 1M) independent tasks. Each task does nothing (the exact command is '`:`'), so the purpose of the experiment is purely stressing the dispatcher. It finished in 16,050 seconds. On the top of Fig. 4 is the number of finished tasks over time. The retirement rate is very stable over time and 62 tasks per second. The number justifies the two level architecture; there are popular batch schedulers and Grid middleware that have a throughput as low as a few tasks per second [8]. It is still much slower than Falkon that is outstandingly faster than any other systems ($> 3,000$ tasks per second as of [20] and $15,558$ tasks per second in a more recent report on the web[1]). It should be noted however that Falkon is a back-end job dispatcher that should be combined with a front-end that analyzes workflows and sends tasks to it. Combined with its front-end Swift, the reported throughput is 56 tasks per second on the TeraGrid [8]. Although these numbers are easy to vary by a significant constant factor, the bottom line is that GXP make is among the fastest systems supporting task dependencies. At the bottom of Fig. 4 is the load average of the dispatcher node over time. It fluctuates and in the beginning went above 200, due to creations of many `mksh` and `gxpc e` processes. Note that they block most of their lifetimes; they use CPU time only right after they start and right before they exit. In systems shared by many users, the usual deployment scenario is the user runs the dispatcher at one of the nodes allocated by a batch scheduler.

## VII. A Real Use Case—Event Recognition from PubMed Abstracts

GXP make was applied to an event recognition workflow that extracts and classifies bio-molecular events mentioned in English texts. Example bio-molecular events of interest are an expression of a certain gene, a phosphorylation of a protein, and a regulation of certain reactions. The main component program is based on the winning algorithm in BioNLP09 shared task on event extraction citeKim09Overview. We refer the reader to [28] for details of the method used and [29], [30] for other components. The input text is approximately 50% of the whole PubMed [31] database as of April 2010 and contains 72 million sentences in total. To the best of our knowledge, this is the largest dataset to which such deep NLP is applied. The entire dataset was split into approximately 300K chunks and the structure of the workflow for each chunk is shown on the top of Fig. 5.

[1]http://dev.globus.org/wiki/Incubator/Falkon

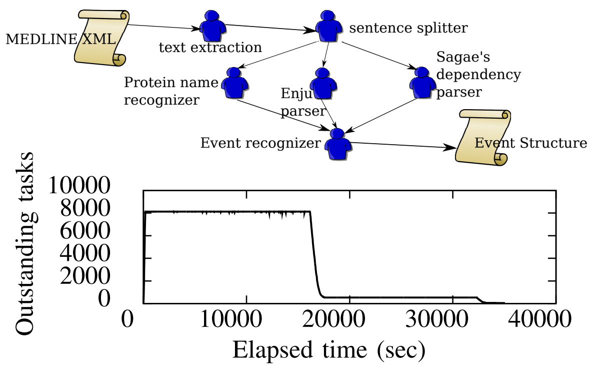

Fig. 5. Event Recognition Workflow (top) and Its Parallelism Profile (bottom)

The workflow is executed by GXP make over two clusters; one is a cluster of 952 Hitachi HA8000 servers. Each node is a four way Quad Core AMD Opteron 8356 (2.3GHz) with 32GB memory. We use 512 nodes (8192 CPU cores) of it. The other is a campus cluster of DELL R610 nodes. Each node has Xeon E5530 2.4GHz and 24GB memory. We used 52 nodes of it (416 CPU cores). At the bottom of Fig. 5 is the parallelism profile over approximately ten hours. Parallelism drops at around 18,000 sec, from about 8,000 to below 1,000. This is because HA8000 is allocated to us only in the first six hours. After those hours, the second cluster joined the computation and remaining tasks were dispatched to it, achieving the parallelism around 400.

## VIII. Conclusion and Future Work

We described a workflow system based on make and demonstrated that it is indeed a powerful and expressive system with a useful performance. We built it based on the unmodified GNU make and a remote shell (GXP), by intercepting process invocation of GNU make. Our hope is the paper provided a refreshing view about how economically a powerful, easy-to-learn/teach, and efficient workflow system can be built. Our future work includes performance investigation and improvement, file systems for distributed data intensive workflows, and workflow systems embedded in a popular scripting language. GXP is available at http://www.logos.ic.i.u-tokyo.ac.jp/gxp/.

## Acknowledgment

This work was supported in part by the MEXT Grant-in-Aid for Specially Promoted Research and Scientific Research on Priority Areas.

## References

[1] K. Taura, "GXP: An interactive shell for the grid environment," in *IWIA*, 2004, pp. 59–67.

[2] "Lustre," http://wiki.lustre.org/.

[3] "IBM general parallel file system," http://www-03.ibm.com/systems/clusters/software/gpfs/index.html.

[4] "SSH filesystem," http://fuse.sourceforge.net/sshfs.html.

[5] O. Tatebe, N. Soda, Y. Morita, S. Matsuoka, and S. Sekiguchi, "Gfarm v2: A grid file system that supports high-performance distributed and parallel data computing," in *CHEP*, 2004.

[6] N. Dun, K. Taura, and A. Yonezawa, "GMount: An ad hoc and locality-aware distributed file system by using SSH and FUSE," in *CCGRID*, 2009, pp. 188–195.

[7] D. Thain and M. Livny, "Parrot: Transparent user-level middleware for data-intensive computing," in *Workshop on Adaptive Grid Middleware*, 2003.

[8] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, I. Raicu, T. Stef-Praun, and M. Wilde, "Swift: Fast, reliable, loosely coupled parallel computation," in *IEEE International Workshop on Scientific Workflows*, 2007.

[9] M. Isard, M. Budiu, Y. Yu, A. Birrell, , and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *EuroSys*, 2007.

[10] T. Hiraishi, T. Abe, Y. Miyake, T. Iwashita, and H. Nakashima, "Xcrypt: Flexible and intuitive job-parallel script," in *SACSIS*, 2010, pp. 183–191, (in Japanese).

[11] "Hadoop," http://hadoop.apache.org/.

[12] T. Oinn, M. Greenwood, M. Addis, N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, P. Li, P. Lord, M. Pocock, M. Senger, R. Stevens, A. Wipat, and C. Wroe, "Taverna: lessons in creating a workflow environment for the life sciences," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, pp. 1067–1100, August 2006.

[13] D. Churches, G. Gombas, A. Harrison, J. Maassen, C. Robinson, M. Shields, I. Taylor, and I. Wang, "Programming scientific and distributed workflow with Triana services," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, pp. 1021–1037, August 2006.

[14] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao, "Scientific workflow management and the Kepler system," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, pp. 1039–1065, August 2006.

[15] M. Tanaka and O. Tatebe, "Pwrake: A parallel and distributed flexible workflow management tool for wide-area data intensive computing (poster)," in *HPDC*, 2010.

[16] L. Yi, C. Moretti, S. Emrich, J. Kenneth, and D. Thain, "Harnessing parallelism in multicore clusters with the all-pairs and wavefront abstractions," in *HPDC*, 2009, pp. 1–10.

[17] "Grid Engine project home," http://gridengine.sunsource.net/.

[18] P. Couvares, T. Kosar, A. Roy, J. Weber, and K. Wenger, *Workflows for e-Science: Scientific Workflows for Grids*. Springer Press, 2007, ch. Workflow in Condor.

[19] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz., "Pegasus: a framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming Journal*, vol. 13, no. 3, pp. 219–237, 2005.

[20] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde, "Falkon: a Fast and Light-weight tasK executiON framework," in *SC*, 2007.

[21] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *OSDI*, 2004, pp. 137–150.

[22] B. Folliot, "Implementation of a parallel and distributed MAKE on NFS with GATOS," in *International Phoenix Conference on Computers and Communications*, 1990, p. 871.

[23] J. Buffenbarger, "A large-scale fault-tolerant distributed software-build process," in *British Computer Society Configuration Management Specialist Group Conference*, 2005.

[24] A. Lih and E. Zadok, "PGMAKE: A portable distributed make system," Computer Science Department, Columbia University, Tech. Rep., 1994.

[25] "GXP Grid and Cluster Shell," http://www.logos.ic.i.u-tokyo.ac.jp/gxp/.

[26] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: the Condor experience." *Concurrency - Practice and Experience*, vol. 17, no. 2-4, pp. 323–356, 2005.

[27] "InTrigger platform," http://www.intrigger.jp/.

[28] M. Miwa, R. Sætre, J.-D. Kim, and J. Tsujii, "Event extraction with complex event classification using rich features," *JBCB*, vol. 8, no. 1, pp. 131–146, Feb. 2010.

[29] T. Ninomiya, T. Matsuzaki, Y. Miyao, and J. Tsujii, "A log-linear model with an n-gram reference distribution for accurate HPSG parsing," in *IWPT*, 2007.

[30] K. Sagae and J. Tsujii, "Dependency parsing and domain adaptation with LR models and parser ensembles," in *The CoNLL 2007 Shared Task (EMNLP-CoNLL'07)*, 2007.

[31] "Pubmed," http://www.ncbi.nlm.nih.gov/pubmed.