Earth Sci Inform (2009) 2:141–155
DOI 10.1007/s12145-009-0021-z

RESEARCH ARTICLE

Efficient clustered server-side data analysis workflows using SWAMP

Daniel L. Wang · Charles S. Zender · Stephen F. Jenks

Received: 25 July 2008 / Accepted: 23 February 2009 / Published online: 17 March 2009

Abstract Technology continues to enable scientists to set new records in data collection and production, intensifying a need for large scale tools to efficiently process and analyze the growing mountain of data. To complement growth in the number of data centers and the volume of data they store, we introduce our Script Workflow Analysis for MultiProcessing (SWAMP) system. Our system provides safe server-side processing capabilities that allow scientists to reuse familiar desktop-based analysis methods represented in shell-scripts. Built-in script compilation isolates file accesses and generates workflows, while a cluster-capable execution engine partitions and executes the resulting workflow. Benchmarks illustrate up to 20X performance gains, as well as the importance of I/O considerations which make other computation systems less effective at geoscience data reduction.

Communicated by: H.A. Babaie

This work was supported by the National Science Foundation under grant IIS-0431203.

D. L. Wang (✉)
SLAC National Accelerator Laboratory,
2575 Sand Hill Road, M/S 97, Menlo Park, CA, USA
e-mail: danielw@slac.stanford.edu

C. S. Zender
Dept. of Earth System Science,
University of California, Irvine, Irvine, CA, USA

D. L. Wang · S. F. Jenks
Dept. of Elec. Engn. and Comp. Sci.,
University of California, Irvine, Irvine, CA, USA

Keywords Data management · Geoscience · Parallel computing · Script compilation

Introduction

While new technologies have spurred a glut in produced data, geoscience data utilization remains hampered by the physical separation between producer and user–data transfer and analysis is always too slow, too expensive, or both. Suppose a researcher is interested in a particular property X, which can be easily computed over her terabyte dataset in a few minutes. Transfer costs make computing over an overseas colleague's own terabyte dataset already cumbersome, and computing over a consortium's hundred terabyte collection practically impossible. In this paper, we explore a method to make the former easy and the latter possible. Our system provides a method of safely and efficiently executing script-defined analyses on remote data.

By making large scale geoscience data analysis and reduction accessible to scientists with only modest network and local computational capacity, our approach will enable geoscientists to evaluate more and larger data sets, enhancing their research ability. Our approach does not require such scientists to learn new, complicated toolsets, but rather enhances script-based data reduction and analysis in common use today.

The rest of this paper will be organized as follows: section "Problem" describes the problem, section "Approach" describes our approach compared to existing systems, section "Design" describes the design of

our system, section "Performance-oriented architecture" discusses performance issues, and section "Experiment" explains our experimental setup and results. We conclude and discuss future plans in section "Conclusions and further study".

Problem

The growing gap between the amount of data produced and the amount analyzed is a significant problem. Computing advances have made it possible for individual scientists to run their own simulations with their own parameters and data at wider scales and finer resolutions, but data analysis remains hampered by the difficulty of moving bulky data and the scarcity of software tools that scale to large volumes. Observed data from local and remote sensing has grown in volume similarly, while presenting additional challenges such as new sensing types, environments, and scales.

Most geoscience analysis and visualization tools, for example, are designed to operate with single variables or two-dimensional plots, but struggle when performing larger, bulk data processing. With individual scientists not uncommonly producing nearly 100 gigabytes in single runs, handling large data volumes has become increasingly important.

Large data volume

Although network bandwidth records are broken regularly, dataset sizes have also grown, keeping data movement a challenge. Because managing high volumes is so difficult, scientists typically avoid the problem, executing high-resolution simulations infrequently, either as followups to exhaustive lower-resolution testing or as team efforts in larger collaborative experiments. Analysis is done tediously, often using custom Fortran code to analyze or to reduce the data to scales manageable by other tools. For example, 100 gigabytes of data can be generated in a few days of atmospheric simulation, but such runs are infrequent due to the difficulty in managing the results. Because each study is unique, each corresponding body of processing code is frequently unique and written with performance optimization as a low priority. As a result, scientists' work is hampered by an implicit directive to keep data sizes small enough for their workstations to handle. Geoscience analysis and visualization tools often assume that the entire dataset can be kept in random-access memory, aiming for interactive usage rather than volume processing.

Even when large data volumes are produced and centralized for larger collaborative studies such as those undertaken by the Intergovernmental Panel on Climate Change (IPCC), usage is limited by the large effort required for users to download and manage the data. Working around this problem requires extreme measures—the Sloan Digital Sky Survey (Szalay et al. 2002) distributed their 1.3 terabyte compressed dataset by shipping complete computers. While the Internet2 Network and National LambdaRail do provide exceptionally large long-haul bandwidth, their connectivity is limited to selected research and education centers. In recognition of the problem of large scale data analysis, F-TDS (Schweitzer et al. 2008) and GDS (Adams 2008), two geoscience data server projects, are implementing rich server-side data analysis capabilities in response to an already common user sentiment that plain, subsetted, and aggregated data access are not enough.

High data-intensity

Recent work on high-performance computing research has recognized the importance of data-intensive workloads (Xue et al. 2008), but serious study remains rare regarding high data-intensive workloads where computation is small relative to data size. *High* data-intensive workloads can be defined as having very low (between 0 and 10) floating point operations per byte of input data (flop/byte). For example, computing an average, a common task in geoscience, requires less than one flop/byte. In these workloads, computational costs are small relative to data movement costs due to disk or network I/O. Whereas traditional compute-bound execution performance is at most *influenced* by I/O considerations, performance is nearly *defined* by I/O considerations in these high data-intensive cases.

Approach

Our solution avoids the data movement problem by enabling scientists to use existing desktop-oriented analysis tools on remote data centers, and optimizes execution with techniques sensitive to data movement. This strategy moves data analysis and reduction operations to the remote data center where the data reside, taking advantage of the computational, networking, and storage infrastructure of the data center to perform the I/O-centric data-intensive operations quickly and efficiently. By exploiting locality and parallelism, our system dramatically improves overall performance and significantly reduces data transfer sizes, both of which benefit both end users and the data centers.

File granularity

Many desktop analysis tools such as MATLAB (Hanselman and Littlefield 2004) and GrADS (Doty and Kinter 1995) are able to access and manipulate data in terms of individual values at grid points. Complementary to these are those tools that work at larger granularities, providing insights into trends and statistics in the data that help scientists decide when to apply the fine-grained tools. The files exchanged in geosciences represent logical datasets, and are frequently broken apart to ease storage management or to cope with technological limitations. Data operations of significant intensity are well-matched to files or sets of files. Granularities smaller than files are easily hyperslabbed from files. Files, therefore, are a logical level of granularity for a system designed for bulk processing.

In some cases, files are aggregated into logical datasets that are published by data servers such as OPeNDAP (Cornillon 2003) or THREDDS (Domenico et al. 2002). These help free scientists from managing inconsistencies in file splitting used by different colleagues. Fortunately, these aggregations are easily mapped to files and are thus complementary to file-based analysis.

Implicitly-defined workflows

Workflows are differentiated from other batch workloads through the interdependency of their constituent parts. Because they contain internal dependencies, they may perform poorly on generic grid frameworks designed for large volumes of independent tasks. Workflows can be executed on grid workflow engines, which can utilize the dependency information to provide maximum parallel performance over grid resources. However, careful work is required to design and specify each workflow with each particular grid framework. With a detailed specification, the workflow scheduler can then provide an optimal schedule. Deelman et al. (2005) describes a system that leverages job performance characteristics to produce efficient execution plans.

Unfortunately, workflow specifications are too complex for individual scientists to construct. While they rarely possess grid workflow hardware, they commonly perform workflow processing, that is, tasks composed of interdependent components, in the form of custom scripts that they execute on whatever data they are interested in. Our approach directly leverages their scripts to construct implicit workflows. Once converted, scripts can benefit from workflow optimization, enabling additional parallelism beyond application-level multithreading.

Shell interface

Instead of choosing a data-focused domain language such as one used in GrADS (Doty and Kinter 1995) or FERRET (Hankin et al. 1996), we chose shell language (used by standard shell command interpreters). Shell language is used to define an ordered sequence of command invocations and operates naturally at the file-level granularity mentioned above.

The choice of shell language offers three key advantages over the use of workflow languages: universal familiarity, applicability, and scalability. Familiarity comes naturally, since nearly every user who processes data of significant volume regularly utilizes shell scripts of some form to automate processing. The remaining users who do not are familiar with executing programs at command-lines. In leveraging these existing scripts, our approach minimizes retraining, a common and significant barrier of adoption in many computing frameworks.

Shell language is applicable for a wide variety of tasks, due to its ability as a glue language, which *connects programs* written in other languages. Though uniqueness in scientists' usage is seen in custom Fortran (or other language) code and in personal scripted sequences, internet-scale collaboration has resulted in common data formats and common toolsets for manipulation. By supporting the netCDF Operators (NCO) (Zender 2008), a popular tool set that supports processing the most common format for exchanging geoscience data, netCDF (Rew and Davis 1990), our approach covers a wide variety of geoscience data analysis and reduction tasks. Specifically, our decision to support the Bourne shell (`sh`) syntax and its most common features allows reuse of much of the existing body of shell scripts *without modification*.

The choice of a shell language interface, specifically one whose syntax mimics `sh`, distinguishes our approach from other distributed dataflow systems such as DataCutter (Beynon et al. 2001) or comprehensive data grid systems like Chimera (Foster et al. 2002). A system shell language like `sh` was never intended to describe workflows, and lacks syntax to define complex workflow topologies and configuration. Workflows automatically created from these scripts are inherently disadvantaged in attaining peak performance. On the other hand, existing analysis tools like NCO can be used without modification, and existing shell scripts can be used (almost) without modification—we believe this tradeoff in favor of usability is sufficiently compelling. It explores how high an individual's existing workstation scripts can be scaled, instead of providing interfaces for maximum

performance on petascale computations created by collaborations.

Scalability is achieved primarily by parallelization and data locality optimization. Script-detected parallelization allows performance to scale with parallel hardware. This provides an additional level of parallelization beyond the programs' own use of parallel techniques such as OpenMP (Dagum and Menon 1998) and MPI (Gropp et al. 1999). Should the programs be optimized further for performance (Zender and Mangalam 2007), a script based system benefits automatically. Thus techniques for parallel I/O such as MPI I/O or Parallel-netCDF (Li et al. 2003), specifically for NCO's netCDF layer, would have cumulative, not conflicting benefits. Combining both script-level and application-level parallelism better positions such a system to exploit increasingly parallel hardware.

Grid-inspired

Grid technologies represent the latest promise in high-performance computing. By flattening access to disparate, homogeneous resources, grid technologies should yield an abundance of raw computational power that should be irresistible to anyone desiring high performance computing. While grid projects such as Cui et al. (2007), Natrajan et al. (2004), Tejedor and Badia (2008), and Rubio-Solar et al. (2008) each enjoy success in their own goals, no single implementation has dominated. Though all aim for generality, all have features that are tuned to their own application domains, and are not generic enough for each other's purposes. All target large-scale computation, but only one maintains an interface simple enough for lightweight tasks from casual users (Tejedor and Badia 2008). The few that are aware of data locality require specialized tools to construct tasks.

In some ways, grid technologies are unsuitable to computation that is largely data-bound instead of compute-bound. Indeed, an underlying assumption is that the availability of a larger computational pool will enhance performance, an assumption that does not hold when a task's performance is dependent on disk transfer rates rather than CPU speed. These data-bound tasks are generally under-resourced, given that high performance computing prizes teraflops rather than secondary storage (disk) bandwidth. Lack of attention to data-boundedness has led to low popularity of data-bound tasks. This low popularity has led to a lack of developer interest in data-bound issues, leading to the development of programs that ignore data intensity. Because programs which address data intensity are so rare, there are few studies of data-bound performance to be used by prospective researchers, leading to the original problem, a lack of attention to data-bound tasks. Some computational grid models consider data movement costs and schedule data movement coupled with computation (Xue et al. 2008), but still assume that computation cost dominates.

Our solution borrows the grid concept of distributing work among loosely coupled machines, but differs from a grid computation service in two ways. First, it provides a computational service necessarily bound to data locality, with an interface similar to grid computational services, but tailored for data rather than computational performance. Second, its execution model groups execution to minimize data movement. While scaling to leverage computer cluster resources, it discards the flexible resource mapping because its workload does not benefit from the availability of remote computational resources. Its tasks can usually be completed in less time than the time to stage the inputs to a large, fast, but remote compute cluster.

Usage model

Because terascale data remains too cumbersome to download, a practical solution must reduce or eliminate the initial data download. This suggests that the computation itself should be relocated to where the

Fig. 1 A sample script

```
for mdl in ccma_cgcm3_1 ncar_ccsm3_0; do
    ncwa -a lat,lon sresa1b_$mdl.nc avg_$mdl.nc
    ncwa -d time,0,11 avg_$mdl.nc 2000_$mdl.nc
    ncdiff  avg_$mdl.nc 2000_$mdl.nc anm_$mdl.nc
done
# Create model ensemble
ncea avg_*.nc sresa1b_avg.nc
# Ensemble mean of year 2000
ncwa -d time,0,11 sresa1b_avg.nc sresa1b_avg_2k.nc
# Create ensemble anomaly
ncdiff sresa1b_avg.nc sresa1b_avg_2k.nc sresa1b_anm.nc
```

data itself resides. Therefore, our solution implements a computational service to complement existing data access service. By re-interpreting unmodified desktop analysis shell scripts, scientists' own custom analyses can be safely relocated to the data server, without requiring user accounts, familiarity with foreign environments, or parallel or advanced computing skill.

The sample script in Fig. 1 computes the time-varying spatial mean and deviation from the year 2000 for two different climate models, along with the ensemble average and its deviation. It executes unmodified on a desktop as well as on our remote computing service. Because the system re-interprets the script, it can ensure that only safe programs are referenced and only permissible files are read, and that program outputs are safely mapped to isolated areas. It is worth noting that (Tejedor and Badia 2008) provides similarly simple computational service, but requires tasks to be specified in a custom Perl-like language (Wall et al. 2000).

Design

We designed our system, which we call SWAMP (Script Workflow Analysis for MultiProcessing), as a computational service designed to be deployed alongside a data service. SWAMP was initially implemented as a plugin to a leading geoscience data server, OPeNDAP, but outgrew the protocol and operating limitations in OPeNDAP Server 3. Basic SWAMP operation is illustrated in the timeline diagram in Fig. 2. It outlines how the SWAMP client program interacts with a frontend instance and how the frontend interacts with worker instances. The client communicates using a SOAP API, which should ease implementation of alternate clients. Users pass a script filename to the client program, which simply passes the script contents, unmodified, to a frontend instance, and polls the frontend for task status. When the task completes, the client downloads the task outputs over HTTP.

The frontend instance accepts scripts, compiles them into workflows, partitions the results into logical clusters, and manages cluster execution. Work is dispatched to workers at cluster-granularity, but workers report completion at command-granularity, enabling a dependent cluster to be scheduled before the entirety of its parent clusters complete. We describe execution in further detail in the next section "Performance-oriented architecture".

Lightweight computation

SWAMP is distinguished from existing grid frameworks and data servers in how and what sort of computational service it provides. By providing a scripting interface, it offers analytical function beyond simple subsetting that is provided by existing data servers. While its simple, lightweight computational capability may seem unjustified in an era of commodity high-performance workstations, its capabilities are matched towards providing access to scientific results from input data that is too large to download. Its parallelization capability enhances scalability, and its sandboxed and lightweight execution make it more feasible for open-access environments than generic grid computation systems. Scripts like the one illustrated in Fig. 1 can be submitted to a SWAMP instance using a simple SOAP (Box et al. 2000) or XML-RPC (Winer 1999) call, which returns an identifying token to be used to check job status and download results.

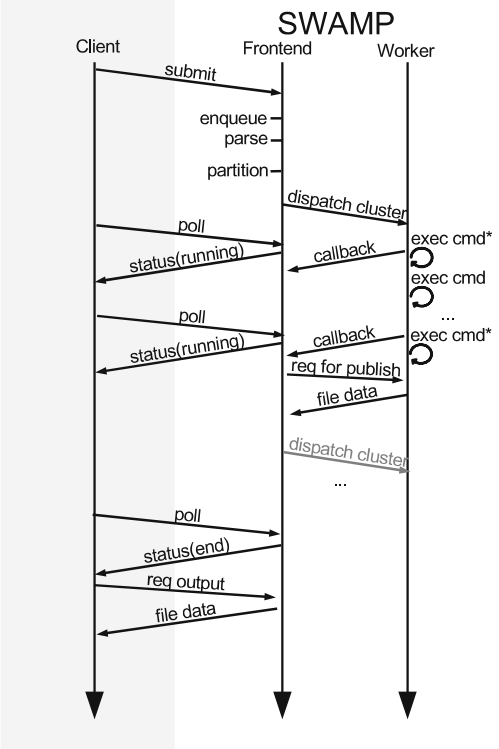

Fig. 2 Simplified timeline for a SWAMP task

Shell language

Computation in SWAMP is specified with the same syntax as ordinary command interpreters, shell language, specifically the Bourne shell syntax (Bourne 1978). Direct, unmodified re-use of existing user scripts is therefore possible in SWAMP. While other parallel systems require users to explicitly annotate their code to exploit parallelism, SWAMP reveals how much can be gained in situations where the user does not. It intentionally

provides the analysis subset of a desktop environment so that remote data can be processed just as simply. Debugging a SWAMP script, for example, can usually be done outside SWAMP using the operating system's own shell.

SWAMP supports most syntactical features used by scientists in analysis scripts. Variables can be defined and referenced. Control structures such as for-loops and conditional branches can be used, easing volume processing and portability. Safe shell "helper" programs such as *seq* (from the GNU `coreutils` package) and *printf* can be used, as can the filename wildcards `?` and `*`. Wildcards are intelligently expanded using the script's local context, properly accounting for files written earlier. Parameter variables, i.e. those referenced by `$1` or `$2`, will be supported in a later revision. For security and implementation tractability, program selection is restricted to a set of programs whose argument and file input/output semantics are programmed into the system. Without semantic understanding, the system would not be able to detect inter-command dependencies or to sandbox script execution. In its current implementation the program set is composed of the NCO tools, which are regularly used in the geoscience domain.

The use of shell syntax allows the system broader applicability than geosciences. The system can operate in any domain where scripts are used by adding support for its programs, given that the programs are *well-behaved*, i.e. their inputs and outputs are files, and can be determined solely by argument parsing. Once the system is in place, parallel execution of scripts becomes a question of "Why not?" rather than "Why?" As an example, UNIX *grep* could be parallelized in such a fashion without significant effort, once written in a script as in Fig. 3.

Locality sensitivity

Without a locality-sensitive design, a system cannot provide computational service to large datasets, because large data sizes make input pre-staging and output post-staging too expensive to be practical. An initial design goal of our system is to enable user analyses to scale to large data volumes. Because long-haul network bandwidth will always fall short with respect to data volumes produced from observation or simulation, any solution for providing computation on such data must not *move* data unless absolutely necessary. Such is the primary motivation for providing computational service at a data repository. Additionally, SWAMP's execution engine dispatches processes with locality sensitivity in two ways. First, it partitions workflows into clusters with reduced inter-cluster dependencies. This minimizes communication between worker nodes. Secondly, clusters are preferentially dispatched to nodes to minimize the need for input file staging. Without these methods, data-intensive performance scales poorly.

Fig. 3 A parallelizable script that searches for a parametrized string

```
for f in *; do
  grep $1 $f
done
```

Administration

SWAMP has a number of features that ease its deployment both at data centers and lab group installations. The server is implemented in the Python language as a self-contained, standalone daemon accessible over SOAP or XML-RPC protocols. It can operate in clustered configurations and is able to add and remove worker instances without restarting. Each installation is configured by editing one file or two files in the case of clustered instances. For enhanced I/O performance, it can exploit ramdisks on machines with abundant physical memory. Built-in logging facilities can be used for troubleshooting or usage tracking. It is dependent only on a few widely available Python libraries, such as Twisted (Fettig 2006), and NCO and operates without an existing web server, making it simple to install and setup. SWAMP may increase load on the storage subsystem, due to the additional interface it provides for accessing and computing over datasets, but the increase in storage traffic is accompanied by a generally far more drastic decrease in external network bandwidth for the variety of analysis tasks supported.

Security

Because our system is designed to provide computation access as widely as plain data access, security issues must be considered. SWAMP resists overall system intrusion by using an interpreted language that resists buffer overflows, and by restricting scripts to accessing a small set of well-understood programs. Program arguments must pass basic validation tests, and invocations occur directly without implicit subshell use, making it difficult to invoke untrusted programs. Filenames in scripts are remapped first to logical filenames in static single assignment form (Alpern et al. 1988), a feature which also enhances execution parallelism. Virtualizing filenames in such a manner not only isolates tasks from each other, but also from accessing system files. Scripts are also executed one at a time, which allows a long-running script to monopolize resources and potentially

deny service, but prevents a flood of requests from overloading and crashing the system.

Dynamic scheduling

Our system schedules tasks on-demand as processing resources are available, whereas other workflow frameworks schedule tasks statically before beginning execution. This choice reflects the practical realities of accepting script-specified user workflows. Static scheduling can create better workflow schedules whose end-to-end execution times, i.e. makespans, are closer to the minimum times, but relies heavily on accurate performance prediction. In operational environments, where workflows are used as part of a well-defined processing pipeline, tasks have approximately deterministic execution times, and their performance can thus be predicted for future iterations. However, while each scientist may focus on a limited number of scripts, the number of scripts used by the larger community is large. Therefore, the execution times of the contained commands cannot be predicted without support in each command for some sort of "dry run" mode where its predicted performance is computed (Zender and Mangalam 2007), along with sufficient metadata to allow dependent programs to predict performance. Such a feature is difficult to implement, and may not be possible for all prospective programs. With this in mind, the more reactive approach, dynamic scheduling, is more appropriate.

The current implementation dispatches script commands for execution on nodes according to each node's configurable maximum parallel processes per node (PPN). Maximum performance is normally achieved when the number of parallel processes equals the number of physical processors, with an important exception described in section "I/O dependence". Another attractive possibility, given the system's target of data-bottlenecked scripts, is to schedule according to I/O load. By limiting parallel disk access, seek frequency can be reduced to bring disk bandwidth higher to its theoretical maximum sustained read rate.

Performance-oriented architecture

By providing computation at the data source, our system avoids the primary problem in using large scale remote data: the lengthy data download time. Figure 4 illustrates the contribution of data transfer to overall task execution time for our two benchmarks (see section "Setup"). In both cases it is clear that transfer cost is just as important, if not more important than computation cost. Additional performance benefits come from the following areas.

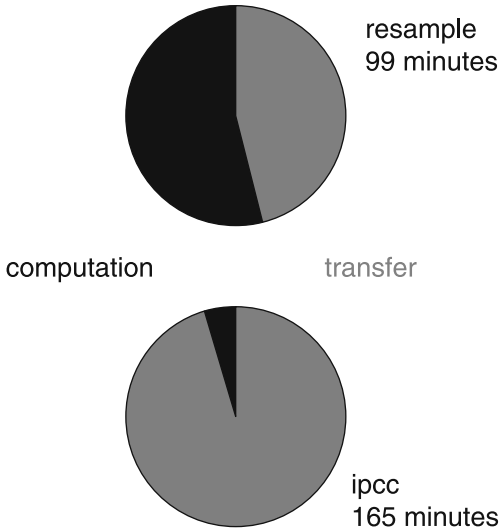

Fig. 4 Components of total wall-clock time for the benchmarks

Compilation

By parsing command semantics, our system is able to transform an otherwise serial script into a directed-acyclic-graph workflow. Each command-line becomes a graph node, and its parent and child nodes are determined by matching input and output filenames. The parser derives these filenames through code that is customized for the supported executables, the netCDF Operators. Because it is intractable to determine argument semantics for *all arbitrary* programs automatically, support is restricted to programs common in a domain, and thus make the required custom code more tractable. NCO tools are easily combined in scripts to form useful geoscience data reduction and analysis tasks, and support for their semantics alone yields a rich computational capability.

By treating filenames as variable names and commands as basic operators, more traditional compiler techniques can be applied. Live variable analysis allows the system to intelligently guess which files are results and which are intermediate "temporary" files. "Dead" files can be allocated to memory instead of disk, reducing disk contention during execution and bandwidth when returning results to clients.

Partitioning

Multi-computer performance in earlier implementations suffered from high synchronization overhead between master and worker nodes. Commands were

dispatched individually, promoting an even balance of commands on available worker nodes, but limiting scalability due to the frequency of updating the master node with command progress. To reduce this problem, the workflow is partitioned into groups of nodes, while minimizing the dependency between groups. By dispatching work in these partitions, multi-node execution avoids unnecessary data transfer and required master-worker communication overhead, at the risk of work imbalance from the coarser scheduling granularity.

The partitioning algorithm is as follows. Consider a directed graph $G = (V, E)$ with the set of vertices V (representing script commands) and the set of directed edges E (representing derived dependencies between the commands). Let $x \prec y$ signify that a vertex x is a direct predecessor (parent) of a vertex y.

Define a *root* of a set of vertices S (S is a subset of G), to be a vertex r that has no predecessors in S, i.e. $r \in S, \emptyset = \{v : v \in S, v \prec r\}$.

- Compute roots of G. Let n_r be the number of roots.
- For each root r_i, compute its corresponding root cluster C_{r_i}, the set including the root and all of its direct and indirect successors.

$$C_{r_i} = r_i \cup \{v : v \in V, r_i \prec v\} \tag{1}$$

- For each root cluster C_{r_i}, compute its intersections with other root clusters, resulting in intersection clusters $C_{r_{i,j}}$.

$$C_{r_{i,j}} = C_{r_i} \cap C_{r_j} \tag{2}$$

(Note that for Eq. 2, computing $C_{r_{i,j}}$ for $i, j \in [0, n_r], i < j$ is sufficient since $C_{r_{i,j}} = C_{r_{j,i}}$ and we are not interested in $C_{r_{i,i}} = C_{r_i}$.)

- Construct C'_{r_i} by removing descendants shared with other root clusters. These modified root clusters can be dispatched independently, in parallel.

$$C'_{r_i} = \{v : v \in C_{r_i} \quad v \notin C_{r_{i,j}} \forall j\} \tag{3}$$

- Construct $C'_{r_{i,j}}$, by removing elements shared with other intersection clusters. These modified intersection clusters can be dispatched independently of other intersection clusters, but may have dependencies on root clusters.

$$C'_{r_{i,j}} = \left\{ \begin{array}{l} v : v \in C_{r_{i,j}} \\ \quad v \notin C_{r_{x,y}} \forall x, y : (x < i) or (x = i,\ y < j) \end{array} \right\} \tag{4}$$

This provides $n_r + |C'_{r_{i,j}}|$ clusters, with n_r initial parallel clusters to execute, and up to $n_r \times (n_r - 1)$ non-root independent clusters. If additional parallelism is needed, $C'_{r_{i,j}}$ can be recursively split according to the original algorithm, since they may have multiple roots. Otherwise if there is only one root, the cluster may be split by cutting the cluster after the first (smallest depth) command that has multiple successors. The algorithm avoids cutting between a parent node and its only-child, because that confers no parallelization benefit and risks penalties from file staging delays if the child were to be scheduled outside its parent's node.

This algorithm has performed adequately in our benchmarks, although standard bi-partitioning or k-partitioning algorithms were considered. These alternate partitioning algorithms operate on flow networks and are common in the computer-aided design (CAD) community for integrated circuit layout purposes. Though attractive in algorithmic complexity, their minimization constraints were found to be unsuitable. They are designed to balance nodes between partitions for physical layout, while our algorithm focuses solely on branching and "fan-out" points.

Execution

Attaining maximum performance for compute-intensive workloads is a well-researched topic in high-performance computing. Our system was designed to reduce or eliminate the prohibitive transfer time cost for scientific data analysis, and since our workloads are correspondingly I/O rather than compute-intensive, maximizing their performance involves different issues.

We define I/O intensive workloads as those with relatively low floating point operation (flop) counts per input byte. For these workloads, parallel performance is limited by data transfer between worker nodes in a cluster, disk contention, and overall disk bandwidth. Management overhead (compilation, scheduling, and dispatch) is an additional bottleneck, but is only important after I/O optimization.

Figure 5 illustrates our simple execution model. Scripts enter the system through a web service RPC interface, and are queued for execution. If no other script is in progress, the script is prepared for execution by parsing, validation, verification, workflow generation, and cluster partitioning. The resultant workflow clusters are then dispatched to the available workers in round-robin fashion, until the list of ready clusters is exhausted or each worker has a number of dispatched clusters equal to its number of processes per node (PPN). Each worker then executes ready commands from its set of clusters, favoring most-recently-ready commands whose inputs have just become available. This most-recently-ready policy exploits cache hotness

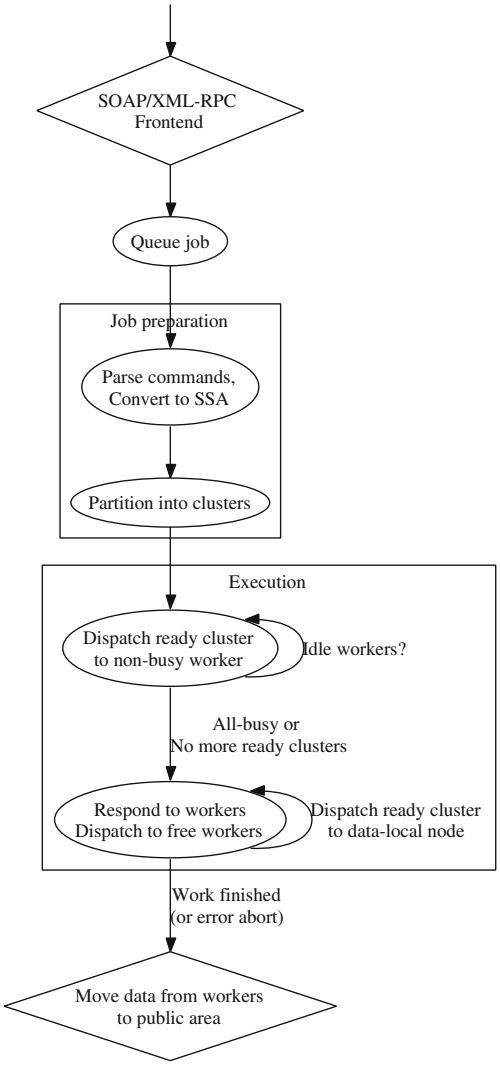

Fig. 5 Simplified flowchart

for performance and shortens temporary file lifetimes so that space in the in-memory filesystem can be reclaimed earlier. As workers complete commands, they report results to the master, except for commands without children outside the cluster–their results are batched and reported when their parent cluster completes. This trades off some accuracy in workflow state by only allowing critical messages that affect global execution scheduling (by causing command clusters to become ready) to be reported synchronously. Batching and deferring less important events drastically reduces master-worker synchronization overhead.

Disk parallelism

In Wang et al. (2007) we showed the sensitivity of data-intensive workloads on disk performance. In those cases, we found that simply parallelizing execution did not improve performance—rather performance was penalized due to the increasing disk seek rate.

Two techniques have been implemented to improve I/O performance in our system. The first, introduced in Wang et al. (2007), improves performance by converting a significant fraction (in some cases, the majority) of disk accesses to memory accesses. While single program executions in isolation are difficult to optimize—input data must be fetched and output data must be written—workflows contain much more promise. Internal edges between nodes in the workflow graph (program invocations) denote data that are produced and consumed during workflow execution. While a general-purpose operating system would certainly perform file buffering, our system explicitly avoids disk writing in those cases by utilizing a large in-memory filesystem. Our execution engine remaps command output files to memory while free space is available, spilling to disk when memory is almost full. Because in-memory filesystem space is so precious, intermediate files there are deleted as soon as they are dead (unneeded for future commands).

The second technique for improving I/O performance is enabled through multi-computer execution and replication. While many cluster installations utilize centralized head-node storage or storage-area networks (SANs), our results show that maintaining input data replicas on cluster nodes yields far greater performance, with aggregate read performance closer to the total available disk bandwidth. By splitting execution among multiple computers, the system can also write output data using the aggregate disk write bandwidth available, rather than contending for write bandwidth on centralized storage.

Despite this advantage, practical installations may find difficulty in providing cluster nodes with private data replicas. In those cases, we believe that aggressive opportunistic caching of data on node-local storage can provide good performance by minimizing concurrent access to a centralized storage system.

Disk reorganization may also improve performance. While magnetic disks have historically been treated as random access devices compared to sequential devices such as tape media, processor clock frequencies have increased and memory latencies have dropped, increasing the relative latency for random access and making disks seem more like sequential media. Hsu has proposed a system (Hsu et al. 2005) for reorganizing disk data layout to bring average random read performance closer to sequential read performance. We expect implementation of such a system would also

help our workloads, although performance would still be limited to disk bandwidth.

Experiment

We began development with the hypothesis that there would be significant process-level parallelism inherent in scientific scripts, but without a quantitative estimate. Intuitively, we predicted that eliminating input data download would make the greatest difference, but we expected that exploiting parallelism would have some impact as well.

Setup

We tested our system using two benchmark scripts in both standalone and clustered configurations. The first subsamples high temporal-resolution global wind predictions to obtain the predictions closest to twice-daily satellite overpasses (Capps and Zender 2008), while the second normalizes surface air temperature predictions from sixteen different models for intercomparison. Results from the first are pending publishing, but both are used for climate research.

The first benchmark *resample*, is conceptually shown in Fig. 6. Its 2MB script of 14,640 unique command-lines represents a practical upper-bound on the length of script expected and can be considered a "stress-test." The second benchmark *ipcc*, shown in Fig. 7 illustrates a task that was one of the motivating problems for this work. The task generates global temperature anomaly trends from 19 different simulations from the World Climate Research Programme's (WCRP's) Coupled Model Intercomparison Project phase 3 (CMIP3) multi-model dataset and compares each trend versus the mean trend of the entire data ensemble. Requiring approximately 30GB of input data this workflow is computationally simple, and results in roughly 500kB of output data. *ipcc* output data can be processed further to create a temperature trend chart similar to that shown in Fig. 8, which differs in showing the temperature trends for California rather than the entire earth.

The benchmarks were tested on a cluster of dual Opteron 270s with 16 GB of memory and dual 500 GB SATA drives, running CentOS 4.3 Linux. We tested execution with one master and one, two, or three worker worker nodes, all configured identically. We varied the number of parallel processes per node (PPN) allowed on each node in order to study the benefit of using more cores (four available per-node) versus the increased I/O contention. Multi-node performance is compared

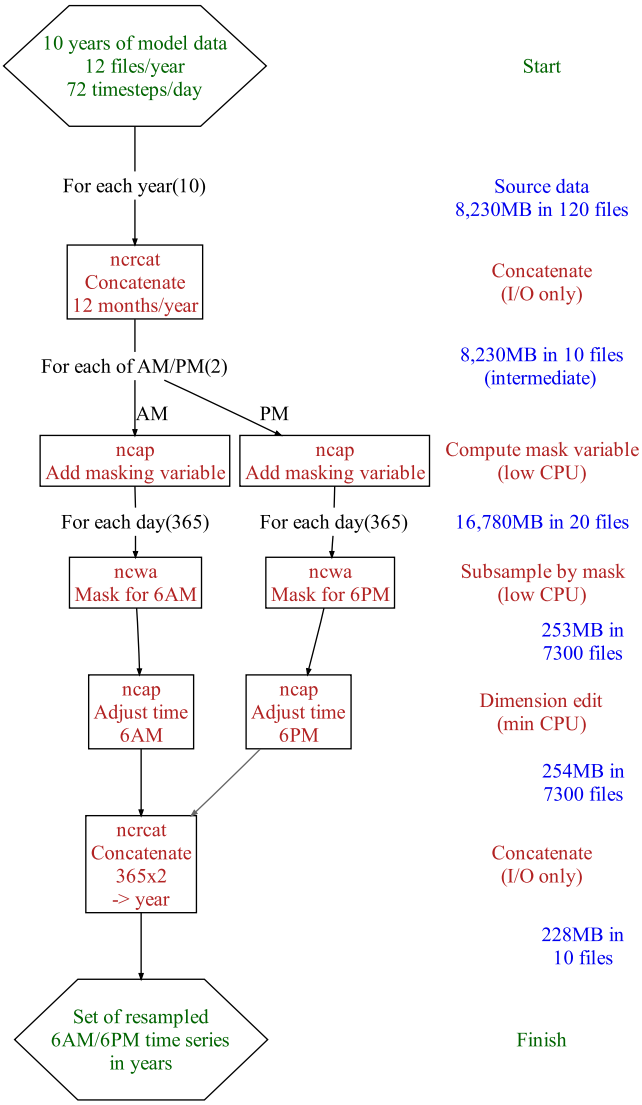

Fig. 6 Benchmark 1: *resample*—data resample

in configurations with and without cluster partitioning in order to better understand scheduling overhead.

Performance is compared with a control setup where the same script is executed at the GNU bash command line. We computed the baseline wall-clock script-execution time including the estimated time to download the input data set. Transfer times are estimated assuming 3MiBytes/s $(3 * 2^{20})$ bandwidth, based on NPAD *pathdiag* (Mathis et al. 2003) measurement of 30Mbits/s bandwidth between our workstation at UCI and the National Center for Atmospheric Research(NCAR).

These benchmarks are highly parallelizable, but are not "embarrassingly parallel" in the classical sense. Many of the individual operations, e.g. simple averaging, could be considered "embarrassingly parallel,", but

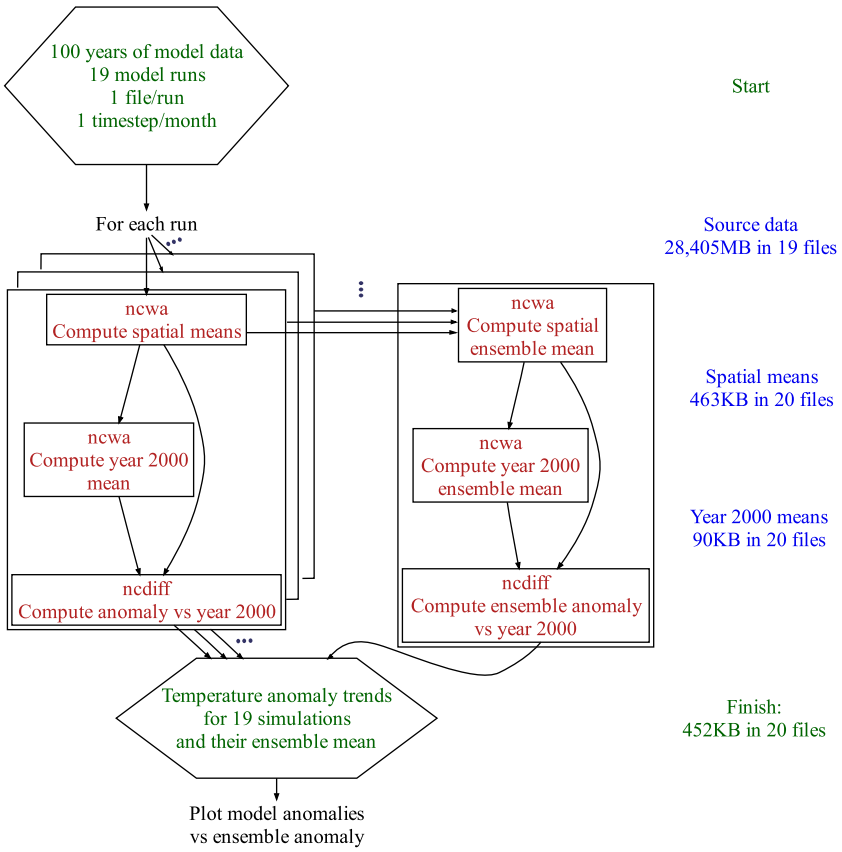

Fig. 7 Benchmark 2: *ipcc*—data intercomparison

parallelization at that fine granularity is independent to our method (and can have debatable benefit due to movement costs). Our contribution to parallelism does not consider the semantics of each executable except for its file I/O, and parallelizes execution similarly as machine instructions can be parallelized on superscalar CPUs.

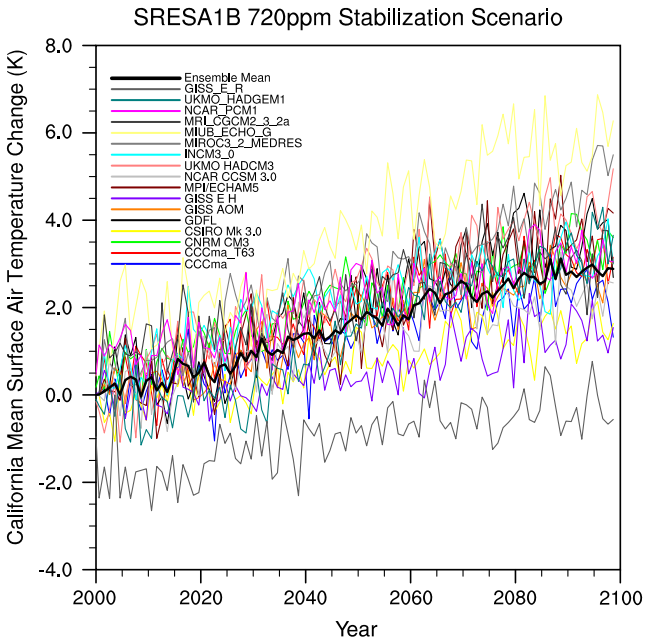

Fig. 8 Plot of *ipcc* data(refined, California area)

Ideal performance

In general, the greatest possible speedup of a parallelized implementation relative to a serial implementation of the same algorithm can be computed by applying Amdahl's law (Amdahl 1967): $\frac{1}{S+\frac{1-S}{N}}$, where S is the fraction of time spent in serial (or non-parallel) execution and N is the number of parallel processors applied. For *resample*, the entire workflow is parallelizable into 10 independent flows, and each flow contains a stage where 730 independent subtasks are available. Thus the maximum possible speedup, ignoring I/O, for $N \leq 10$ is N, and somewhat less for $N > 10$ (a small portion can exploit 730×10 parallelism).

For *ipcc*, the workflow contains a non-parallelizable portion, which should account for approximately 1/20 of execution time. If we assume $S = 0.05$, then the maximum speedup should be $\frac{1}{0.05+\frac{0.95}{N}}$, or approximately 3.5, 5.9, 7.7, and 9.1, for N={4, 8, 12, 16}, respectively.

Overall performance

Both benchmarks show significant benefits from SWAMP. Comparing SWAMP's performance in a 4-node × 4 core configuration versus the baseline non-SWAMP case, we find that overall end-to-end time can be reduced from approximately 99 min to about 10 min in the resampling case, and from 165 min to 3 min in the IPCC case, giving roughly 10× and 64× reductions, respectively. Eliminating transfer time alone accounts for roughly 2×, and 22×, with efficient parallel execution accounting for the remaining savings.

Raw computational performance is shown in Table 1, excluding transfer time and job preparation overhead (parsing, compilation, and workflow generation), which are both dependent on static workload and not on execution configuration.

Table 1 Computational performance sorted by CPU count

# CPUs	# Nodes	PPN	$t_{compute}$ s	Input Rate MB/s	Speedup
resample					
1	1	1	3202.36	2.69	1
2	1	2	1703.97	5.06	1.88
2	2	1	1710.29	5.05	1.87
3	1	3	1264.77	6.82	2.53
3	3	1	1264.53	6.82	2.53
4	1	4	1022.67	8.44	3.13
4	2	2	987.24	8.74	3.24
4	4	1	1052.91	8.2	3.04
6	2	3	694.84	12.42	4.61
6	3	2	744.33	11.59	4.3
8	2	4	584.29	14.77	5.48
8	4	2	687.97	12.54	4.65
9	3	3	487.03	17.72	6.58
12	3	4	432.41	19.96	7.41
12	4	3	383.83	22.49	8.34
16	4	4	317.68	27.17	10.08
ipcc					
1	1	1	455.55	65.38	1
2	1	2	494.60	60.22	0.92
2	2	1	232.57	128.07	1.96
3	1	3	496.58	59.98	0.92
3	3	1	152.59	195.2	2.99
4	1	4	520.33	57.24	0.88
4	2	2	246.67	120.75	1.85
4	4	1	138.62	214.86	3.29
6	2	3	326.07	91.35	1.4
6	3	2	181.50	164.1	2.51
8	2	4	323.72	92.01	1.41
8	4	2	139.00	214.28	3.28
9	3	3	252.2	118.1	1.81
12	3	4	235.91	126.26	1.93
12	4	3	154.98	192.19	2.94
16	4	4	152.72	195.03	2.98

(Input Rate= $\frac{Input\ size}{t_{compute}}$. $t_{compute}$ does not include parse/workflow generation overhead: ≈ 300 s for *resample* and ≈ 0.5 s for *ipcc*)

Clustered and multi-core scalability

Figure 9a and b show plots of speedup versus node counts for *resample* and *ipcc* respectively. For completely CPU-bound workloads, ideal scalability should be reflected in linear speedup in the total number of CPUs, without regard to the number of CPUs per node. In both benchmarks, speedup increases linearly with the number of nodes but only in *resample* does speedup increase with the number of CPUs (see Subsection "I/O dependence"). From these results, we infer that larger systems with more processor cores per node, or more physical nodes, will have proportionally greater performance, except for extremely I/O bound workflows, where performance benefits come only with more physical nodes that can provide more read and write disk bandwidth.

In Fig. 10, we compare performance in three different cluster configurations while the total number of CPUs is held constant to 4. For *resample*, we see that performance is roughly constant, meaning that performance is effectively dependent on CPU count and is unaffected by master-worker management effects or resource contention. The 2 × 2 configuration seemed best, likely due to the increase in available disk bandwidth from 1 × 4 combined with low file dispersion (which impacts the 4 × 1 configuration). In contrast, *ipcc* performance is clearly dependent on node count rather than CPU count for reasons elaborated in Subsection "I/O dependence".

Clustered partitioning

In an earlier implementation (Wang et al. 2008), execution performance suffered from communication between master and worker nodes. As expected, performance improved by using a coarser-grained scheduler, which guaranteed that blocks of commands could execute without data staging and thus significantly reduced worker synchronization requirements. Without batching, we observed that while data traffic was reduced, scheduling traffic increased with the higher execution throughput, which limited scalability. In some cases, workers reported results to the master scheduler at an aggregate bandwidth of 500 kB/s.

In Wang et al. (2008) where a non-partitioned implementation was tested, we observed increasing performance as more worker nodes participated, but maximum speedup in a 4-PPN, 3 node configuration was limited to 3.5 ($t_{compute}$ =906 s). With partitioning,

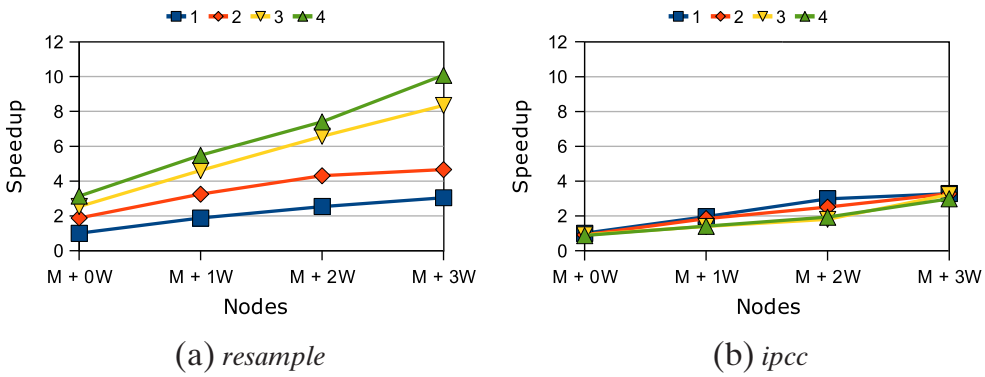

Fig. 9 Speedup by node count for PPN ∈ {1,2,3,4} (**a**, **b**)

the same configuration yields a speedup of 7.4, which is significantly closer to the theoretical ideal speedup, which should be close to 12 (see Subsection "Ideal performance").

I/O dependence

Because our workloads are far more data-intensive than compute-intensive, overall performance should be highly influenced by I/O performance, and this expectation is well supported by our tests. Intuitively, highly data-intensive computation (with less than one flop per input-byte) is usually bottlenecked by bandwidth to the disk (or other subsystems), and benefits little from increased processor core counts.

In Wang et al. (2007), we demonstrated the benefits of redirecting file I/O to an in-memory filesystem for single-machine installations, showing mild speedup (1.2 for *resample*) even without parallelization, and more significant speedup (1.5 for *resample*) with four parallel processes, relative to using only an on-disk filesystem. We observed similar results in testing the current system with and without using the in-memory filesystem. High data-intensive workloads thus incur a drastic penalty as processor parallelism increases without corresponding increases in parallelism of the I/O subsystems. For *resample*, redirecting file I/O to an in-memory filesystem proved effective, because of the majority of file I/O occurred on intermediate (temporary) files rather than input files.

The *ipcc* benchmark illustrates I/O dependence more clearly. Consider Figs. 9b and 11, which plot the same speedup data for *ipcc* showing speedup versus node count and PPN, respectively. For *ipcc*, performance is almost completely dependent on the machine count rather than the processor-count. Note that the Input Rate, or the average read bandwidth to the input, ranges between 48 MB/s and 65 MB/s, which is close to the disk manufacturer's specified maximum sustained transfer rate of 65 MB/s (Seagate Technology 2006). Indeed, additional process-parallelism without changing disk count reduces performance slightly, which is consistent with the expectation that additional disk contention lowers overall disk read bandwidth. In this case, disk access patterns were voluminous, but sequential, and OS buffering seems to have limited the penalty from contention.

Overhead and limitations

The parsing stage incurs significant overhead in the resampling benchmark, due to shell and environment

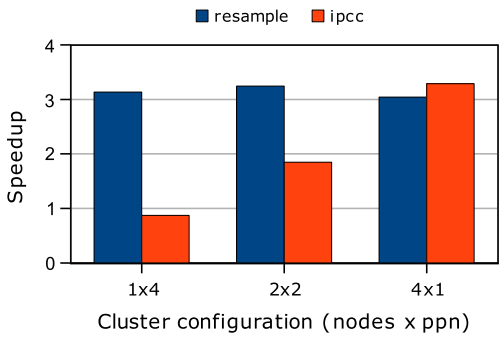

Fig. 10 Speedup in three cluster configurations

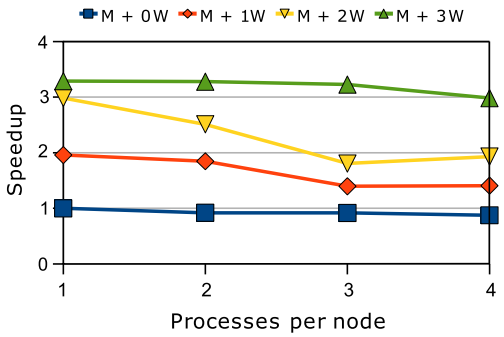

Fig. 11 *ipcc* speedup by PPN for various cluster configurations

variable handling in pure Python code. The excessive script length of 14,000 commands in 22,000 lines translates to roughly 5 min of overhead, pointing to a need for optimization in the parsing and command generation code, should such lengthy scripts become commonplace. The parsing overhead can be also reduced to insignificance for long scripts by implementing an early start mechanism that dispatches commands as soon as they are discovered as ready, before parsing completes. We expect that with tasks that are mostly or completely I/O bound, an early start mechanism would have the same effect on performance (or, in some cases, greater) as a fast parser.

Conclusions and further study

The high costs of data transport and large data volume mean that the traditional separation of data archival and user analysis is a barrier to scaling up data-intensive scientific analysis and exploration. We have addressed this problem by introducing a system that leverages familiar shell-script interfaces and NCO tools to specify safe computation on remote data. Although these analysis workloads have required low amounts of computational power and routinely perform less than one floating point operation per byte of input, their large data sizes (≈30–100 GB) has made them cumbersome to execute using other methods. Our implementation has made such computation efficient, by compiling scripts into implicit workflows, partitioning the workflows to reduce overhead and minimize data transfer, and scheduling execution according to data locality. Because of the minimal dependence on application semantics, our approach can be applied to provide some parallelism where it is not practical to modify or reimplement existing tools. Future work is aimed at implementing more advanced syntax support in the parser, and coarser work delegation for better performance where input data resides at multiple locations. SWAMP is scheduled to be tested at the National Center for Atmospheric Research (NCAR) as a method to provide server-side analysis capability for their community data portal. The SWAMP website is located at http://code.google.com/p/swamp.

Acknowledgements We acknowledge the modeling groups, the Program for Climate Model Diagnosis and Intercomparison (PCMDI) and the WCRP's Working Group on Coupled Modelling (WGCM) for their roles in making available the WCRP CMIP3 multi-model dataset. Support of this dataset is provided by the Office of Science, U.S. Department of Energy.

References

Adams JM (2008) Ensemble handling in GrADS and GDS: working with TIGGE data. In: Proceedings of the 7th global organization for earth system science portal (GO-ESSP) workshop, Seattle, 17–19 September 2008

Alpern B, Wegman MN, Zadeck FK (1988) Detecting equality of variables in programs. In: POPL '88: proceedings of the 15th ACM SIGPLAN-SIGACT symposium on principles of programming languages, pp 1–11. ACM, New York. doi:10.1145/73560.73561

Amdahl GM (1967) Validity of the single processor approach to achieving large scale computing capabilities. AFIPS Conf Proc 30(8):483–485

Beynon M, Kurc T, Catalyurek U, Chang C, Sussman A, Saltz J (2001) Distributed processing of very large datasets with datacutter. Parallel Comput 27(11):1457–1478

Box D, Ehnebuske D, Kakivaya G, Layman A, Mendelsohn N, Nielsen HF, Thatte S, Winer D (2000) Simple object access protocol (SOAP) 1.1. Tech. rep., W3C. http://www.w3.org/TR/2000/NOTE-SOAP-20000508/

Bourne S (1978) An introduction to the UNIX shell. Bell Syst Tech J 57(6):1971–1990

Capps SB, Zender CS (2008) Observed and CAM3 GCM sea surface wind speed distributions: characterization, comparison, and bias reduction. J. Climate 21(24):6569

Cornillon P (2003) OPeNDAP: accessing data in a distributed, heterogeneous environment. Data Sci J 2:164–174

Cui Y, Moore R, Olsen K, Chourasia A, Maechling P, Minster B, Day S, Hu Y, Zhu J, Majumdar A (2007) Enabling very-large scale earthquake simulations on parallel machines. In: Proc intl conf computational science, part I, vol 4487. Springer, Berlin Heidelberg New York, pp 46–53

Dagum L, Menon R (1998) OpenMP: an industry standard API for shared-memory programming. IEEE Comput Sci Eng 5(1):46–55 (see also Comput Sci Eng)

Deelman E, Singh G, Su MH, Blythe J, Gil Y, Kesselman C, Mehta G, Vahi K, Berriman GB, Good J, Laity A, Jacob JC, Katz DS (2005) Pegasus: a framework for mapping complex scientific workflows onto distributed systems. Sci Prog 13(3):219–238

Domenico B, Caron J, Davis E, Kambic R, Nativi S (2002) Thematic real-time environmental distributed data services (THREDDS): incorporating interactive analysis tools into NSDL. J Digit Inf 2(4):2002–2005

Doty BE, Kinter III JL (1995) Geophysical data analysis and visualization using GrADS. Visualization techniques in space and atmospheric sciences. In: Szuszczewicz EP, Bredekamp JH (eds) NASA. Washington, D.C., pp 209–219

Fettig A (2006) Twisted network programming essentials. O'Reilly Media, Sebastopol

Foster I, Voeckler J, Wilde M, Zhao Y (2002) Chimera: a virtual data system for representing, querying, and automating data derivation. In: Proceedings of the 14th conference on scientific and statistical database management, Edinburgh, 24–26 July 2002, pp 37–46

Gropp W, Lusk E, Skjellum A (1999) Using MPI: portable parallel programming with the message-passing interface. MIT, Cambridge

Hankin S, Harrison DE, Osborne J, Davison J, O'Brien K (1996) A strategy and a tool, ferret, for closely integrated visualization and analysis. J Vis Comput Animat 7(3):149–157

Hanselman DC, Littlefield BL (2004) Mastering matlab 7. Prentice Hall, Englewood Cliffs

Hsu WW, Smith AJ, Young HC (2005) The automatic improvement of locality in storage systems. ACM Trans Comput Syst 23(4):424–473. doi:10.1145/1113574.1113577

Li J, Liao K-W, Choudhary AN, Ross RB, Thakur R, Gropp W, Latham R, Siegel A, Gallagher B, Zingale M (2003) Parallel netcdf: a high-performance scientific i/o interface. In: SC. ACM, New York, p 39

Mathis M, Heffner J, Reddy R (2003) Web100: extended tcp instrumentation for research, education and diagnosis. SIGCOMM Comput Commun Rev 33(3):69–79. doi:10.1145/956993.957002

Natrajan A, Crowley M, Wilkins-Diehr N, Humphrey MA, Fox AD, Grimshaw AS, Brooks III CL (2004) Studying protein folding on the grid: experiences using charmm on npaci resources under legion. Concurr Comput Pract Exper 16(4):385–397

Rew RK, Davis GP (1990) NetCDF: an interface for scientific data access. IEEE Comput Graph Appl 10(4):76–82

Rubio-Solar M, Vega-Rodríguez MA, Pérez JMS, Gómez-Iglesias A, Cárdenas-Montes M (2008) A FPGA optimization tool based on a multi-island genetic algorithm distributed over grid environments. In: Cluster computing and the grid, 2008. 8th IEEE international symposium on CCGRID '08, pp 65–72. doi:10.1109/CCGRID.2008.96

Schweitzer R, Manke A, Hankin S (2008) Server-side OPeNDAP analysis—concrete steps toward a generalized framework via a reference implementation using F-TDS. In: Proceedings of the 7th global organization for earth system science portal (GO-ESSP) workshop, Seattle, 17–19 September 2008

Seagate Technology (2006) Barracuda 7200.9 Serial ATA product manual, rev. d edn

Szalay A, Gray J, Thakar A, Kunszt P, Malik T, Raddick J, Stoughton C (2002) The SDSS skyserver: public access to the sloan digital sky server data. In: Proceedings of the 2002 ACM SIGMOD international conference on management of data, Madison, 3–6 June 2002, pp 570–581

Tejedor E, Badia RM (2008) COMP superscalar: bringing GRID superscalar and GCM together. In: Cluster computing and the grid, 2008. 8th IEEE International Symposium on CCGRID '08, pp 185–193. doi:10.1109/CCGRID.2008.104

Wall L, Christiansen T, Orwant J (2000) Programming perl, 3rd edn. O'Reilly Media, Sebastopol

Wang DL, Zender CS, Jenks SF (2007) Server-side parallel data reduction and analysis. In: Advances in grid and pervasive computing: 2nd international conference, GPC 2007, Paris, France, May 2–4, 2007. Proceedings, lecture notes in computer science, vol 4459. Springer, Heidelberg, pp 744–750

Wang DL, Zender CS, Jenks SF (2008) Cluster workflow execution of retargeted data analysis scripts. In: Cluster computing and the grid, 2008. 8th IEEE international symposium on CCGRID '08, pp 449–458. IEEE Computer Society, Lyon. doi:10.1109/CCGRID.2008.69

Winer D (1999) XML-RPC specification. http://www.xmlrpc.com/spec

Xue Y, Wan W, Li Y, Guang J, Bai L, Wang Y, Ai J (2008) Quantitative retrieval of geophysical parameters using satellite data. Computer 41(4):33–40. doi:10.1109/MC.2008.132

Zender CS (2008) Analysis of self-describing gridded geoscience data with netCDF operators (NCO). Environ Modell Softw 23(10):1338–1342. doi:10.1016/j.envsoft.2008.03.004

Zender CS, Mangalam HJ (2007) Scaling properties of common statistical operators for gridded datasets. Int J High Perform Comput Appl 21(4):458–498. doi:10.1177/1094342007083802