Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Parallel Computing xxx (2007) xxx–xxx
PARALLEL COMPUTING
www.elsevier.com/locate/parco

Cluster versus grid for operational generation of ATCOR's MODTRAN-based look up tables

Jason Brazile[a,b,*], Rudolf Richter[c], Daniel Schläpfer[a], Michael E. Schaepman[d], Klaus I. Itten[a]

[a] *Remote Sensing Laboratories, University of Zurich, Winterthurerstrasse 190, CH-8057 Zurich, Switzerland*
[b] *Netcetera AG, Zypressenstrasse 71, CH-8040 Zurich, Switzerland*
[c] *Remote Sensing Data Center, German Aerospace Center (DLR), D-82234 Wessling, Germany*
[d] *Centre for Geo-Information and Remote Sensing, Wageningen University, Droevendaalsesteeg 3, 6708 PB Wageningen, The Netherlands*

Received 18 May 2007; accepted 8 November 2007

Abstract

A critical step in the product generation of satellite or airborne earth observation data is the correction of atmospheric features. Due to the complexity of the underlying physical model and the amount of coordinated effort required to provide, verify and maintain baseline atmospheric observations, one particular scientific modelling program, MODTRAN, whose ancestor was first released in 1972, has become a *de facto* basis for such processing. While this provides the basis of per-pixel physical modelling, higher-level algorithms, which rely on the output of potentially thousands of runs of MODTRAN are required for the processing of an entire scene. The widely-used ATCOR family of atmospheric correction software employs the commonly-used strategy of pre-computing a large look up table (LUT) of values, representing MODTRAN input parameter variation in multiple dimensions, to allow for reasonable running times in operation. The computation of this pre-computed look up table has previously taken weeks to produce a DVD (about 4 GB) of output. The motivation for quicker turnaround was introduced when researchers at multiple institutions began collaboration on extending ATCOR features into more specialized applications. In this setting, a parallel implementation is investigated with the primary goals of: the parallel execution of multiple instances of MODTRAN as opaque third-party software, the consistency of numeric results in a heterogeneous compute environment, the potential to make use of otherwise idle computing resources available to researchers located at multiple institutions, and acceptable total turnaround time. In both grid and cluster environments, parallel generation of a numerically consistent LUT is shown to be possible and reduce ten days of computation time on a single, high-end processor to under two days of processing time with as little as eight commodity CPUs. Runs on up to 64 processors are investigated and the advantages and disadvantages of clusters and grids are briefly explored in reference to the their evaluation in a medium-sized collaborative project.
© 2007 Elsevier B.V. All rights reserved.

Keywords: Atmospheric correction; Imaging spectroscopy

* Corresponding author. Address: Remote Sensing Laboratories, University of Zurich, Winterthurerstrasse 190, CH-8057 Zurich, Switzerland. Tel.: +41 44 635 51 61; fax: +41 44 635 68 46.
E-mail address: Jason.Brazile@geo.uzh.ch (J. Brazile).

0167-8191/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.parco.2007.11.002

1. Introduction

The ancestor of the MODTRAN line of atmospheric modelling software was first released in 1972, based on band models developed in the 1950's and 1960's used to describe atmospheric transmission and absorption behaviour [1]. Over time, not only has the software has been continuously updated [2,3], but also the underlying molecular spectroscopic database [4]. Although competing models also exist, MODTRAN has become established as a *de facto* standard in fields related to atmospheric physics and remote sensing. While many applications employing MODTRAN have usage patterns that require only tens of executions, others require hundreds or thousands. Examples of this usage include sensitivity studies [5,6], and the radiometric and spectral calibration of imaging spectrometers using known characteristics of solar and atmospheric absorption features [7–9].

When MODTRAN became more widely-used in operational, remotely sensed spectroscopy, invasive modifications of the typically end-user-compiled Fortran source code were developed by a group of users in order to introduce single-run parallelism, which resulted in efficient and scalable speedups [10]. For whatever reason, these code modifications were not officially incorporated into the standard MODTRAN release, which has grown to over 80,000 lines of code. The code base has moved on, producing multiple releases since then, and these parallel modifications have effectively been lost to the wider scientific community.

Meanwhile, further standard applications, including atmospheric correction software such as employed by ISDAS [11] and the widely-disseminated ATCOR family of software [12,13] were being built using MODTRAN calculations as the basis of their own algorithms. Atmospheric correction of multispectral/hyperspectral imagery involve calculations that depend on a number of varying MODTRAN-modifiable parameters defining atmospheric conditions (e.g., molecular absorber concentrations, aerosol scattering, optical depth) as well as observer and solar geometry, i.e., flight altitude, heading, ground elevation, view and solar zenith and azimuth angles. Therefore a large number of MODTRAN calls are required to process a single satellite or airborne scene. In software such as ISDAS [14] and ATCOR [15], these computations are usually performed off-line, and stored in LUTS prior to actual atmospheric correction of a scene to enable reasonable operational hyperspectral data processing times. The use of LUTS support the processing of a large variety of unrelated scenes in a much shorter amount of time than would be needed for direct computation.

In the case of ATCOR, researchers from multiple institutions began to collaborate, in an informal way, on extending functionality into novel special-purpose areas, ranging from the haze removal of low-spectral, high-spatial IKONOS imagery to the processing of hyperspectral, wide field-of-view (FOV) imagery as obtained from airborne (as opposed to space-bourne) instruments. This increased collaboration also increased the desire for higher turnaround on LUT generation and, in turn, created the opportunity to pool the collaborators computing resources, in an informal way, to speed up LUT generation.

In the LUT generation usage of MODTRAN, individual executions are independent of each other and input data parameters can be pre-computed ahead of time, leading to a workload that is "embarrassingly parallel" – i.e. there is no particular effort needed to segment the problem into a large number of parallel tasks. It is a common pattern in the employment of parallel computing for embarrassingly parallel problems to run multiple instances of a particular program over varying input parameters within a given problem space. This usage is explored here in both grid and cluster processing environments.

2. Method

The stated goals of this study include: the parallel execution of multiple instances of opaque third-party software, the consistency of numeric results in light of a heterogeneous compute environment, the ability to make use of otherwise idle computing resources available to researchers located at multiple institutions, and acceptable total turnaround time. These goals in addition to the parallel decomposition strategy are addressed individually.

2.1. High-level granularity of MODTRAN processing

It is tempting to consider making changes directly to the MODTRAN code itself, since only a minor number of input parameters are to be changed between executions. It is likely that common initialization code need be run only once and could thereafter be shared among multiple calls to only the subroutines affected by the changed parameter. However, in a multi-decade project like MODTRAN, the cost of software maintenance becomes a dominating factor to be considered in the development of any derivative solution. As mentioned in Section 1, this invasive approach was once taken [10], resulting in presumably positive short-term benefits, but of more questionable benefit in the long run, due to the changes not being merged into the official code. In this case, it can become infeasible to merge such changes after even a single follow-on release, especially when code restructuring is performed. Even the reformatting of comments, as occurred between two releases of the MODTRAN four series on its approximately 80,000 lines of code, can make externally tracked sets of changes very difficult to re-apply.

Therefore, the less invasive approach of treating "standard" MODTRAN as a "black box" and achieving parallelism at this higher-level of granularity seems attractive – at least when large numbers of independent runs are expected. As mentioned in [6], there still exist complex usages of MODTRAN (e.g. high resolution at-sensor simulations with both DISORT and correlated-k features enabled), which lead to run times measured in hours for a single execution. This mode of operation could still benefit from the single-run parallelism described above in [10]. However, typical executions such as those used for the ATCOR LUT each require less than a minute on current hardware, making a single-execution parallel version less important than it once was.

2.2. Parallel decomposition

The parameter space of the ATCOR LUT is chosen to adequately cover the problem domain at enough resolution such that interpolation of values between entry samples introduces limited error as compared with directly computed results. The six parameters that are varied are altitude, ranging from 1000 to 99,000 m; water vapour, from 0.4 to 2.9 g/cm^2; four built-in MODTRAN aerosol models, from rural to desert; ground elevation, from 0 to 2.5 km; solar zenith angles, from 0° to 70°; and visibility, from 5 to 120 km. These parameters are computed using 45,056 executions of MODTRAN – each performing six runs. The large number of data points is required to cover the wide range of geometry and weather conditions with a narrow grid enabling a linear interpolation in 6D space. This first-order interpolation is similar to the well-known bilinear interpolation in 2D, but neglects the mixed multilinear terms.

ATCOR's sequential Interactive Data Language (IDL) [16] code for generating the LUT consists of a single thread of execution that uses nested loops over the LUT dimensions to generate an input file, execute MODTRAN, parse the results and merge them into a data compressed binary data structure. Due to task independence, modifications for parallelizing this work involved only minor coding changes (see individual boxes in Fig. 1) to split the single thread of execution into two separate passes – one for input file generation and one for output merging. The resulting two pass process is broken into five stages described below and summarized in Fig. 1:

1. Generate inputs The original sequential code is run in two passes: a generation pass and a merging pass. This is done simply by guarding blocks of code with "if" statements. Additionally, a small amount of driver code was written to automatically permute all cases that were previously input from a graphical user interface.

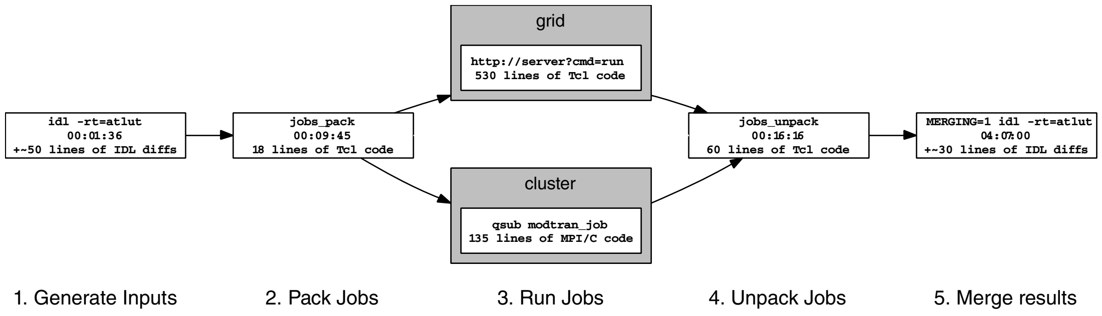


Fig. 1. Stages of processing.

2. *Pack jobs* A small script is used to bundle the generated input files into a collection of jobs organized for grid/cluster processing in a way that allows the data to be later re-arranged back to its original organization.
3. *Run jobs* The jobs are run in parallel either on a Message Passing Interface (MPI)-based [17] cluster or on the grid.
4. *Unpack jobs* A script is used to perform the re-arrangement as described in step 2 in preparation for the merging phase of the original sequential program.
5. *Merge results* In the 2nd pass, the original sequential code is re-run with a switch enabling the alternate (merging) blocks of code not taken during the step 1. A minor modification was also added to check for platform-related numerical problems discussed above.

2.3. Numeric consistency evaluation

Scientific programs such as MODTRAN are susceptible at many levels to numerical issues resulting from floating point calculations [18] and their inherent complexity [19]. The additional problems introduced due to heterogeneous computing environments have also been well-reported [20]. These include, but are not limited to, differences in: floating-point hardware parameters – which can influence results even in light of full IEEE-754 [21] compliance; operating system, compiler and language runtime library interfaces, which have the potential both to degrade results by masking exceptions or neglecting to save/restore parameters across calls, or to improve results by working around hardware problems in software; issues with application algorithmic integrity, such as the absence of proper scaling to avoid harmful underflow or overflow conditions; and finally, the communication of floating-point values. At least on the latter point, MODTRAN produces its results as tables of floating point numbers in ASCII format – thereby reducing byte-ordering and binary floating-point representation issues that might arise as data pass through multiple systems.

The apparent effect of some of the aforementioned issues were encountered in the first test run even on the homogeneous cluster – in the case corresponding to input parameters altitude 2000 m, 0.4 g/cm^2 water vapour, urban aerosol model, visibility 120 km, solar zenith angle 30° and ground elevation 1900 m. The computed results produced sporadic physically meaningless negative values for solar scattering and radiances. After this particular case was cross-checked on other hardware architectures without problem, it was decided to experiment with all five different compiler combinations (GNU's g77 3.3.4 and g77 3.4.4 [22], the Portland Group's pgf77 5.2-4 [23], and Intel's ifort 8.0, and ifort 8.1 [24]) available to the authors on the target platform, using only the default optimization (`-O`) configuration.

For this case, only MODTRAN executables compiled using the Intel compilers produced negative results, but other differences were also seen as shown in Fig. 2.

Since no investigation was made into whether the problem is due to a compiler/run-time error or a MODTRAN coding error, it was decided to universally use the GNU Fortran compiler, since it produced usable results over all cases on the cluster platform and was assumed to produce the most consistent results across the heterogeneous grid platforms – even though GNU Fortran generated code was up to 15% slower than executables produced by other compilers.

Due to this experience, it was furthermore decided to make the extra effort to roughly quantify numerical consistency across the multiple platforms. Therefore all cases were eventually computed on all platforms. Since it is not known which one provides "true" results, relative deviations [25] for each of the 45,056 cases are computed across the four platforms.

2.4. Runtime performance evaluation

Although runtime performance is not the highest priority of this study, careful analysis of the results can still be constructive. Therefore, in order to support comparison of runtime execution, the entire ATCOR LUT as computed from 45,056 calls to MODTRAN 4 Version 3 Revision 1 was generated multiple times using both a homogeneous cluster and a heterogeneous grid, varying the number of CPUs in powers of two up to 64. The Linux-based Matterhorn cluster [26] consists of 522 AMD Opteron 244 (1.8 MHz) CPUs connected with

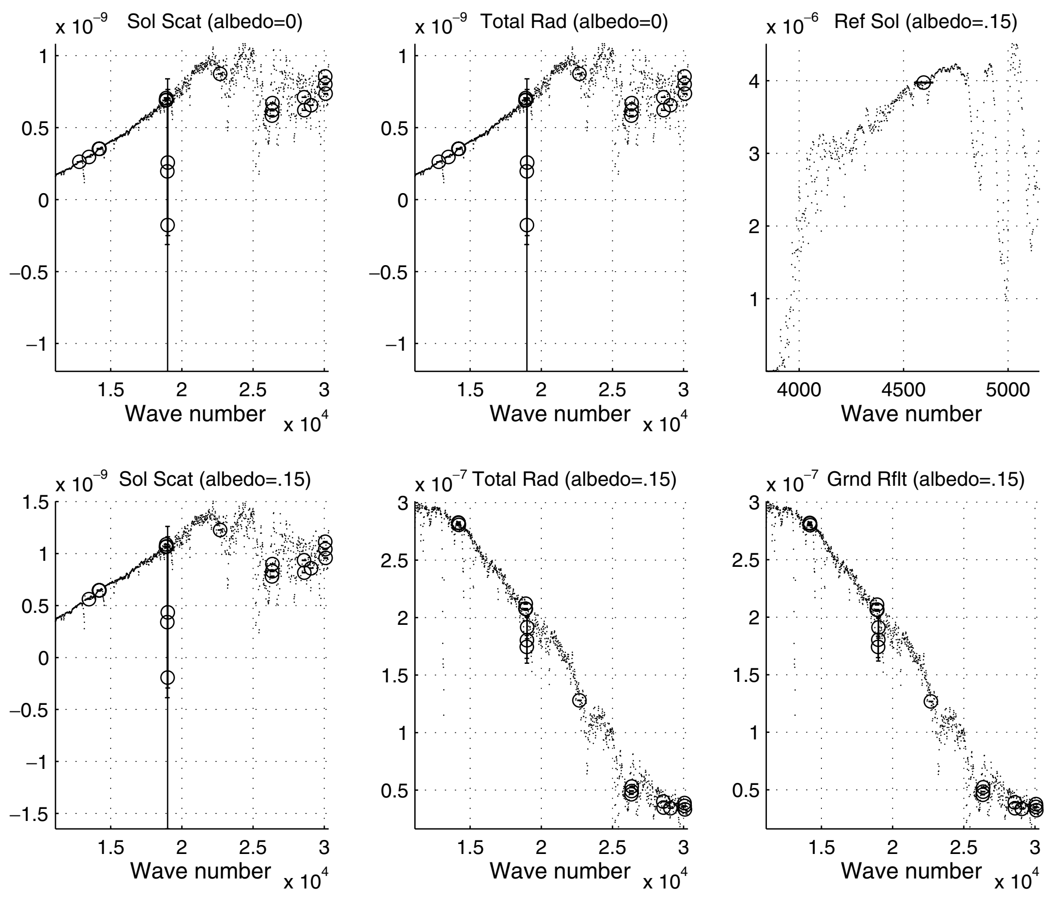


Fig. 2. Numerics problems with case 4332 of 45,056: circles/error bars show where any version differs 0.1% from mean of MODTRAN results produced from five common Fortran compiler/version combinations on an AMD Opteron. Note the negative values in Solar Scattering and Total Radiance (produced by two versions of a compiler from a single vendor).

100 Mbit Ethernet or Myrinet. The grid was composed of the personal workstations of the authors and their colleagues and the public Internet was used for network transport. A total of 99 CPU grid resources were available to the authors, including Intel x86-compatible Linux nodes, AMD 64-based Linux nodes, Apple G5-based MacOSX nodes, and Sparc-based Solaris nodes. The slowest grid CPU (a Sun SPARCstation-5) runs MODTRAN 72.4 times slower (20 min, 30 s versus 17 s) than the fastest grid CPU (an AMD Athlon 64 3500+) when running a particular single case (see Fig. 4).

2.5. Grid versus cluster

The clearest choice for new implementations of embarrassingly parallel problems intended for execution on a commodity cluster is the use of the MPI (Message Passing Interface) programming library [17] due to its ubiquity.

In contrast, for new grid implementations of embarrassingly parallel problems, there are several possibilities with varying features and trade-offs, but no clear choice for all cases has yet emerged. Some environments focus on the appeal of the client end-user experience ([27], via Folding@Home and SETI@Home), others focus on integration into a fully global network [28]. Perhaps the most well-known platform for the general scientific community is Condor [29]. Because of this, Condor was the initial selection for representing a typical grid implementation for this work.

2.6. Condor versus low fat grid

Since the simple batched queuing of multiple executions of a single program with varying input parameters is a common use case, Condor covers this explicitly in its users manual, with examples using the `initialdir` and `queue` parameters. However, after initial Condor installation, setup, and tests were performed, the authors wished to collectively add all spare machines to which they had access to a common processing pool. This is where the first problem with Condor was encountered – that it does not directly support pooling processors across firewalls. After discovering only complicated workarounds [30], which still did not solve the problem that none of our potential servers allowed incoming connections on non-HTTP ports, and all of our potential compute nodes only allowed outgoing connections on the HTTP port, our query on the Condor-users mailing list confirmed that the only potential solutions involved complex tunnelling over ssh and/or requiring several intermediary Condor Generic Connection Broker (GCB–) nodes.

Additionally, a second problem with Condor was encountered. When tests were performed on a subset of the problem, overall run-time results were worse than expected. Investigation revealed that Condor does not hide that it is designed for high-throughput but not low-latency. For example, there is a hard-coded (not end-user configurable) pause between each job invocation, which in our case represents a non-trivial percentage of the run time of a single MODTRAN execution on our fastest processors. Although there exists a third-party extension to Condor [31] which aims to improve this, the authors determined that a simple HTTP-based [32] grid could solve both problems that were encountered. When such a simple tool could not be found, we implemented our own.

The initial prototype, ghack [33], consists of three short standalone programs (<500 total source lines of code (SLOC)) capable of enabling geographically disperse groups, with only web browsing privileges and access to a single commodity ISP account, to pool their machines toward working on collective problems. The two primary design goals were ease-of-use (e.g. non-professional programmers) and ease-of-deployment (e.g. aside from write access to a CGI-script capable directory on a web server, no system administrator support, elevated system privileges, or firewall configuration is necessary.) A zip archive containing a collection of per-job directories of input files is HTTP file-uploaded to an Internet accessible web server from which worker nodes (possibly at different institutions) voluntarily request jobs when they are idle. The worker nodes use the same protocols as web-based email programs to download input files and, after processing, upload results. End-users can visit a simple status page to monitor operation. This software enables volunteers to informally collaborate in an ad-hoc resource-sharing grid that can be realized on the order of hours rather than the more typical weeks or months required when more formal coordination is involved.

Since performance was not a primary objective for our HTTP-based tool, it was surprising to observe that in a test within a single institution (the only possibility for standard Condor) involving 1/16th of the data set, a grid of 16 compute nodes ran in half the time needed for Condor.

This success of the initial prototype prompted its re-implementation reduced to two standalone programs and the use of the Representational State Transfer (REST) [34] web service architecture, for an even simpler-to-deploy, more flexible solution known as the *low fat grid* project [35,36].

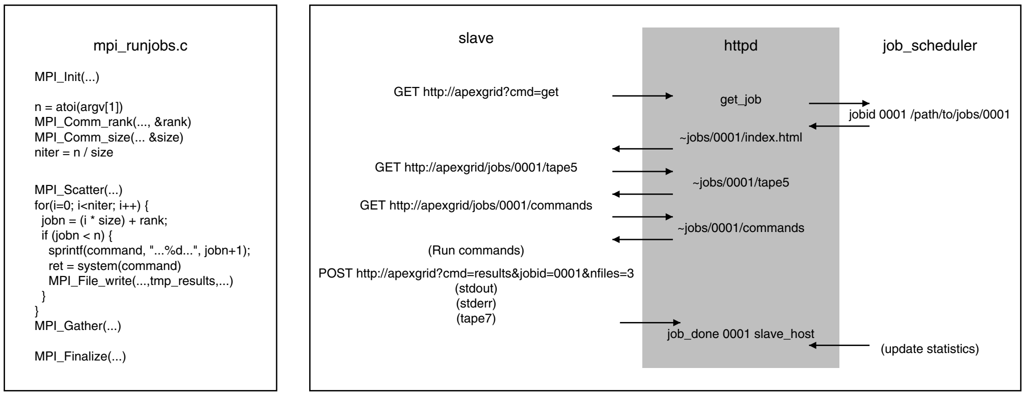
2.7. Scheduling

The scheduling of nearly identical jobs on a homogeneous cluster is not a difficult problem and initially a simple algorithm (shown in Fig. 3a) was chosen to evenly distribute jobs among all processors.

Scheduling nearly identical jobs for a heterogeneous grid is not as trivial. Not only do times vary due to CPU capacity but also due to disk and network latency and bandwidth, especially when using HTTP (or HTTPS when security is desired or required) over the public Internet.

For the grid, a simple scheduling algorithm was chosen whereby the nodes themselves request jobs from the job server. This automatically distributes more jobs to faster nodes. The resulting initial program flow for both cases can be seen in Fig. 3.

After the first cluster run, it was noted that the homogeneous compute nodes finished at unexpectedly different times – with the “fastest” node finishing several hours earlier than the “slowest” – amounting to about 10% of the total execution time in this case. After searching for and not finding any coding errors, it was



```
mpi_runjobs.c

MPI_Init(...)

n = atoi(argv[1])
MPI_Comm_rank(..., &rank)
MPI_Comm_size(... &size)
niter = n / size

MPI_Scatter(...)
for(i=0; i<niter; i++) {
  jobn = (i * size) + rank;
  if (jobn < n) {
    sprintf(command, "...%d...", jobn+1);
    ret = system(command)
    MPI_File_write(...,tmp_results,...)
  }
}
MPI_Gather(...)

MPI_Finalize(...)
```

slave
httpd
job_scheduler
GET http://apexgrid?cmd=get
get_job
jobid 0001 /path/to/jobs/0001
~jobs/0001/index.html
GET http://apexgrid/jobs/0001/tape5
~jobs/0001/tape5
GET http://apexgrid/jobs/0001/commands
~jobs/0001/commands
(Run commands)
POST http://apexgrid?cmd=results&jobid=0001&nfiles=3
(stdout)
(stderr)
(tape7)
job_done 0001 slave_host
(update statistics)

(a) Cluster (initial) (b) Grid

Fig. 3. Cluster versus Grid program flow.

decided to simply use the same scheduling algorithm as performed with the low fat grid. The cluster nodes request jobs when they are ready, leading to efficient self-adjusting job distribution.

In order to obtain a rough estimation of total run time on the heterogeneous grid, a single run of MODTRAN was timed on all available grid hosts and a speedup estimate was made by cumulatively adding nodes ranked from the fastest to the slowest. The results are shown in Fig. 4. For this problem and set of grid nodes, it was estimated that in the best case, using only 15 grid nodes would be only two times slower than using all nodes. Using more of the slow but numerous additionally available CPUs produce ever-dwindling returns.

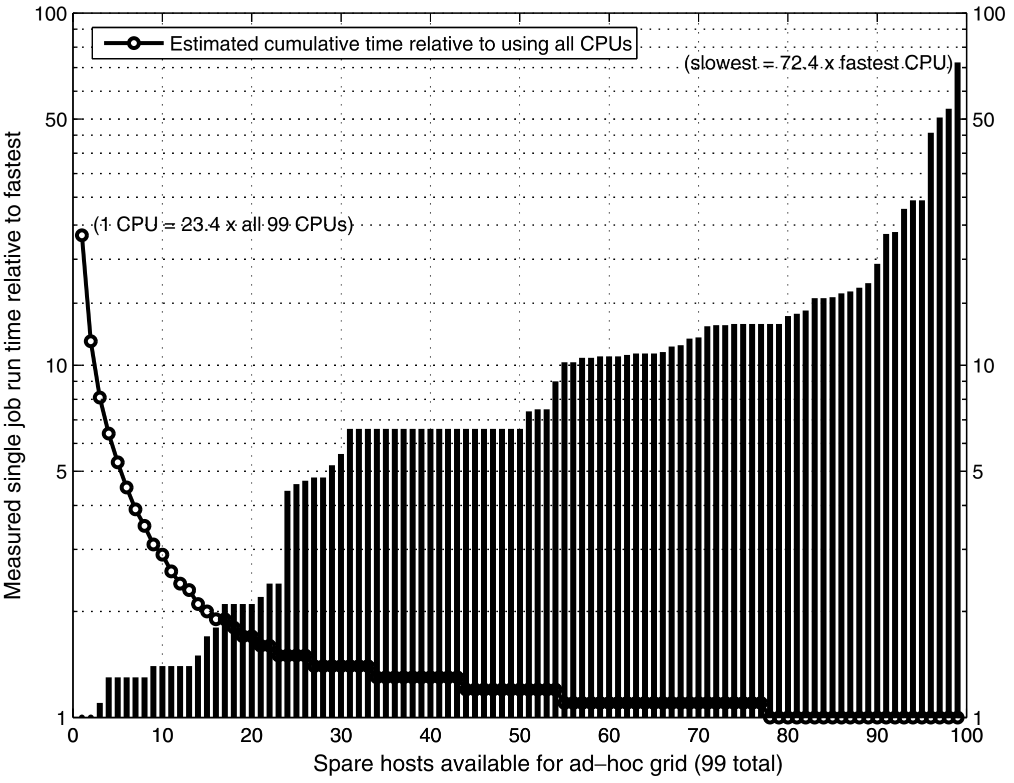


Fig. 4. Heterogeneous grid runtime estimation – relative CPU speeds of the 99 hosts are plotted along with an estimation of total runtime for a grid composed of increasingly more CPUs. It is estimated that 15 CPUs would run only 2 times slower than using all 99.

3. Results

Numerical differences of the GNU-compiled MODTRAN-generated results on the four different platforms are characterized in Fig. 5 by mean relative deviations [25] of a per-case "profile" across all 45,056 cases. The "profile" for a case, which is only used for comparison of that case with the other platforms, is defined as the mean of the subset of only those component radiance values that are eventually recorded in the resulting LUT. All cases which produced bad i.e., negative, results (see Section 2.3) are marked by vertical lines.

Total running times for the full computation are summarized in Table 1.

For characterizing per-node statistics, the mean and standard deviation of all MODTRAN run times (as measured from the node's point of view) were computed for each node that was used in any of the cluster or grid runs. These are shown in Fig. 6. Cluster nodes are named with the pattern `nodeNNNN` – the others are grid nodes.

Additionally, for analyzing per-case statistics, minimum, mean and maximum run times (measured from the node's point of view) were computed for each of the 45,056 MODTRAN cases. In Fig. 7, a quadratic fit of both the grid node and cluster node mean times are plotted, as well as the particular minimum, mean, and maximum times of the .05% most anomalous cases. A quadratic fit was deemed appropriate enough to show the trend, while reducing plot clutter enough to allow displaying of anomalous cases. Showing more anomalous cases would be difficult to read in the plot and would not reveal additional relevant information.

The speedup and efficiency of the parallel portion of the code (e.g. stage 3 described in Fig. 1) is plotted in Fig. 8 according to a standard algorithm [37].

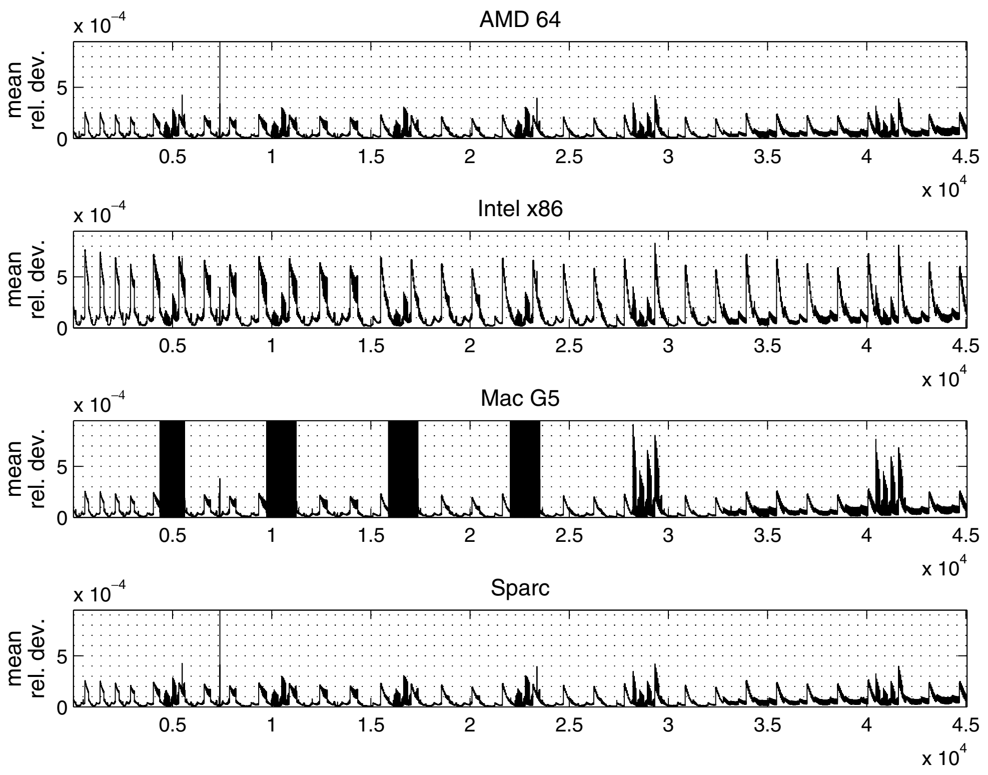


Fig. 5. Mean relative deviation [25] of the per-case "profile" for each of the four platforms across all 45,056 MODTRAN executions – vertical lines denote cases with bad (e.g. negative) values.

Table 1
Total running times in hours

(a) Sequential stages

Stage #	Time	Description
1	0.03	Generate inputs
2	0.16	Pack jobs
3	*see (b)*	Run jobs
4	0.27	Unpack jobs
5	4.12	Merge results

(b) Parallel stage (stage 3)

# CPUs	Grid	Cluster
2	113.87	125.09[a]
4	58.70	59.29[a]
8	36.64	31.26
16	21.53	16.19
32	15.33	7.68
64	12.95	3.93

[a] This was unfortunately not possible for the two and four CPU cluster runs, which is why the timing results for these cases are simulated by concatenating the results of the entire job split into eight equally sized partitions.

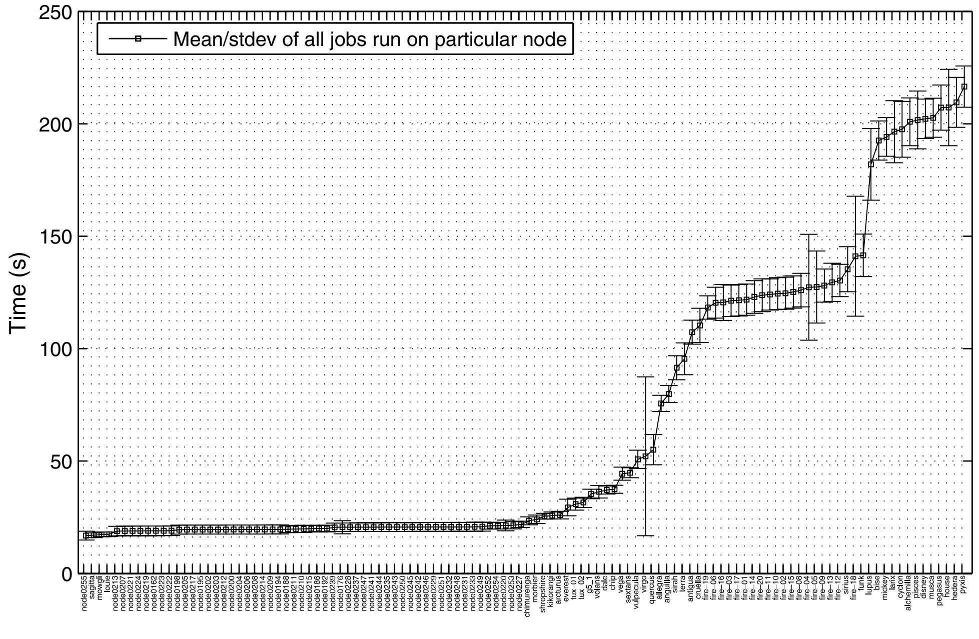


Fig. 6. Mean and standard deviation of run times per-node.

4. Discussion

4.1. High-level granularity of MODTRAN

Although it cannot be known what performance effect would have resulted from applying parallelization at a lower level of granularity i.e., within MODTRAN itself, according to Table 1, a sufficiently fast turnaround time is clearly achievable using either of the aforementioned grid or cluster approaches on current hardware. With continuous hardware improvements, this situation is only expected to improve.

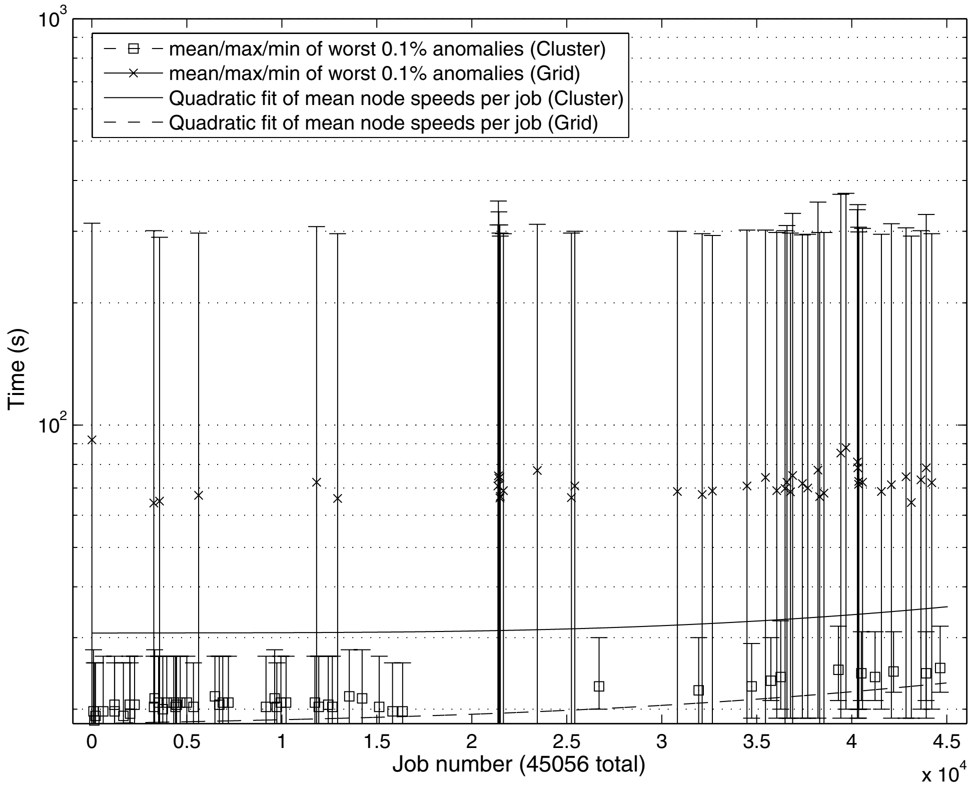


Fig. 7. Quadratic fit of node speeds (s) per job and extreme anomalies.

4.2. Numeric consistency evaluation

The most important information depicted in Fig. 5 is that other than the large number of cases where the Mac G5 produces bad (negative) values, the numerical results do not differ unreasonably from one another. The single 32-bit platform (Intel x86 compatible) was the only platform to produce no negative results, yet its results deviated the most from the other (64-bit) platforms, which showed consistent agreement. It is not known why the Mac G5 resulted in so many bad cases, but an attempt to use a more recent version of the compiler (GNU Fortran version 4.0) resulted in a compiler crash when attempting to build MODTRAN.

The most comforting outcome is that the AMD 64 and SPARC results are almost identical. Even the single "bad" case (7386 – corresponding to altitude 2000 m, 2.9 g/cm^2 water vapour, maritime aerosol model, visibility 15 km, solar zenith angle 60° and ground elevation 1500 m) for both platforms shows agreement where the AMD 64 produces only one negative value (at wavenumber 17,115 cm^{-1}) and the SPARC produced three (the same as the AMD 64 case but additionally at neighbouring wavenumbers 17,110 cm^{-1} and 17,130 cm^{-1}). Yet, for unknown reasons, neither the Intel 86 nor the Mac G5 produced bad values for this case.

4.3. The role of middleware

Interesting questions arise regarding the role of distributed and/or parallel computing middleware for implementing embarrassingly parallel problems. A large amount of services are offered, including: support for problem decomposition, problem characterization analysis, fine-grained monitoring, ease of deployment, ability to access large pools of resources, maximum configurability, exotic scheduling/rescheduling possibilities, high-throughput, low-latency, etc.

In this study, which we consider to be a medium-sized problem, we were primarily interested in two orthogonal questions: how does one achieve the fastest overall turnaround and how does one get the task imple-

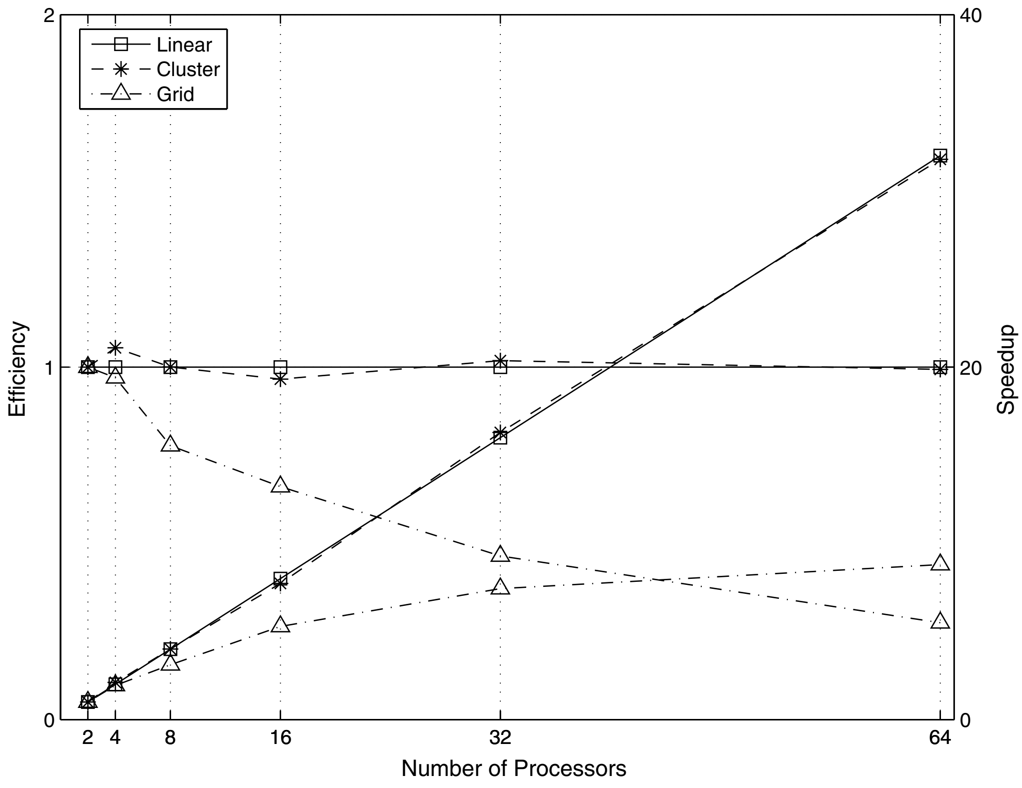


Fig. 8. Speedup/Efficiency of parallel MODTRAN jobs on cluster and grid.

mented with the least amount of effort? To artificially allow for the comparison between cluster and grid, we performed our own problem decomposition and merging of jobs and expected to only make use of queuing/scheduling/execution functionality. However, this effort would not have been much different if we needed to implement only one of the two solutions.

The answers we obtained to these two questions were surprising. Although Table 1 shows total run times which imply clearly higher potential turnaround on the cluster, it does not show *effective* turnaround times. In the cluster case, our centralized computing center's job queuing system is configured in a way that penalizes two and four CPU cases due to expected run time requiring more than the 48-h short-job queue time cut-off. The 32 and 64 CPU cases are penalized due to needing so many CPUs simultaneously. During this study, such jobs sat in the waiting queue for up to two months on multiple occasions before being killed due to cluster maintenance or system failure. In practice, end-users learn to get maximum job throughput by manually partitioning problems into single CPU jobs running just under the 48-h priority queue cut-off time and manually scheduling these themselves in ways that avoid job quantity penalties imposed by the queuing system. This causes the effect described above of starving jobs that were not so optimized to the queuing/scheduling middleware. Although such optimizations could potentially have been performed because our problem is embarrassingly parallel, for this study it was desired to accurately characterize cluster operation at specific CPU sizes, and therefore these techniques were avoided in running ATCOR LUT jobs. Over the course of a year, two cluster maintenance days were encountered that serendipitously allowed the author to be among the first to submit new jobs after a cluster reboot, allowing the 32 and 64 CPU jobs to finally run.

In contrast, none of the *low fat grid* cases ever required waiting longer than the upcoming weekend, which was done as a courtesy to the workstation owners.

Therefore, since cluster utilization was high and the cluster middleware was centrally tuned for a workload different from the natural workload of our problem, the fastest overall turnaround for ATCOR LUT generation was achieved using the grid.

Additionally, not including implementing the grid middleware ourselves in <500 source lines of code (SLOC), the least amount of realization effort was needed for the grid since implementation merely involved pre-generating varying input parameters to be placed one per directory in a zip archive, and convincing colleagues to run a small standalone client program on otherwise idle workstations.

4.4. Runtime performance evaluation

The speedup/efficiency plot in Fig. 8 for the parallel-only portion of the cluster runs is as expected for an "embarrassingly parallel" task. The plot of the grid result is more interesting, showing that speedup clearly diverges from ideal starting with 8 CPUs, and that in this setup, there is no good reason to use 32 CPUs over 16 – which largely validates the grid estimation predicted in Fig. 4.

The total running times in Table 1 show that for small numbers of CPUs (up to four), the grid can outperform the cluster. All of the grid CPUs used in these runs were newer than the CPUs in the homogeneous cluster, so this merely reflects Moore's law [38], which implies that commodity computational power doubles every 18 months. The difference between the two systems starts to clearly diverge after eight nodes. This is mostly due to there being fewer "new" CPUs available for a grid but it is also due to the increasing affect of the inefficiency of HTTP over the public Internet as the grid's network transfer protocol. In an after-the-fact investigation of possibilities for minimizing the effects of this inefficiency, a few experimental modifications were explored. First, a test run was performed using a 16 (homogeneous) node grid against a server running each of four different freely available web server programs in their standard configurations. On a subset of the parallel-only portion of the computation, this showed average multiple-trial run times ranging from 3.7 to 4.5 h, with the ubiquitous Apache software performing best. It was observed that at least one of the competing offerings did not support concurrent CGI processes in its standard configuration, which could explain the larger than expected spread. Another possibility for reducing the inefficiency of HTTP over the public Internet was investigated through the use of data compression. Again, a subset of only the parallel portion of the computation was run on a 16 homogeneous node grid comparing the difference between applying data compression on the output data before transferring the results over the network. In this case, running time was reduced from the 3.7-h average to an average of 1.3 h, where the per-job output data size was reduced from about 1 MB to about 300KB. Even with only 16 nodes, it appears that reducing the data bandwidth requirement also reduces strains due to concurrency on the server. In our case, input data is small compared to output data. However if the opposite were true, data compression could also be applied to the input. Likewise, if input were to consist of several small files, it is expected that improvement could be seen by using pipelined HTTP download [39] from the server to the grid nodes. In short, there is often potential for reducing the effect of inefficiencies inherent in using HTTP over the public Internet as a grid network transfer protocol.

4.5. Per-node statistics

There are three noteworthy features in the per-node graph (Fig. 6). First, the single cluster node, named `node0255`, seems to clearly be faster than the others. It has been confirmed by the cluster administrator that this node in fact contains faster CPUs and could be one reason why the original scheduling algorithm described in Section 2.7 was anomalous. Second, there is extreme variation shown by the grid node named `virgo`. A closer look at the data revealed the pattern that the slow run times occurred at clusters in time. After consulting the owner of the workstation it was revealed that he also often ran long-running jobs at night and on the weekends, implying that the variation was due to CPU contingency between multiple simultaneous CPU-bound processes. The final striking feature is that the slower grid nodes were not only clearly slower, but often had much larger variation. The fact that they are slower is due to their being SPARC-based CPUs as opposed to the others based on commodity PC hardware. The larger variation is explained in that these are nodes at a remote institution and the inefficiency of the public Internet network transport contributed more to their total run time.

4.6. Per job statistics

The quadratic fit of the mean running times per-case reveals two items of information. First, there is a consistent per job difference between the mean grid and the mean cluster fitting across the entire workload, as expected. Second, and unexpectedly, the per-case running times consistently increase as the case number increases. Examination revealed that the two slowest changing of the six varying MODTRAN parameters are flight altitude and amount of water vapour. It then makes sense that calculations such as multiple scattering take more computation at higher altitudes when there are more atmospheric layers to be considered.

When examining the extreme anomalies in Fig. 7, it is first noticeable that the grid variations are so much larger than the cluster variations. This is certainly expected due to the cluster being a closed non-interactive system with dedicated network. In fact, more interesting is that such large variations are possible in a closed-network cluster. This also probably contributed to the anomalous results of the original simple scheduling algorithm described in Section 2.7. The biggest variation was due to a Network File System (NFS) server reboot during the running of a job and the smaller variations are also due to NFS contingency during initial input and final output of data files.

The grid anomalies show the extreme effects of intermittent network partitioning on the public Internet – the largest grouping of effects appearing in the early morning as people begin to arrive at work.

4.7. Scalability

Assuming that all compute nodes are homogeneous, total job time is dominated by processing time (as opposed to I/O time), all jobs involve roughly the same amount of computation, and that the non-parallel portion of the problem (merging results) is relatively constant and fixed, then scalability is effectively linear since this is an embarrassingly parallel problem.

These assumptions are essentially true in the cluster case but certainly not with the grid, where compute nodes are heterogeneous, total time for any one job might be dominated by I/O for nodes that are far-away or behind slow-haul links. Because of the variability in bandwidth and latency of the internet transport and the differences in profile of fast/slow machines available at any one time for computation, it is impossible to give general scalability results for the grid case. However, for the grid case performed in this study, the level of scalability can be seen in Figs. 4 and 8 and Table 1.

5. Conclusions

The application of grid and cluster parallelization in executing thousands of runs of third party software with varying input parameters has been investigated, specifically for the generation of ATCOR's LUT, widely-used in earth observation applications. The importance of validating the numerical consistency of the results for numerically intensive scientific software such as MODTRAN, especially in light of heterogeneous computing environments, has been shown. In this case study, the potential was also shown for an ad-hoc, voluntary grid to provide faster overall-turnaround time than a better-equipped but highly-utilized and inconveniently-configured cluster.

It has been shown that for producing the MODTRAN based ATCOR LUT, both grid and cluster implementations can be used to reduce runtime from ten days on a single CPU, down to under two days using modest computing resources. When measuring only compute time (i.e. not wall-clock time) cluster-based runs sized above the final merging step runtime threshold, scaled linearly. In a heterogeneous grid using HTTP transport over the public Internet, run-time scalability is far from linear, however the detrimental additional I/O cost can be limited by proper use of web server software and the use of data compression on large input and/or output files.

High quality, free grid and cluster middleware is available, but for a particular set of uses (e.g. implementation by non-programmers, no access to privileged servers or services, execution across firewalls, etc.) the simple *low fat grid* has advantages over the well-known Condor middleware.

When both grid and cluster implementations solve a given embarrassingly parallel problem relatively well, it is worthwhile considering the non-runtime advantages and disadvantages that the two alternatives provide. Clusters allow for higher and more predictable I/O throughput, while grid environments may provide access to (fewer instances of) more recent and therefore higher performing processors. Cluster environments typically offer high quality software compilers and tools affecting numeric accuracy and/or aid in development time, but grid environments, composed of the resources of a circle of collaborative researchers may provide more immediate turnaround of small-to-medium-sized jobs due to lack of resource contingency. Clusters are typically homogeneous which can be an important characteristic for some problems, but for others the variation in platforms usually inherent in a grid can be a benefit, such as in the cross-validation of numeric results.

Also important is the correct granularity of problem decomposition. In this case study, critical third-party core software was treated as a "black box" – even though source code was available. Software evolves over time and it is likely that the cost of maintaining and re-applying locally produced code changes is too high, not because of the cost of initial development but either because the core software might change its structure too much between releases, or because the developers of the local changes are no longer available to re-apply them to a later release.

For prolific software such as the Fortran-based MODTRAN modeler, which may remain in wide use over spans of decades, longevity of parallel/distributed implementation is improved by allowing such decoupled control over job granularity. If the absolute performance of the core software stays roughly the same, the number of jobs that can be run could possibly be doubled on the same number of processors merely by upgrading the hardware. By building this insight into the software driving the "black box" executions, one can increase the useful longevity of the overall system.

Another possible dimension of scalability lies in the number of results generated, which in this case was optimized to fill the capacity of a DVD. As DVD capacities increase, higher numbers of samples can be generated to provide less average error due to inter-sample interpolation. Furthermore, additional dimensions (such as sensor viewing angle, in our case) could be added to the parameter space, in order to enable novel applications.

Acknowledgements

This work was supported in part by ESA/ESTEC contracts 16298/02/NL/US and 15449/01/NL/Sfe. The authors wish to acknowledge the support of the University of Zurich IT Services Department for access to the Matterhorn cluster [26]. J. Brazile acknowledges the support of Netcetera AG throughout the period of this work.

References

[1] G.P. Anderson, J. Wang, M.L. Hoke, F.X. Kneizys, J.H. Chetwynd, L.S Rothman Jr., L.M. Kimball, R.A. McClatchey, E.P. Shettle, S. Clough, W.O. Gallery, L.W. Abreu, J.E.A. Selby, History of one family of atmospheric radiative transfer codes, in: D.K. Lynch (Ed.), Proceedings of the SPIE Passive Infrared Remote Sensing of Clouds and the Atmosphere II, vol. 2309, pp. 170–183. <http://dx.doi.org/10.1117/12.196674>.

[2] A. Berk, L. Bernstein, G. Anderson, P. Acharya, D. Robertson, J. Chetwynd, S. Adler-Golden, MODTRAN cloud and multiple scattering upgrades with applications to AVIRIS, Remote Sensing of Environment 65 (3) (1998) 367–375.

[3] A. Berk, G.P. Anderson, P.K. Acharya, L.S. Bernstein, L. Muratov, J. Lee, M. Fox, S.M. Adler-Golden, J.H. Chetwynd, M.L. Hoke, R.B. Lockwood, J.A. Gardner, T.W. Cooley, C.C. Borel, P.E. Lewis, MODTRAN5: a reformulated atmospheric band model with auxiliary species and practical multiple scattering options, in: S.S. Shen, P.E. Lewis (Eds.), Proceedings of the SPIE Algorithms and Technologies for Multispectral, Hyperspectral, and Ultraspectral Imagery XI, vol. 5806, 2005, pp. 662–667. <http://dx.doi.org/10.1117/12.606026>.

[4] L. Rothman, A. Barbe, D.C. Benner, L. Brown, C. Camy-Peyret, M. Carleer, K. Chance, C. Clerbaux, V. Dana, V. Devi, A. Fayt, J.-M. Flaud, R. Gamache, A. Goldman, D. Jacquemart, K. Jucks, W. Lafferty, J.-Y. Mandin, S. Massie, V. Nemtchinov, D. Newnham, A. Perrin, C. Rinsland, J. Schroeder, K. Smith, M. Smith, K. Tang, R. Toth, J.V. Auwera, P. Varanasi, K. Yoshino, The HITRAN molecular spectroscopic database: edition of 2000 including updates through 2001, Journal of Quantitative Spectroscopy and Radiative Transfer 82 (2003) 5–44.

[5] D. Schläpfer, M.E. Schaepman, Modeling the noise equivalent radiance requirements of imaging spectrometers based on scientific applications, OSA Journal of Applied Optics 41 (2002) 5691–5701.

[6] D. Schläpfer, J. Nieke, Operational simulation of at sensor radiance sensitivity using the MODO/MODTRAN environment, in: Zagojiewski (Ed.), Proceedings EARSeL Fourth Workshop on Imaging Spectroscopy, Warsaw, Poland, 2005, pp. 611–619.

[7] R.O. Green, B.E. Pavri, T.G. Chrien, On-orbit radiometric and spectral calibration characteristics of EO-1 Hyperion derived with an underflight of AVIRIS and in-situ measurements at Salar de Arizaro, Argentina, IEEE Transactions on Geoscience and Remote Sensing 41 (6) (2003) 1194–1203.

[8] R.A. Neville, L. Sun, K. Staenz, Detection of spectral line curvature in imaging spectrometer data, in: S. Shen, P. Lewis (Eds.), SPIE Algorithms and Technologies for Multispectral Hyperspectral and Ultraspectral Imagery IX, vol. 5093, Orlando, FL, USA, 2003, pp. 144–154.

[9] J. Brazile, R.A. Neville, K. Staenz, D. Schläpfer, L. Sun, K.I. Itten, Scene-based spectral response function shape discernibility for the APEX imaging spectrometer, IEEE Geoscience and Remote Sensing Letters 3 (2006) 414–418.

[10] P. Wang, K.Y. Liu, T. Cwik, R. Green, MODTRAN on supercomputers and parallel computers, Parallel Computing 28 (2002) 53–64.

[11] K. Staenz, T. Szeredi, J. Schwarz, ISDAS – a system for processing/analyzing hyperspectral data, Canadian Journal of Remote Sensing 24 (2) (1998) 99–113.

[12] R. Richter, D. Schläpfer, Geo-atmospheric processing of airborne imaging spectrometry data. Part 2: atmospheric/topographic correction, International Journal of Remote Sensing 23 (13) (2002) 2631–2649.

[13] D. Schläpfer, R. Richter, Geo-atmospheric processing of airborne imaging spectrometry data Part 1: parametric orthorectification, International Journal of Remote Sensing 23 (13) (2002) 2609–2630.

[14] K. Staenz, D.J. Williams, Retrieval of surface reflectance from hyperspectral data using a look-up table approach, Canadian Journal of Remote Sensing 23 (4) (1997) 354–368.

[15] R. Richter, Atmospheric/topographic correction for satellite imagery, Tech. Rep. DLR-IB 565-01/05, DLR, Wessling, Germany, 2005.

[16] D. Stern, Interactive Data Language. <http://www.ittvis.com>, 2007 (accessed December 2007).

[17] The MPI (Message Passing Interface) Forum. <http://www.mpi-forum.org/>, 2007 (accessed December 2007).

[18] D. Goldberg, What every computer scientist should know about floating-point arithmetic, ACM Computing Surveys 23 (1) (1991) 5–48. <http://citeseer.ist.psu.edu/goldberg91what.html>.

[19] J. Barhen, V. Protopopescu, D.B. Reister, Consistent uncertainty reduction in modeling nonlinear systems, SIAM Journal on Scientific Computing 26 (2) (2004) 653–665.

[20] L. Blackford, A. Cleary, A. Petitet, R. Whaley, J. Demmel, I. Dhillon, H. Ren, K. Stanley, J. Dongarra, S. Hammarling, Practical experience in the numerical dangers of heterogeneous computing, ACM Transactions on Mathematical Software 23 (2) (1997) 133–147.

[21] ANSI/IEEE standard for binary floating point arithmetic: standard 754-1985, 1985.

[22] The GNU Fortran compiler. <http://www.gnu.org/software/fortran/fortran.html>, 2007 (accessed December 2007).

[23] The Portland Group compiler products. <http://www.pgroup.com/products/index.htm>, 2007 (accessed December 2007).

[24] Intel Fortran compiler for Linux. <http://www.intel.com/cd/software/products/asmo-na/eng/compilers/flin/>, 2007 (accessed December 2007).

[25] E.W. Weisstein, Relative deviation. <http://mathworld.wolfram.com/RelativeDeviation.html>, 2007 (accessed December 2007).

[26] A.J. Godknecht, C. Bolliger, University of Zurich Matterhorn cluster. <http://www.matterhorn.uzh.ch>, 2007 (accessed December 2007).

[27] D.P. Anderson, Boinc: a system for public-resource computing and storage, in: Fifth IEEE/ACM International Workshop on Grid Computing, 2004, pp. 4–10.

[28] I. Foster, C. Kesselman, Globus: a metacomputing infrastructure toolkit, The International Journal of Supercomputer Applications and High Performance Computing 11 (2) (1997) 115–128.

[29] M. Litzkow, M. Livny, M. Mutka, Condor-a hunter of idle workstations, in: The 8th International Conference on Distributed Computing Systems, 1988, pp. 104–111.

[30] B. Beckles, S.-C. Son, J. Kewley, Current methods for negotiating firewalls for the Condor system, in: S. Cox, D.W. Walker (Eds.), Proceedings of the Fourth UK e-Science All Hands Meeting, 2005.

[31] N. Palatin, G. Kliot, Low latency invocation in condor, Tech. Rep., Technion Computer Science Department, Israel, 22 pages, 2003.

[32] Hypertext transfer protocol. <http://www.w3.org/Protocols/>, 2007 (accessed December 2007).

[33] J. Brazile, Grid hack. <http://sourceforge.net/projects/ghack/>, 2007 (accessed December 2007).

[34] R.T. Fielding, Architectural styles and the design of network-based software architectures, Ph.D. Thesis, University of California, Irvine, 2000.

[35] J. Brazile, Low fat grid: a restful grid service for non-programmers, in: Jazoon'07 – The International Conference on Java Technology, Zurich, 2007.

[36] J. Brazile, Low fat grid. <http://code.google.com/p/lowfatgrid/>, 2007 (accessed December 2007).

[37] J. Demmel, Speedup. <http://www.cs.berkeley.edu/~demmel/cs267-1995/disc2/disc2.html>, 2007 (accessed December 2007).

[38] G.E. Moore, Cramming more components onto integrated circuits, Electronics 38 (8) (1965) 114–117. ftp://download.intel.com/research/silicon/moorespaper.pdf.

[39] C. Percival, Pipelined http get utility. <http://www.daemonology.net/phttpget/>, 2007 (accessed December 2007).