HEURISTIC APPROACH TO MULTIPLE-JOB SUBMISSION: A CASE STUDY

Yashwanth Chintana\textsuperscript{1} and Bijaya B. Karki\textsuperscript{1,3}

\textsuperscript{1}Department of Computer Science, \textsuperscript{2}Department of Geology and Geophysics, \textsuperscript{3}Center for Computation and Technology, Louisiana State University, Baton Rouge, Louisiana 70803, USA
ychint2@tigers.lsu.edu, karki@csc.lsu.edu

ABSTRACT
The number of parallel simulations (jobs) needed to solve a given scientific/engineering problem can be arbitrarily large. The computational performance is expected to depend on how the user submits these jobs using available computing resources. One option is to run one job at a time using the maximum number of processors that can be allocated to execute each job in the shortest time. The other option is to run all jobs simultaneously using the minimum number of processors per job so that each simulation can be very long. Here, we propose a heuristic approach to multiple-job submission in which a subset of jobs can be submitted at a time using some intermediate number of processors to significantly improve the total execution time. The subset size and the number of processors can be decided based on the number of jobs to be completed within a given time. In a shared system, the total job completion time also depends on the system load. We present some idea on choosing the best time for job submission to minimize the waiting time. Our heuristics are based on the scaling results of a parallel simulation program (VASP) for a Linux cluster.

KEY WORDS
Parallel Simulations, Job Submission, Scaling, VASP Program

1. Introduction
Over last two decades, computational approach has taken on increased importance in all areas of science and engineering research. In each computing center, numerous simulations are running thereby keeping computing nodes or processors busy all the time. While there have been major advances in computer hardware and the relevant simulation software, the computational demand is always increasing. The parallel/distributed computers are shared among many users who are often allocated limited amounts of computing time. A user job has to wait until it is dispatched for execution by a scheduler whenever the requested number of computing nodes or CPUs becomes available. As such, a lot of research has been going on job scheduling and its related areas, e.g., [1, 2].

Here, we deal with the job scheduling-like problem at the user level. To study a given research problem often requires many simulations to be completed within a finite time period using allocated computing resources, e.g. [3]. The number of needed simulations depends on parameter space to be sampled. For example, you might be calculating the density of a material as a function of pressure and temperature using molecular dynamics method. You need to complete 64 runs if you want to cover 16 pressures by 4 temperatures. Assume that you have access to 64 CPUs at any time (may be you have your own small cluster of 64 CPUs). Now the question is how to run the needed 64 simulations. One option is to run one job at a time using all 64 processors and perhaps this way you expect each job to be completed in shortest time. However, you need to run all jobs one by one. The total time is 64 times this shortest time. Other option is to run all jobs at the same time, each using only one CPU. This way each job takes the longest time to complete, however, by the time one job finishes all other jobs also finish. An intermediate option is to run a subset of jobs (e.g., 4 jobs) at a time using multiple processors (i.e., 16 processors per job). This is similar to the bag-of-tasks scheduling method [4, 5]. The selection of the right number of jobs to be submitted or run at a time depends on the computational efficiency of the parallel program under consideration. It is important to estimate the suitable number of processors for a particular job before submission so that the remaining processors can be used to run the remaining one or more jobs. Our aim is to suggest heuristics for submission of multiple jobs by the user at a time using available processors. This requires computational scaling analysis considering multiple jobs by the user at a time using available processors. This requires computational scaling analysis considering multiple jobs by the user at a time using available processors. Completion of all jobs (hence complete results) may require unusually long computational times so it is often desirable to look for partial results. In the context of many jobs, these partial results required by the user can be relevant in constraining the number of jobs to be run at a time. For above example, if the user wants to analyze the temperature variation at one pressure point (say, atmospheric pressure),
it makes sense to simultaneously run four jobs at four different temperatures. Thus, the response requirement can be considered in terms of the time taken to complete a chosen subset of jobs. Alternatively, it can be specified in wall-clock time and see how many jobs can be completed within that time. It is important to explore how the response time and the number of jobs influence each other during the execution.

In general, the users submit jobs to PBS or Loadleveler or any other scheduler used by the shared computing systems. The scheduler allocates the requested processors to these jobs and can handle multiple queues dynamically controlling their priority levels. This scheduling doesn’t consider external factors relevant to specific user applications. When one or more jobs are dispatched in a shared computing system is beyond the user’s control. The waiting time for a job to be dispatched can vary arbitrarily thereby delaying the completion of a subset of jobs and hence the completion of all jobs. If you have dedicated number of processors, your jobs do not need to wait as long as the request does not exceed the maximum number of processors available. In a shared system, understanding how busy the system is at different time (a particular time of day, a particular day in a week, and so on) is important[12,13]. The waiting time is highly unpredictable since it is also affected by other factors like user priority, job priority and load on the system. It is difficult to come up with a quantitative approach to reduce the waiting time but a qualitative idea can be useful to choose the right time to submit multiple jobs over a period of time.

The organization of the paper is as follows. First, we describe the essence of the parallel application program, which is used to develop a multiple-job submission approach. The scalability analyses for single job and multiple jobs are presented. The response and waiting time results and their relevance are also discussed.

2. Analyses

2.1 Parallel Application Program

Our case study uses the parallel program known as VASP (Vienna Ab-initio Simulation Package) [6]. We choose VASP because it is computationally intensive program and shows generally a poor scaling but yet one of the widely used software in computational materials research [3, 7]. This allows us to model a real material problem using first-principles molecular dynamics (FPMD) method within the quantum mechanical framework. VASP is written in Fortran 90 and, and supports dynamic memory allocation. It uses a single executable to do any type of calculation. The basic input parameters used by VASP are atomic configurations (crystalline structure or random arrangement), number of valence electrons, and various convergence parameters. The standard outputs include optimized atomic configurations, bulk properties such as energy, pressure and volume, electronic structures, magnetic moment, etc. More information including the scaling of VASP on NCSA Origin2000 can be found in [6].

We now present an empirical analysis of computational efficiency of VASP on Tezpur Linux Cluster [8] by simulating MgO material – which adopts a cubic crystal structure with well-defined positions of both Mg and O atoms. Execution time of a VASP job strongly depends on the number of atoms used in simulation box (supercell). Two supercells are considered; small system consisting of 64 atoms (32 Mg and 32 O) and large system consisting of 512 atoms (256 Mg and 256 O). For each system, the execution time varies with the number of processors used. As shown in Figure 1, small and large systems show
different behavior over the range of processors (4 to 256) considered.

The execution time behaves abnormally with increasing number of processors used in the case of small atomic system. The execution time decreases initially as the number of processors is increased up to 32 processors. This is normal trend – the larger the number of processors used the faster the execution. But the time starts to rapidly increase when more processors (64 and 128 processors) are used. This can be attributed to the communication overhead. On the other hand, for large atomic system the execution time decreases with increasing processors over the entire range considered as normally expected. The times with 128 and 256 processors are very similar. Also note that the execution times of large system are much larger than the execution times of small system. Our analysis thus shows that VASP scales poorly with both the number of atoms used in supercell and the number of processors used in simulation.

There is no point of using more than 32 processors for the small system and 256 processors for the large system. If there is a single job under consideration, it is natural to select the number of processors, which give the lowest execution time (i.e., 32 processors in the case of small system and 256 processors in the case of large system). However, if you have many jobs to run, then using that too many processors per job is not good idea as we discuss in the next section.

2.2 Average Total Execution Time with Multiple-Job-Submission

Total execution time is the total time taken to execute all jobs under consideration using a given number of processors. If one job is to be executed, choose the maximum number of processors with least execution time as suggested by Figure 1, e.g., 32 processors in the case of small system. However, if many jobs are to be executed, the total execution time and response time should be considered in choosing the right number of processors per job and hence the right number of jobs to be submitted at the same time. This way we can improve the total execution time, i.e., we can have all jobs done in the shortest time or acceptable short time. Our approach is explained below with some quantitative analysis.

Let $N_p$ be the number of processors available for executing jobs ($N_j$, total job number) under consideration, $P$ be the number of processors used for each job, $t_p$ be the time taken to execute a job with $P$ processors. The average total execution time ($T_{jp}$) for $N_j$ jobs to be executed with $P$ processors per job can be expressed as

$$T_{jp} = \frac{N_j t_p}{P}$$

where $N_j P > N_p$  \hspace{1cm} (1)

$N_j P$ is the total time for all jobs to finish and $N_j P / P$ is the number of jobs submitted at a time. In special case when $N_j P = N_p$, i.e., all required processors are directly available or all jobs are submitted simultaneously with $P$ processors per job:

$$T_{jp} = t_p$$

This means that all $N_j$ jobs can be completed using $N_p$ processors within time $t_p$, i.e. the time taken to execute one job with $P$ processors. We can assume that both $N_j$ and $N_p$ are constant so $T_{jp} \propto P t_p$. Figure 2 shows how the product $P t_p$ varies with the number of processors ($P$) used per job.

For example, if $N_j = 64$, $N_p = 256$, $P = 16$

$$T_{jp} = 64 \times 16 \times t_p / 256 = 4 \times t_p$$

Figure 2. Execution time $P t_p$ versus $P$ for two atomic system sizes on Tezpur Linux cluster
This means that all 64 jobs with each job using 16 processors will be completed within the time four times $t_p$ (16 jobs running at a time).

The main idea is to select the right number of processors per job required to complete all jobs within acceptable duration. In the case of the perfect scaling, the plots in Figure 2 should be horizontal line at the vertical axis value for $P = 4$. These plots give a measure of average total execution time and help us find out the right number of processors to complete execution optimally. The best choice is the case, which gives the lowest average total execution time. In both small and large systems studied, multiple-job submission

Figure 3. Numbers of jobs executed within response times of 90000 and 60000 seconds

2.3 Response Times in Multiple-Job Submission

Response time is the time taken by a job to start producing some output from the time the job is submitted to the queue. It is thus the sum of the waiting time and partial execution time during which initial results are produced. The response time needs to be considered when quick partial or intermediate results are needed. Here, these results represent the completed subset of jobs rather than the outputs from partially executed jobs. The response time is thus the time taken to complete a chosen subset of jobs. The question is how many processors should be used to finish all jobs in this subset. Alternatively, we can estimate the number of jobs, which can be completed within a given response time. As in the case of the execution time, we can ignore the waiting time in our analysis of the response time for the case of dedicated computing resources.

We give some idea on how response time and number of jobs are related. As shown earlier in Eq.1, $T_{p} = \frac{N_j + P \times t_p}{N_p}$ is the average total time execution. If we want to get the response time for a subset of jobs, we need to calculate the total time taken to execute $N_{rj}$ jobs where $N_{rj} < N_j$. The response time is given by

Figure 4. Response times in seconds when required subset of jobs $N_{rj}$ = 5 of 10 and response time of 5 jobs when $N_j$ = 10 jobs to be completed.
On the other hand, the number of jobs completed in a given response time \( (T_r) \) is given by

\[ N_{rj} = \left\lfloor \frac{T_r \times \frac{P}{T_p}}{N_p} \right\rfloor \]  

(4)

For example, if \( N_{rj} = 8 \), \( P = 64 \), \( N_p = 256 \), then 

\( T_r = \text{ceil} \left( \frac{8 \times 64}{256} \right) \times T_p = 2T_p \). The response time is twice the job execution time with 64 processors. Similarly, if \( T_r = 3T_p \), \( P = 64 \) and \( N_p = 256 \), then \( N_{rj} = \text{floor}(3T_p/T_p) \times (256/64) = 12 \). Therefore, 12 jobs can be completed within the response time of \( 3T_p \). Using available data and the above equations, we can see that reducing the number of processors per job will increase the number of jobs completed for a given response time. Figure 3 represents the number of jobs that can be completed with two different response times of 90000 seconds and 60000 seconds. This may be affected by the waiting time, which varies with the number of processors requested and the desired time period.

If the response time for subset of jobs is to be known, then the same equation (Eq. 3) has to be used to get the response time for \( N_{rj} \) jobs. To identify the best response time for a subset of tasks, choose the lowest response time for the set by varying the number of processors. Figure 4 shows the response time in seconds with the number of jobs required set to \( N_{rj} = 5 \) (only 5 jobs are running at a time) and \( N_j = 5/10 \) (5-job response time with a total of 10 jobs running at a time).

2.4 Waiting Time Variations

Running multiple jobs may take a lot of computational time not only because of long execution but also because these jobs have to wait in the queue for dispatch to the processors. If some pattern in job dispatch during a period of time is identified, then it may be easier to guess the right time or the right day to submit jobs so that the waiting time can be reduced significantly. Some guidelines can be set up by taking into account the human working pattern in their day-to-day activities. In most cases, these activities repeat over a period of time like a day or a week.

From results obtained from running several sets of jobs at different times in a day and different days in a week, some patterns are observed (Figure 5). In general, jobs, which are submitted on Saturday and Sunday, have very short waiting time. Sometimes, the waiting time is small when jobs are submitted during late night hours. When the system is very busy or has heavy load (because of relatively large jobs being dispatched during the weekend), it turns out to be the reverse case, i.e., Sundays and Saturdays may have much longer waiting times than those of weekdays. On government holidays, especially if it is a long weekend, many jobs can be submitted and executed. It is also observed that on the next day of a holiday, the systems seems to be less busy and has less waiting time. Figure 5 shows that irrespective of the number of processors used, the average waiting time is much less during weekends than that in weekdays. Thus, submitting jobs during weekends and holidays clearly have advantages over submitting jobs on weekdays.

3. Conclusion

The paper deals with a pre-job submission analysis, which may save time and resources considerably for a parallel program that needs to be executed multiple times. Some ideas are presented for choosing the appropriate number of processors per job in a multiple-job submission environment, the right time to submit jobs over a period of time like a week or a day, and defining the response time.
This requires the knowledge of the average total execution time as a function of the number of processors used per job. The size of subset of the jobs to be run simultaneously and the number of processors requested per job can be chosen to satisfy the user defined response requirements, i.e., how many jobs need to be completed by what time. Finally, it may be more productive if jobs are submitted during weekends and holidays. Our heuristics are based on the scaling results of a parallel materials simulation program in moderate size computing system. We plan to test them for more computationally intensive program in large-scale computing systems where our proposed multiple-job submission approach can make considerable difference on the overall computational performance.

Acknowledgements

This work was supported by National Science Foundation (EAR-0809489 and EAR-1014514). Computing facilities were provided by the Center for Computation and Technology (CCT) at Louisiana State University.

References


