ECCOMAS Thematic Conference on Computational Methods
in Structural Dynamics and Earthquake Engineering
M. Papadrakakis, D.C. Charmpis, N.D. Lagaros, Y. Tsompanakis (eds.)
Rethymno, Crete, Greece, 13-16 June 2007

PERFORMING INCREMENTAL DYNAMIC ANALYSIS IN PARALLEL USING COMPUTER CLUSTERS

Dimitrios Vamvatsikos

University of Cyprus
75 Kallipoleos Str, 1681 Nicosia, Cyprus
e-mail: divamva@ucy.ac.cy

Keywords: Structures, Earthquake Engineering, Nonlinear Dynamic Analysis, Incremental Dynamic Analysis, Parallel Computing.

Abstract. *Incremental Dynamic Analysis (IDA) is a novel procedure that has recently emerged to accurately estimate the seismic performance of structures using multiple nonlinear dynamic analyses under scaled ground motion records. Being a computer intensive method, IDA can benefit greatly by parallel execution. Our aim is to accelerate the computation of IDA analyses using realistic structural models and multiple ground motion records on commercial or academic analysis platforms that were designed to be run on a single processor. Taking advantage of an environment of multiple network-connected processors, it becomes possible to complete such difficult tasks "over the weekend". Several approaches in distributing the computational load between the processors are discussed, examining the feasibility of breaking up tasks at the level of a model (sub-structuring), a single dynamic run, or a single-record IDA study. The latter two methods are the simplest to implement using a task-farming technique where a master processor prescribes tasks for the independent slave processors. It is shown that this approach can be efficiently implemented by modifying the IDA hunt&fill tracing algorithm to balance the computational load among a number of non-identical processors. The result is a flexible, efficient and fault-tolerant parallel platform with excellent scaling that can rapidly perform multiple multi-record IDA studies within the typical computer network found in any engineering office.*

1 INTRODUCTION

Incremental Dynamic Analysis (IDA) is powerful computer-intensive method that offers a comprehensive evaluation of the seismic performance of structures (Vamvatsikos and Cornell [1]). Still, performing an IDA is a time-consuming procedure that is often considered to lie beyond the computational resources of professional engineers, most realistic structural models often requiring several days of computation on a single computer. With the proliferation of Local Area Networks (LANs) and powerful workstations, it is only natural to investigate methods for running IDA in parallel (Grama et al. [2]) using computers within a LAN as a loose cluster to simulate an expensive parallel computer.

What we propose emphasizes the need to rapidly perform IDA analyses "over the weekend" using an ensemble of processors, or Central Processing Units (CPUs). Such a group of CPUs is typically connected by a LAN or even via the Internet and can be used to analyze realistic, engineering-level models using with a large suite of ground motion records on an existing commercial or open-source analysis platform (e.g. OpenSEES McKenna et al. [3]).

There are two paths one can take when attempting to parallelize such a problem. One method would be to attempt parallelization within the dynamic analysis itself. Then we need to write special parallel code, that will often have to be custom-made or at least custom-compiled for optimal performance, and it will concurrently solve the system of equations for the model, using for example a domain-partitioning technique. This would effectively be a fine-grained parallelization of the IDA problem that is ideally suited to very large models with numerous degrees of freedom. It has the advantage that it can potentially allow a vast number of CPUs to participate in the problem (the larger the model, the more CPUs can effectively be combined to solve it) but it suffers from increased communication overhead (message passing) among CPUs and, most importantly, the need for efficient parallel programming techniques integrated in the analysis platform. In other words, we can't use most existing analysis programs to partition an IDA in this way, unless we have a version that can already run in parallel.

Fortunately, we do not necessarily need to take this route, especially as long as a single dynamic run of our model can still be performed within the memory and CPU limits of each single computer in the cluster. Since an IDA involves by nature the execution of numerous such single tasks, the problem easily falls into the "embarrassingly parallel problems" category. Instead of solving a single large-scale problem by partitioning it to several CPUs, we only have a large collection of smaller independent problems that may be serially executed in a single CPU but they could just as easily be assigned to a larger number of CPUs that run in parallel. In this way we minimize the communication overhead among computers while enjoying the additional advantage of using any existing single-CPU analysis software. Such is indeed the nature of IDA, and it would make sense to take advantage of it.

Perhaps the easiest way to parallelize such a process is the master–slave model, where a single master node controls task generation, while the slave nodes are simple workhorses that wait for the next assignment to perform. Still, assigning a priori the tasks among the processors is not optimal. Some tasks are simply more time-consuming than others. At the level of a single dynamic run, depending on the number of excursions into nonlinearity and the convergence difficulties, there may be significant differences in the computations needed. Additionally, the length of a ground motion record itself in some part dictates the time needed for a dynamic analysis. At the level of a single IDA study, even among records having equal lengths one could easily cause more inelastic cycles, or just have some steep pulse that demands many iterations for convergence. Thus, it is always best to let the master post the jobs in a shared address space,

or broadcast them to the slaves, and then let the slaves themselves pick a task whenever they become available, a process that is known as self scheduling (Grama et al. [2]).

Thus the only question left to answer is picking the level where we are going to divide the IDA. There are two obvious routes one could take: Assign tasks to CPUs at the level of single dynamic analysis or assign single-record IDA studies.

2 PARTITIONING IDA INTO SINGLE-RECORD TASKS

The simplest alternative is to choose a coarse-grained partition, where each node (i.e., processor) is assigned each time to run a single-record IDA on the structural model. Thus, we are able to use our existing code both for running the dynamic analyses, but also for tracing IDAs. The efficient hunt&fill technique (Vamvatsikos and Cornell [1]) may be employed locally at each node to its to achieve maximum reusability of our existing code.

The only possible loss when taking this path is the scalability. Now for N records we can only employ N-CPUs at best, i.e., as many as the records that we want to run, unless we are running more than one models, e.g., say a parameter study or a comparison of different design alternatives. Then we could efficiently assign $J \times N$ CPUs, where J is the number of different multi-record IDA studies we want to perform. Either way, in a typical engineering office it would seem rather unlikely to have more than 5-10 computers dedicated to this task, so the problem will run quite efficiently. Therefore, this is the simplest route that will let us use our existing non-parallel analysis software, plus any IDA running capabilities that we may have available.

Another problem with such a coarse-grained parallelization is the inefficient load balancing that may appear at the end of the analysis, when we may have more CPUs than records available. Whenever the number of CPUs is not a divisor of the number of records (e.g., 6 CPUs for 20 records), we will have some CPUs having finished their records and idling while the rest complete the IDA. For example if we have 6 CPUs and we are running a 20 record IDA study, assuming all records take the same time to complete would mean that 4 CPUs will run 3 records while the other 2 CPUS will run 4 records. When the final two records are being run we will have 4 CPUs idling and not contributing at all to the effort.

Such issues only become important when the number of CPUs is comparable to the number of records times the number of IDA cases. For most practical applications though, assuming a small-size cluster for a typical engineering office, this would hardly be the case. Whenever we expect to run a large number of records with a handful of CPUs, there is no reason to be concerned about scalability or load-balancing.

2.1 Single multi-record IDA study

All that is left now is deciding for a way to perform the actual task assignment. The classic master node model is a simple technique, but adapting it to be employed efficiently on IDA is an interesting process in itself that will be dealt with in the following pages. There are also several logistic issues regarding how much centralized is our scheme, i.e. what parts of a single-record IDA should the slave run and what parts should the master retain.

One approach would be to let the master pick up all the work of tracing (i.e. running the hunt & fill algorithm) and postprocessing the results supplied by the slaves. This may seem attractive, as it leaves all the decision making to the master and makes the slaves pure workhorses that run analyses and return data. Unfortunately it also raises the cost of communication quite far while it also makes the master a bottleneck in the whole process. The other obvious extreme

is letting the master only supply the record and model information and let each slave trace and postprocess all the results on its own. Then it will only return high grade, low size data to the master. This is actually a very efficient scheme, absolutely minimizing any communication costs, therefore it is the one we will adopt in our development.

Having decided on what the responsibilities of master and slave are going to be, the most important remaining issue when programming the tasks is making sure that the master node does not remain idle while the other nodes are already running analyses. In general the organizing, task communication and assembly of the results are simple work that takes almost no time compared to a full single-record IDA. Therefore, there is a good chance that our master CPU will be idling while the slaves sweat it out. Still, when talking about a cluster of 5-10 CPUs, having one idling is actually a very inefficient way to balance loads as we are immediately wasting 20-10% of the available capacity. The obvious choice is to have the master node assign one task to itself while it waits for other nodes to complete their own. Then another ugly problem appears, as the faster slaves may now have to wait for the master node to finish its self-assigned task before it manages to send them another.

A simple way out of this problem is to have the master broadcast in advance all the tasks that need to be performed for this IDA (i.e., send all records) and let the slaves perform self scheduling by picking a task themselves. Each task that is to be run by a slave is immediately removed from the list so that it is not executed by any other. When each slave finishes the analysis, it posts the results back in the common memory and goes for the next task. When all the tasks are finished, the master reads the fragmented, single-record IDAs and assembles them in the final multi-record IDA.

Assuming a single case/model IDA is to be run, the algorithm needed to coordinate the tasks is quite simple:

Master script:

Post model file and records to common memory
run slave script
assemble the results

Slave script:

while jobs exist
  pick first available job (record)
  run traceIDA on single-record & postprocess
  send results to common memory
end while

As long as we have a routine available that can be run a single-record IDA, we can use these two scripts on the respective master and slave nodes to get an instant parallel machine. Note that this structure is simple enough to be implemented not only with standard messaging libraries like MPI (MPI Forum [4]), but also with a simple file-keeping scheme at some common network drive. The master needs no knowledge of how many CPUs are available, therefore CPUs may be added to the task as they are powered on or become available from other jobs. In this form, the scheme is extremely resistant to CPUs crashing or going offline for whatever reason. As long as the common memory remains online, the remaining CPUs will finish the run. Even if the master node itself goes offline, the slaves will do all the runs. All we need is rerun the master script at any time to aggregate the results, a simple and quickly executed task.

2.2 Multiple multi-record IDA studies

When we want to run multiple IDAs in a parametric analysis, as we discussed before, it would be advantageous to modify our task-assignments to minimize idling of CPUs. The goal is to have the slaves running tasks all the time, therefore even if the current IDA is not finished due to some node running the final record, the rest of the CPUs should be set running the next case already. Obviously the above presented algorithms cannot achieve that.

The way to resolve this is to simply have the master post more that one IDA case at a time, and then check back each time to make sure that there are more than enough jobs posted in the common memory before it undertakes any single task. Also, due to different performance capabilities of the slave nodes (one may be a much slower CPU) there is a real chance that a single CPU may still be running the last remaining analyses for IDA case i while the others (and perhaps the node itself) are already running analyses for IDA case $i + 1$. Therefore the master must be flexible enough to take into account all such difficulties, keep posting and running jobs while at the same time checking back to assemble results from any IDA case that has finished. Whenever such a finished case is detected, the results are removed from the common memory and stored locally to save space.

Now, the requirements are much steeper, therefore the algorithm gets a little bit more complicated:

Master script:

while non-posted IDA cases exist OR posted ones are still being run
  if less than two IDA cases remain posted then post another one
    post new case model file and records to common memory
  endif
  run slave script
  if all records from a case are finished, assemble the results
end while

Slave script:

while jobs exist
  pick first available job (record)
  run traceIDA on single-record & postprocess
  send results to common memory
  if master node has called this procedure EXIT
end while

Note that we have slightly modified the slave node script as well. Since this is also used by the master, it should not trap the master to keep running until all posted tasks have finished. While this is happening the list of posted tasks may be emptied and the slaves will become idle. So we have added an if-clause that allows the master node to escape after finishing a single task in order to check whether more tasks need to be posted.

3 PARTITIONING IDA INTO SINGLE-ANALYSIS TASKS

While the above approach is adequate for the needs of a small cluster, a different approach is needed when using more CPUs. The idea is to assign tasks at the level of single dynamic analyses instead of single-record IDAs. This we could call a medium-grained partitioning of

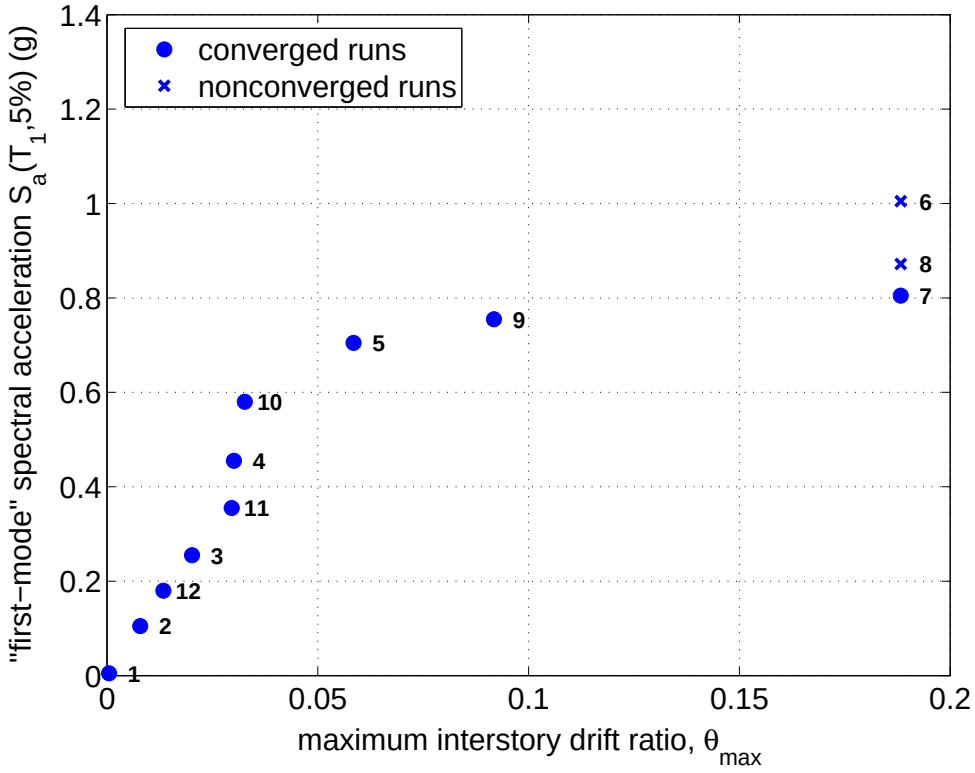

Figure 1: Tracing with one CPU.

tasks. One dynamic run per CPU is after all a simple and easy way to divide up an IDA. Since, say for an N-record IDA, each record needs about K-runs, we could ideally use up to $N \times K$ CPUs and still achieve excellent scalability for each CPU that is added to our total up to that number. With typical values of $N = 20$ records and $K = 8$ runs, we could use up to 160 CPUs in parallel to solve the problem theoretically in 1/160 of the time it would take us to do it with just one CPU. Of course, theoretically is the catch phrase here: As we have seen, not all runs are actually equal in computational load. Therefore, a more accurate estimate would be to say that with such a scheme scheme we can finish an IDA at best at the time it would take the worst single dynamic analysis to complete.

3.1 Serial hunt&fill

Despite the apparent advantages of this partitioning method, especially its superb scalability, we can easily run into problems when trying to implement it due to the nature of IDA. As Vamvatsikos and Cornell [5] have shown, it is not easy to predefine the number and level of the Intensity Measure (IM) of the runs to be performed. Currently, the best available way is to use a search algorithm, like the hunt&fill scheme (Vamvatsikos and Cornell [1]).

The algorithm actually consists of three distinct stages: (a) The hunt-up where the IM is increased at an quadratically accelerating rate until collapse (the flatline) is reached, (b) the bracketing phase where runs are performed between the highest converged and the lowest non-converged IM in an attempt to pinpoint the flatline, and (c) the fill-in phase where runs are performed between the largest gaps between convergent IMs that were left due to the accelerating steps of the hunt-up. An example of its application appears in Table 1 and Figure 1 for a nine story building (Vamvatsikos and Cornell [5]) where the 5%-damped first mode spectral

No.	calculations	$S_a(T_1, 5\%)$ (g)	$\theta_{\max}$
1		0.005	0.05%
2	$0.005 + 0.10$	0.105	0.79%
3	$0.105 + 0.10 + 1 \times 0.05$	0.255	2.02%
4	$0.255 + 0.10 + 2 \times 0.05$	0.455	3.01%
5	$0.455 + 0.10 + 3 \times 0.05$	0.705	5.85%
6	$0.705 + 0.10 + 4 \times 0.05$	1.005	$+\infty$
7	$0.705 + (1.005 - 0.705)/3$	0.805	18.83%
8	$0.805 + (1.005 - 0.805)/3$	0.872	$+\infty$
9	$(0.805 + 0.705)/2$	0.755	9.18%
10	$(0.705 + 0.455)/2$	0.580	3.27%
11	$(0.455 + 0.255)/2$	0.355	2.96%
12	$(0.255 + 0.105)/2$	0.180	1.34%

Table 1: Sequence of runs generated by the serial hunt & fill tracing algorithm for a nine story structure subjected to a single record.

acceleration $S_a(T_1, 5\%)$ is the IM of choice and the maximum interstory drift $\theta_{\max}$ is the desired response quantity (or engineering demand parameter, EDP). The pseudo-code of the algorithm is as follows:

% hunt-up
repeat
increase IM by the step
scale record, run analysis and extract EDP(s)
increase the step
until collapse is reached

% bracket
repeat
select an IM in the gap between the highest converged and lowest non-converged IMs
scale record, run analysis and extract EDP(s)
until highest converged and lowest non-converged IM-gap $<$ tolerance

% fill-in
repeat
select an IM that halves the largest gap between the IM levels run
scale record, run analysis and extract EDP(s)
until largest gap in converged IMs $<$ tolerance

Since, at least in the hunting-stage, the next step (and IM-level of the run) that the algorithm is going to take depends entirely upon the results of the previous run performed, this algorithm cannot be implemented with a single-analysis parallelization scheme. If on the other hand we use a simpler stepping algorithm with a predefined -stepping we may have a predefined set of steps but as long as we have no idea where the flatline might appear we always run the risk of having several nodes run dynamic analyses for the same record and one of them may discover a flatline that will make the other higher runs totally useless. Add to that the problem that constant stepping is not an efficient algorithm for IDA tracing and we can see that scalability and IDA-tracing efficiency cannot be achieved at the same time.

Still, it is worth exploring this direction even for small clusters where the number of CPUs is not a divisor of the number of records. Then, when we reach the final stages of our IDA running (be it a single or a group of IDAs in a parametric study) and we will have some CPUs idling. Even then it would be very attractive to somehow retain the efficiency of the previous scheme while not wasting the available computing power at the very end of the IDA completion.

3.2 Parallel hunt&fill

In order to achieve efficient partitioning at the single-analysis level we need to work within the guts of the IDA tracing algorithm, the hunt&fill scheme. Clearly, in the first two stages it is not possible to predict where the next run will be. Therefore, if we had two or more CPUs in our disposal at these stages of tracing, there is a good chance that we might be wasting runs. One or more CPUs may register an earlier collapse, thus invalidating all runs that are still being performed by other nodes at higher IMs. In general, the more CPUs we have dedicated to a record that is on the hunt-up or bracketing phase, the more runs we are likely to waste. On the other hand, if we are at the fill-in stage, the sequence of runs is completely predictable. Therefore, no matter how many CPUs we assign here, up to the maximum number of runs we want to expend, there will be no waste. Therefore the clear priority is to assign as many spare CPUs as possible to perform runs for any record that is in the fill-in phase. When no such cases exist and we only have records in hunt-up or bisecting stage, there is no obvious way to decide what the best CPU assignment would be.

Assuming we have M CPUs to apply to a hunt-up problem, the obvious method would be to assign each to a single run in the hunt-up sequence, always having the possibility that one of these runs (at worst the very first) will prove to be non-convergent and thus invalidate all the ones above. There is no obvious limit to how many CPUs one could supply to such a procedure. Still, due to the quadratically increasing steps in the hunt-up phase, it is makes sense that the more CPUs we assign here, the more runs we are likely to waste. If we have multiple records in the hunt-up phase, the trick is to evenly distribute the CPUs so as to minimize the number of CPUs working on a single hunted-up record, and thus reducing significantly the potential for wasted runs.

If on the other hand we have M CPUs to apply to a bracketing problem, an efficient way to do this would be to use as many of them as needed to achieve the desired accuracy and iteratively divide the space between the highest converged (C) and the lowest non-converged (NC) run to $M + 1$ segments. When the length of these segments becomes lower than the required minimum dictated by the user-specified resolution, we should simply assign all the remaining CPUs to filling-in below the C run.

One small wrinkle appears though, namely run-counting. A typical method of ensuring a consistent quality at a reasonable price for traced IDAs is to specify a given collapse capacity resolution of $a\%$ as above but allow only a specified maximum number of runs (Vamvatsikos and Cornell [5]). The problem is that if we count any run performed by spare CPUs during hunt-up or bracketing, there are going to be a lot of non-converged runs (especially in hunt-up) or many closely spaced convergent runs in bracketing. If we count all these, we may easily exceed our total allowance and still have a poorly traced IDA in the lower IMs. The only way to ensure consistent quality amongst all records is to count runs as if a standard single-CPU tracing was performed on each and every one of them.

Having said that, it seems that the bracketing stage is the least wasteful of the two. We have at least bracketed the position of the flatline, therefore any runs performed, even if they are non-converged, will help to narrow down the C and NC bounds that bracket the flatline. On the other

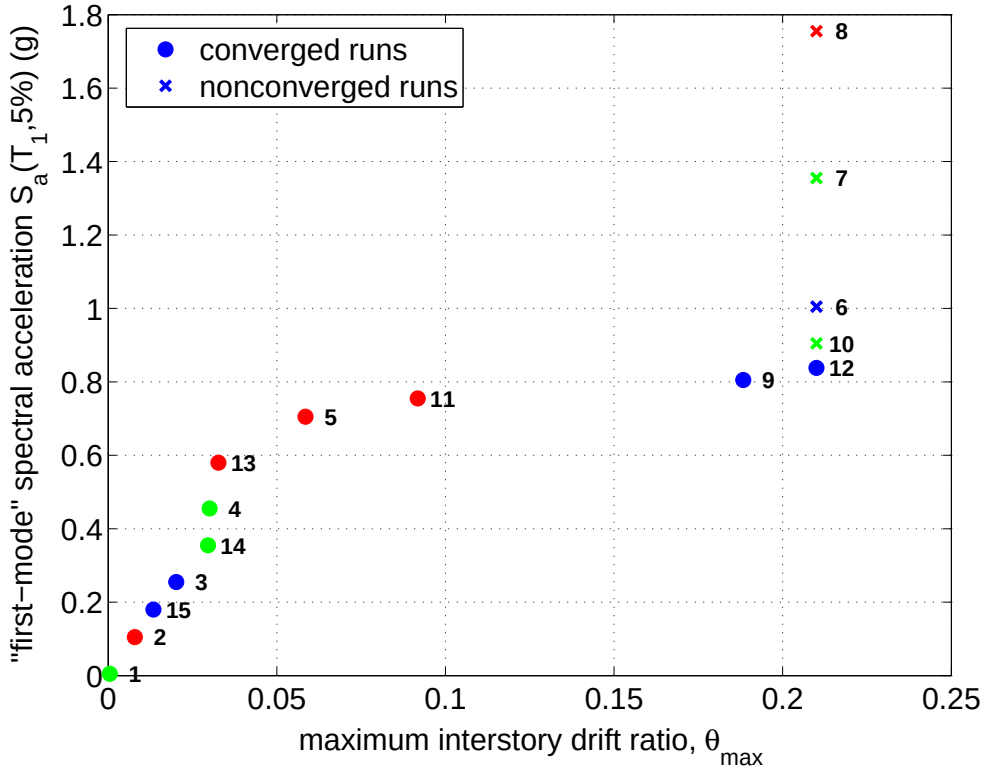

Figure 2: Tracing with three CPUs.

hand, the hunt-up stage can be potentially much more wasteful due to the rapidly increasing IM-steps. Therefore it makes sense to apply more CPUs to records that are in a bracketing phase rather than a hunt-up.

There are also other variations one could devise that might result to a lower number of runs wasted based on the concept of performing the fill-in even as we are hunting-up or bracketing. Conceivably, when hunting-up with one processor we could already assign extra processors to fill in between the large steps. Unfortunately, the largest gaps are always left close to the flatline, therefore there is a good chance that we are going to be filling in from the start the smaller gaps that might need no filling at all.

The resulting parallel hunt & fill script is slightly more complex, due to the need for continually posting new runs for other CPUs to perform and reading back the results, while performing a run or two in the meantime instead of waiting. An example of its application on the same building and record as before, using 3 CPUs appears on Table 2 and Figure 2. What is important to note is that two runs were wasted during hunt-up by cpu2 and cpu3. The rest of the phases happened to proceed in much the same way as for the serial hunt&fill.

% hunt-up
repeat
for i=1 to # of idle cpus
increase IM by the step
post job in common memory
increase the step
end for
Pick one of the posted IMs, scale record, run analysis and extract EDP(s)

No.	CPU	calculations	$S_a(T_1, 5\%)$ (g)	$\theta_{\max}$
1	cpu3		0.005	0.05%
2	cpu2	$0.005 + 0.10$	0.105	0.79%
3	cpu1	$0.105 + 0.10 + 1 \times 0.05$	0.255	2.02%
4	cpu3	$0.255 + 0.10 + 2 \times 0.05$	0.455	3.01%
5	cpu2	$0.455 + 0.10 + 3 \times 0.05$	0.705	5.85%
6	cpu1	$0.705 + 0.10 + 4 \times 0.05$	1.005	$+\infty$
7	cpu3	$1.005 + 0.10 + 5 \times 0.05$	1.355	$+\infty$
8	cpu2	$1.355 + 0.10 + 6 \times 0.05$	1.755	$+\infty$
9	cpu1	$0.705 + (1.005 - 0.705)/3$	0.805	18.83%
10	cpu3	$0.805 + (1.005 - 0.805)/3$	0.872	$+\infty$
11	cpu2	$0.705 + (0.805 - 0.705)/2$	0.755	9.18%
12	cpu1	$(0.705 + 0.455)/2$	0.580	3.27%
13	cpu3	$(0.455 + 0.255)/2$	0.355	2.96%
14	cpu2	$(0.255 + 0.105)/2$	0.180	1.34%

Table 2: Sequence of runs generated by the parallel hunt & fill tracing algorithm for a nine story structure subjected to a single record.

```
    Read EDP results from runs performed by other CPUs
  until collapse is reached

% bracket
  repeat
    delete any posted jobs left
    for i=1 to min(# of idle cpus, i_max)
      list = {highest converged IM, posted IMs, lowest non-converged IM}
      select an IM that halves or divides into 1/3-2/3 the largest gap in the list
      post job in common memory
    end for
    Pick one of the posted IMs, scale record, run analysis and extract EDP(s)
    Read EDP results from runs performed by other CPUs
  until highest converged and lowest non-converged IM-gap < tolerance

% fill-in
  repeat
    select an IM that halves the largest gap between the converged IM levels run
    post job in common memory
  until largest gap in converged IMs < tolerance
  repeat
    Pick one of the posted IMs, scale record, run analysis and extract EDP(s)
  until no jobs are left
```

3.3 Multi-level master-slave model

Having established the desired priorities for our algorithm, we need to set up the mechanics of task scheduling. The nature of the proposed algorithm dictates again the use of a multi-level hierarchical master/slave-1/slave-2 model. The master hands out single-record tasks as before. The slaves, performing self scheduling, attempt to pick single-record task. These become the

Algorithm	CPUs	runs	time (hrs)	performance
serial hunt& fill	1	20x12 = 240runs cpu1: 20 recs	40	
serial hunt& fill	3	20x12 = 240runs cpu1: 6 recs cpu2: 7 recs cpu3: 7 recs	14	40/14 = 2.86
parallel hunt& fill	3	18x12 + 2x13 = 242runs cpu1: 6 recs + 8runs cpu2: 6 recs + 8runs cpu3: 6 recs + 8runs	13.5	40/13.5 = 2.96

Table 3: Performance comparison of classic IDA tracing versus parallel tracing using 3 CPUs for a nine story structure subjected to 20 records.

slave-1 nodes which themselves act as masters to the remaining CPUs, the slave-2 nodes, and hand out single-run tasks to them. This becomes a dynamic three-tiered hierarchy were we need to make sure that the master and the slave-1 nodes are always busy; they should simply become slave-2 nodes once they are done distributing tasks and run analyses themselves. Then, they only need to check back once in a while whether the record (for a slave-1) or the full IDA (for the master) have been finished, to assemble the respective results and proceed to the next case, if any.

Therefore, the master script remains the same as before, e.g., for a parametric analysis as in the previous section. The slave script though has to change dramatically. Since it is the same script all slave nodes will be running, it has to figure out dynamically whether it needs to function as a slave-1 or a slave-2 node. If available records exist, it will take on a full record acting as a slave-1 node and start dealing out tasks to the available slave-2 nodes. If on the other hand there are no records left, it starts functioning as a slave-2 node and waits for single-run tasks to be posted, which it undertakes in the aforementioned priority. Thus, if we have fewer CPUs than records available, they will all run analyses as slave-1's and maintain all the advantages of a coarse-grained parallelization, i.e., no runs wasted.

Slave script:

```
while jobs exist
  if single-record job exists
    Pick first available job (record)
    Run parallel hunt&fill on single-record & postprocess
    Send results to common memory
  else
    Pick first available job (single run)
    Run single dynamic analysis & postprocess
    Send results to common memory
  endif
  If master node has called this procedure EXIT
end while
```

4 PERFORMANCE COMPARISON

In order to test the algorithms presented we used the nine story building with 20 ground motion records (Vamvatsikos and Cornell [5]). Each run takes about 7-15min, the longer times usually needed when close to collapse. We ran the analysis using (a) the serial hunt & fill algorithm on a single cpu (classic case) (b) the serial hunt & fill algorithm on three cpus (c) the parallel hunt&fill algorithm on three cpus.

When using the serial algorithm with 3 cpus, we are only losing efficiency when tracing the last two records. As seen in Table 3, cpu1 will remain idle while cpu2 and cpu3 trace the final records. Since we have many more records than CPUs, this is not a huge waste, simply underusing one cpu. Therefore we achieve a performance of 2.86 for 3 cpus, a slightly sublinear efficiency of 95%. When using the parallel version of hunt & fill, the idle cpu1 will alternate between helping cpu2 and cpu3 with their workload. Therefore, the wasted runs will be much less than what we observed in Table 2, where 3 cpus were tracing a single record, resulting to about 13 runs per each of the final two records. The efficiency thus rises to 2.96 for 3 cpus, or almost 99%.

Unfortunately such almost linear performance cannot be maintained if more cpus are brought into this problem. For example, if we were to use 60 cpus, i.e., 3 per record, then we would roughly have the case presented in Table 2 times 20. At an average of 14 runs per record, we would get a total of 280 runs, each cpu performing at most 5 runs. Thus the total process would take about 50mins or 0.83hrs, indicating an efficiency of 48 for 60 cpus, or 80%, obviously sublinear. Without the parallel version of hunt&fill we would only be able to use 20 CPUs and we would not be able to drop the processing time below $12 \times 10\text{min} = 120\text{min}$, i.e., 2 hours.

5 CONCLUSIONS

Several implementations of parallel IDA tracing for multiple cpus have been presented. The proposed algorithm uses a multi-level master/slave model with self-scheduling in order to exploit the idiosyncracies of a multi-record IDA study and achieve good performance by solving relatively efficiently the parallelization of a single record IDA. The final product is an excellent methodology that achieves almost linear efficiency for a relatively large number of cpus, much beyond what is typically found in an engineering office.

References

[1] D. Vamvatsikos, C.A. Cornell, Incremental dynamic analysis. *Earthquake Engineering and Structural Dynamics*, **31**(3), 491–514, 2002.

[2] A. Grama, A. Gupta, G. Karypis, V. Kumar, *Introduction to Parallel Computing*, *2nd Edition*. Addison-Wesley, Boston, MA, 2003.

[3] F. McKenna, G. Fenves, B. Jeremic, M. Scott, Open system for earthquake engineering simulation, 2000, URL `http://opensees.berkeley.edu`, [Feb 2007].

[4] MPI Forum, MPI: A message passing interface standard (ver. 1.1), 2003, URL `http://www.mpi-forum.org`, [Mar 2007].

[5] D. Vamvatsikos, C.A. Cornell, Applied incremental dynamic analysis. *Earthquake Spectra*, **20**(2), 523–553, 2004.