Performing Incremental Dynamic Analysis in Parallel[☆]

Dimitrios Vamvatsikos

University of Cyprus, Department of Civil and Environmental Engineering,
75 Kallipoleos Str, 1681 Nicosia, Cyprus

Abstract

Incremental Dynamic Analysis has recently emerged to offer comprehensive evaluation of the seismic performance of structures using multiple nonlinear dynamic analyses under scaled ground motion records. Being computer-intensive, it can benefit from parallel processing to accelerate its application on realistic structural models. While the task-farming master-slave paradigm seems ideal, severe load imbalances arise due to analysis non-convergence at structural instability, prompting the examination of task partitioning at the level of single records or single dynamic runs. Combined with a multi-tier master-slave processor hierarchy employing dynamic task generation and self-scheduling we achieve a flexible and efficient parallel algorithm with excellent scalability.

Keywords: structures, earthquake engineering, incremental dynamic analysis, distributed computing, parallel processing

1. Introduction

Incremental Dynamic Analysis (IDA) is a powerful computer-intensive method that has recently emerged to offer a comprehensive evaluation of the seismic performance of structures (Vamvatsikos and Cornell [1]) within the framework of performance-based earthquake engineering. Using numerous nonlinear dynamic analyses under a suite of multiply-scaled ground motion records, it allows for the detailed assessment of the seismic performance of structures for a wide range of limit-states, ranging from elasticity to dynamic instability and eventual collapse. 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.

Despite its computationally heavy nature, IDA has met wide acceptance and it is being used more and more by researchers to evaluate the performance of structures in a variety of settings. For example, Lee and Foutch [2, 3] and Yun et al. [4] employed IDA to evaluate the collapse capacity of multiple steel moment-resisting frames, while Liao et al. [5], Tagawa et al. [6] performed IDA to assess the performance of several 3D structural models. Pinho et al. [7] used it to evaluate the accuracy of static pushover methods on twelve bridges and Goulet et al. [8] relied on IDA to estimate seismic losses for a reinforced-concrete frame structure.

Apart from such focused studies, IDA has also been used extensively in wide parametric studies of single-degree-of-freedom systems (SDOF) featuring, sometimes, millions of individual dynamic runs. Especially within the framework of estimating seismic performance uncertainty and sensitivity to model parameters, such extensive investigations have appeared repeatedly in the literature. Ibarra [9] has run thousands of IDA analyses to evaluate the response sensitivity and uncertainty of SDOF performance to modeling parameters. Similarly, the development of the SPO2IDA oscillator-performance prediction tool by Vamvatsikos and Cornell [10] necessitated the execution of about 7,200 30-record IDAs for fitting and another 540 for error estimation. Lastly, in the course of the ATC-62 project [11] the evaluation of different single-story systems with a wide range of in-cycle and cyclic degradation characteristics demanded at least 800 IDAs of 56 records each.

More recently, with the improvement in computer power, large parametric IDA assessments have become feasible even for complex multi-degree-of-freedom (MDOF) structures. Among the first were Ibarra [9] who used IDA to evaluate simple moment-resisting frames with parametric beam-hinges, and Haselton [12] who employed it to ascertain the collapse capacity of 30 ductile reinforced-concrete moment frames with heights ranging from one to twenty stories. Vamvatsikos and Papadimitriou [13] performed design optimization of a highway bridge, running 10-record IDAs on almost 1800 different bridge configurations. Finally, the recent focus on IDA as a tool to evaluate the performance uncertainty of structures via Monte Carlo simulation has led many researchers to run tens or hundreds of IDA analyses of complex MDOF structures: Liel et al. [14], Dolsek [15], and Vamvatsikos and Fragiadakis [16] have used anywhere from 10 to 200 multi-record IDAs each, employing, e.g., classic Monte Carlo with a response surface approximation, Monte Carlo with latin hypercube sampling or even approximate moment-estimating techniques.

All in all, in the past few years IDA has moved from the realm of the eccentric academic method, to an every-day tool that is being used en masse. Therefore, it is quite natural that

[☆]Based on a short paper presented at the ECCOMAS COMPDYN 2007 Conference

Email address: `divamva@ucy.ac.cy` (Dimitrios Vamvatsikos) there have been several attempts to improve its speed and reduce the considerable computational load it incurs. One avenue of research has been the move to better, more efficient intensity measures (IMs), than the typical 5%-damped first-mode spectral acceleration $S_a(T_1, 5\%)$, as exemplified by Vamvatsikos and Cornell [17], Luco and Cornell [18] and Baker and Cornell [19]. Using improved scalar or vector IMs reduces the record-to-record dispersion of IDA curves, thus needing less records to achieve results with the required confidence level. Another unique strategy with the same goal was implemented by Azarbakht and Dolsek [20] who used a genetic algorithm for optimal selection of records based on equivalent-SDOF IDA results, thus estimating the summarized IDA results for MDOF structures with a reduced number of records.

Nevertheless, all such approaches invariably result back to the evaluation of a number of single-record IDA curves, where little has been done to accelerate their computation. They are typically run sequentially on a single computer, or one central processing unit (CPU), or at most broken up manually into a few subsets of records that are fed to 2–3 CPUs with the results often being combined afterwards by hand. With the proliferation of computer clusters and parallel computing (Grama et al. [21]), it is only natural to investigate methods to accelerate IDAs. What we propose emphasizes the need to rapidly perform IDAs "over the weekend" using an ensemble of computers, typically connected by a Local Area Network (LAN) or the Internet, to analyze realistic, engineering-level models for a large suite of ground motion records using an existing commercial or open-source analysis platform (e.g., OpenSEES McKenna et al. [22]). In other words, we aim to utilize existing distributed computing methods to enable IDA computations in parallel using the resources already found in any engineering office or research laboratory. While this might seem straightforward at first, the idiosyncracies of IDA can make efficient parallelization a challenging issue that deserves to be discussed in detail.

2. IDA Fundamentals

Performing IDA for a structural model involves running nonlinear dynamic analyses of the model under a suite of ground motion records, each scaled to progressively increasing intensity levels, appropriately selected to force the structure to display its entire range of behavior, all the way from elasticity to final global dynamic instability [1]. The ground motion intensity level is measured by an intensity measure (IM), e.g., the 5%-damped first-mode spectral acceleration $S_a(T_1, 5\%)$, while the resulting structural response is typically represented by an engineering demand parameter (EDP), e.g., the maximum interstory drift ratio $\theta_{\max}$. Thus, each dynamic analysis can be visualized as a single point in the EDP-IM plane, as seen in Fig. 1(a) for a nine-story steel moment-resisting frame. Such points can be interpolated for each record to generate IDA curves of the structural response. The resulting curves can be seen for the nine-story building in Fig. 1(b) were we observe their complex nature: Rising in a linear elastic slope from zero intensity, then twisting their way after yield to the final plateau, termed the flatline, where the structure responds by displaying disproportionately large EDP-increase for very small increments of the IM. This is where the structure approaches global dynamic instability, ultimately evidenced as numerical non-convergence in a well-executed, accurate analysis [23]. By appropriately post-processing the IDA curves, one can determine the distribution of EDP-response given IM (or vice-versa), define appropriate limit-states and, in combination with probabilistic seismic hazard analysis [24], determine the mean annual frequency of exceeding designated performance goals [23].

As our goal is accelerating the estimation of the IDA curves, of primary interest in our development hereafter is the existence of this flatline, the enormous record-to-record variability it displays and how this complicates the subtle issue of "appropriately selecting" the IM-levels to efficiently arrive at the results of Fig. 1(b) by expending only a small number of runs. For example, the simplest algorithm that can be devised to trace a single-record IDA curve is the stepping-algorithm:

Algorithm. SERIAL_STEP
```
1 repeat
2     increase IM by the step
3     scale record, run analysis and extract EDP(s)
4 until collapse is reached
```

While extremely simple, it is rendered completely inefficient by our inability to properly choose the all-important IM-step. As seen in Fig. 1(b), any step selected will be too small for some records and too large for others, and we have no way of knowing *a priori* which is which, thus leading to a large waste of runs [23]. Therefore, we need adaptive tracing algorithms, like the hunt&fill [23] discussed later, that can be very efficient but remove our ability to know in advance the runs that are to be performed. As we will discuss in the following sections, this is what precludes wide use of the simplest parallelization schemes, since most tasks are dynamically decided after every dynamic analysis is processed.

IDA parallelization thus becomes an interesting topic that we will tackle by investigating it within the constituents of a modern IDA study: a) The basic level of a single run, a single (nonlinear dynamic) analysis of a structural model under a given ground-motion record (i.e., any point in Fig. 1(a)) b) a single-record IDA study, i.e., a set of single runs/analyses of the model for increasing intensity of a given record (e.g., any of the 20 curves in Fig. 1(b)), c) a multi-record IDA study, which is a collection of single-record studies of the same model under different records (represented, for example, by the set of 20 curves from Fig. 1(b)) and d) multiple multi-record IDA studies where many different models are analyzed via a multi-record IDA, sometimes differing only in their parameters and maybe even using the same set of ground motion records. While a single run/analysis might take anywhere from 30sec to 1hr, depending on the structural model, the record and the CPU, a set of multiple multi-record IDA studies may easily contain thousands of single runs, often lasting weeks or months on a single processor.

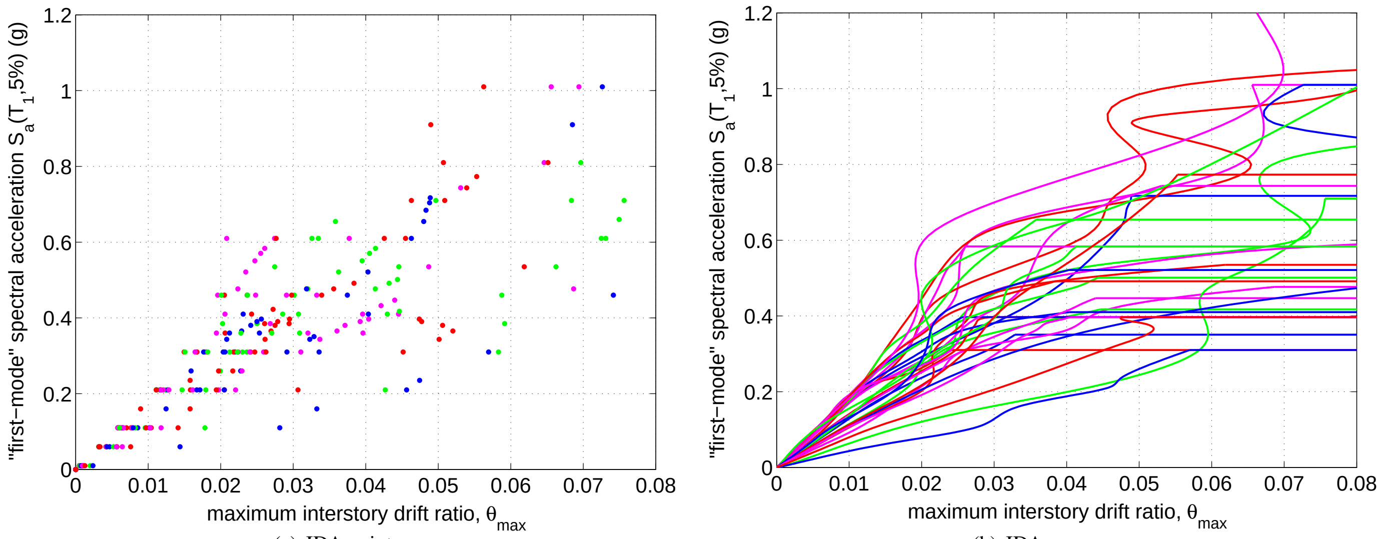


Figure 1: The discrete analysis points and the corresponding IDA curves calculated for 20 records for a nine-story steel moment-resisting frame.

3. Approaching IDA parallelization

There are two paths one can take when attempting to parallelize IDA. One method would be to attempt parallelization within the dynamic analysis itself. Then we need to employ special parallel execution software, that will often have to be custom-made or at least custom-compiled for optimal performance on the client's computer cluster, and it will concurrently solve the system of equations for the model using, for example, a domain-partitioning technique [e.g., 25]. 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, it needs a knowledgeable user and appropriate software. In other words, we cannot use most existing analysis programs to partition an IDA in this way, unless we have a version that can already run in parallel [e.g., 25].

Fortunately, we do not necessarily need to take this route as long as a single dynamic run of our model can still be performed within the limitations of each single CPU in the cluster. Since an IDA involves by nature the execution of numerous such single tasks, it becomes amenable to task-farming techniques [21]. 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 just workhorses that wait for the next assignment to perform. Still, *a priori* assignment of the tasks to the processors is not optimal. Some tasks are appreciably 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 dictates, in part, the time needed for a dynamic analysis. At the level of a single-record IDA study, even among records having equal lengths one could easily cause more inelastic cycles than the other, 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 common memory (e.g., a common filespace), or broadcast them via message-passing (MPI Forum [26]) to the slaves, and then let the slaves themselves take on tasks whenever they become available, a process that is known as self-scheduling [21].

Thus, the only question left to answer is selecting the level where we are going to divide the IDA process. 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.

4. Partitioning into single-record tasks

The simplest alternative is to choose a coarse-grained partitioning, where each node (i.e., processor) is assigned each time to run a single-record IDA on the structural model. This problem essentially falls into the "embarrassingly parallel" category (Foster [27]). 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 [1] may be employed locally at each node to achieve maximum reusability of existing code.

The only possible loss when taking this path is the scalability. For N records we can only employ N-CPUs at best, i.e.,

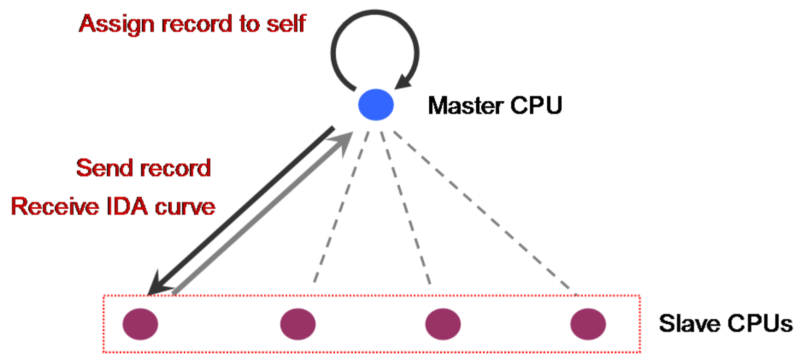


Figure 2: Two level master-slave hierarchy for partitioning IDA studies on a record-by-record basis.

as many as the records that we want to run, unless we are running more than one models, e.g., in a parametric 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. In a typical engineering office it would seem rather unlikely (at present) to have more than 5-10 computers dedicated to this task; processing will run quite efficiently for such small clusters. Anyway, this is the simplest route that will let us use our existing non-parallel analysis software, plus any IDA running capabilities that it may already incorporate, e.g., as happens with the Seismostruct software (Seismosoft [28]).

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, we will have some CPUs having finished their records and idling while the rest complete the IDA. For example if we have 6 CPUs running a 20 record IDA study, assuming all records take the same time to complete would mean that, in total, 4 CPUs will run 3 records each while the other 2 CPUS will run 4 records each. 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 studies. 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 with this approach.

4.1. Two-level master-slave model

All that is left now is deciding on a way to perform the actual task assignment. The classic master-slave model is a simple technique, but adapting it to be employed efficiently on IDA is an interesting process in itself. There are also several logistic issues regarding how much centralized our scheme should be, 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 communication overhead while it may also turn the master into 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, the slaves will only return high-grade, low-size data to the master. This is actually a very efficient scheme for our purposes, 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 issue remaining when assigning 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 already-postprocessed results is simple work that takes negligible time compared to any single-record IDA. Therefore, there is a good chance that our master CPU will be idling while the slaves sweat it out. 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%, respectively, of the available computing power. The obvious choice is to have the master node assign one task to itself while it waits for other nodes to complete their own.

Another issue appears then, 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 is to have the master broadcast in advance all the tasks that need to be performed for the IDA study (i.e., send all records), or, if that number is too large, dynamically keep adding more tasks to the work-pool than the number of CPUs 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 pool so that it is not executed by another. When a slave finishes the analysis, it sends the results back and starts with the next task. When all the tasks are finished, the master reads the fragmented, single-record IDA studies and assembles them in the final multi-record IDA.

The final communication model adopted is schematically shown in Fig. 2. In essence, it is a two-level master-slave hierarchical model where the master adds tasks to the work-pool while all CPUs, including the master, compete to grab tasks from the pool, thus implementing dynamic self-scheduling. The actual implementation of the proposed procedure depends on the size of the problem, so it differs slightly when running just one or many multi-record IDA studies.

4.2. Single multi-record IDA study

Assuming a single case (i.e., single structural model) IDA is to be run, the algorithm needed to coordinate the tasks is quite simple:

Algorithm. MASTERSCRIPT_ONECASE

```
1 post model file and records to shared memory
2 run slave script
3 assemble results
```

Algorithm. SLAVESCRIPT_ONECASE

```
1 while jobs exist do
2     pick first available job (record)
3     trace the single record & postprocess
4     send results to master
5 end while
```

As long as we have a tracing routine available that can perform a single-record IDA on one CPU, we can use these two scripts on the respective master and slave nodes to get an instant parallel machine. Note that this structure and all the similar ones that will follow are simple enough to be easily implemented with either a shared memory or a standard message-passing model, e.g., using libraries like MPI (MPI Forum [26]). Actually, the communication overhead is so low that it can also be realized via 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 computation as they are powered on or become available from other jobs. In this form, the scheme is extremely tolerant to CPUs crashing or going offline for whatever reason, not an uncommon occurrence in networks. 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 perform all the posted runs. All we need to do is rerun the master script to aggregate the results.

4.3. Multiple multi-record IDA studies

When we want to run IDA for multiple cases (structural models) in a parametric analysis, as discussed earlier, it would be advantageous to modify task-assignment to minimize idling of CPUs. The goal is to have the slaves running tasks non-stop until all IDA studies are finished: Even if the current IDA study is not finished due to some node running the final record, the rest of the CPUs should be set running the next case available. Obviously the above presented algorithms need some modifications to achieve that.

The way to resolve this is to simply have the master post more that one IDA study case at a time (together with appropriate model data), 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 master 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 study that is finished. Whenever such a finished case is detected, the results are assembled and, if applicable, removed from the common memory to be stored locally for saving space. Now, the requirements are steeper, therefore the algorithm becomes slightly more complicated:

Algorithm. MASTERSCRIPT_MULTICASES

```
1 while non-posted cases exist OR posted ones running do
2     if insufficient number of cases posted then post another
3     run slave script
4     if any case's records are finished, assemble results
5 end while
```

Algorithm. SLAVESCRIPT_MULTICASES

```
1 while jobs exist do
2     pick first available job (record)
3     trace the single record & postprocess
4     send results to master
5     if master node has called this procedure, EXIT
6 end while
```

Note that we have modified the slave node script as well. Since this is also used by the master, the script should not let it be trapped and kept 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. The same functionality can be achieved by having the master node spawn its own analysis process (the slave script) as a separate thread on the same CPU while maintaining a constant monitoring of the slaves.

5. Partitioning into single-analysis tasks

While the above approach may be adequate for the needs of a small office-level network, a different approach is needed when using higher numbers of CPUs, as available, for example, in a research facility. Actually, even for small clusters running a small number of records, it may be worth exploring a more efficient approach when the number of CPUs is not a divisor of the number of records. In such cases, when we reach the final stages of our IDA-running (be it a single or a group of cases in a parametric study) 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 computation.

The idea is to assign tasks at the level of single dynamic analyses instead of single-record IDAs, thus achieving a medium-grained partitioning of tasks. One dynamic run per CPU is after

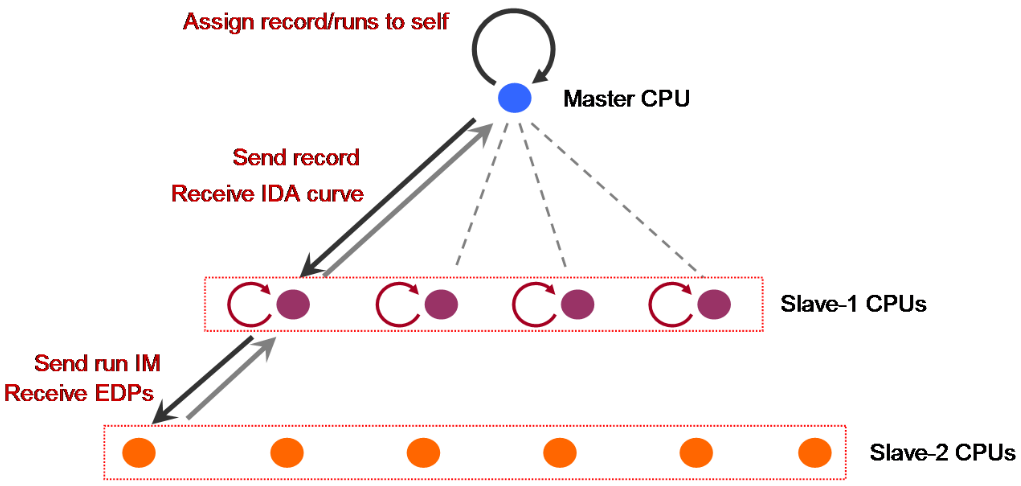


Figure 3: Three-level master-slave hierarchy for partitioning IDA studies on the basis of single dynamic analyses.

all a simple and easy way to divide an IDA. If 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. 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 with just one CPU. Of course, "theoretically" is the catch phrase here: As we have seen, not all runs are equal in computational load. Therefore, a more accurate estimate would be to say that with such a scheme, and assuming perfect task-scheduling, we can finish an IDA case at best at the time it would take the worst single dynamic analysis to complete. Let us now consider how such near-perfect scheduling can be achieved.

5.1. Three-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 the use of a multi-level hierarchical master/slave-1/slave-2 model. The master generates and adds single-record tasks to the work-pool, just like before. Then, all CPUs implement self-scheduling by attempting to pick single-record tasks. The processors that manage to get one become the slave-1 nodes which themselves generate single-analysis/run tasks. The remaining CPUs are the slave-2 nodes and select single-run tasks to perform. This becomes a dynamic three-tiered hierarchy where 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 completed, to assemble the respective results and proceed to the next case, if any. In this scheme, only the assignment of the master-CPU is static. The other CPUs always compete to get the single-record tasks, thus one CPU may take the mantle of a slave-1 for a given record but then, once it is done tracing it, it may not be able to find another record and have to become slave-2.

The proposed strategy is much more efficient than trying, for example, to further decentralize the role of slave-1 processors: We could have each slave-1 processor release its slot when it assumes even briefly its slave-2 role. Once more runs are needed, or a collapse has been hit, the first slave-2 that has seen the collapse or is out of runs will assume the slave-1 role for this record and pick up from where its predecessor stopped. While this offers robustness to network failures or computers shutting down, it makes for very complicated programming and a higher communication overhead. As each new slave-1 reads up the whole history of tracing for the record, it takes up valuable time and bandwidth because, in order to restart the tracing, it needs to process several pieces of information that are not available locally. Unless extreme resilience (e.g., due to CPUs being shut down and suddenly removed from the cluster) is required there is no need to go to such lengths. Having the same CPU maintain the role of slave-1 until a given record is traced makes for a simpler and slightly faster system.

The resulting three-level master-slave hierarchical model is shown schematically in Fig. 3. It is an efficient scheme that can operate at a significantly reduced communication overhead while keeping the work-pool filled and leaving no CPU idle, at least until the final stages of processing. Still, the devil is in the details, and in order for it to work we need to answer two important questions: a) how will a slave-1 orchestrate the tracing of a single-record IDA curve by generating tasks that can be run in parallel and b) how a slave-2 will self-schedule by prioritizing tasks in order to appropriately select which runs to perform to avoid wasting computing power.

5.2. Serial hunt&fill

The efficient tracing of an IDA curve is not a trivial issue, simply because of the existence and the unpredictability of the flatline. As we have already discussed, it is not easy to predefine the number and level of the intensity IM of the runs to be performed, which is why simple, easily-parallelizable schemes like the stepping algorithm become so inefficient. Currently, the

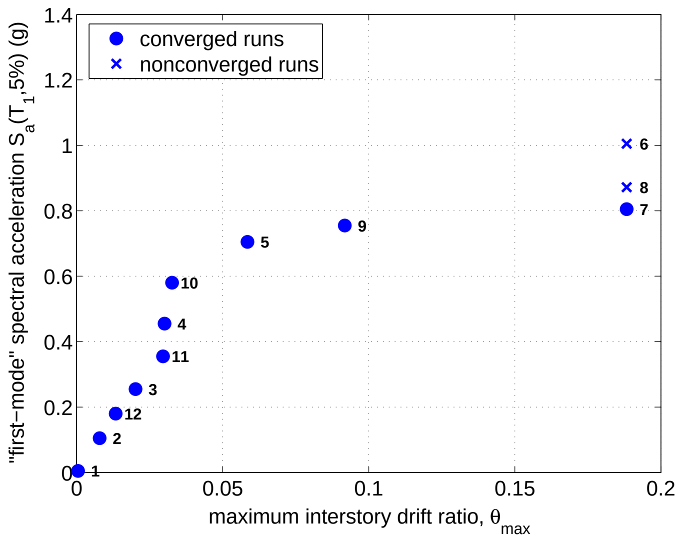


Figure 4: Tracing with one CPU.

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

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%

best available *serial* tracing-algorithm is the adaptive hunt&fill scheme [1]. In the interest of adapting it to work in a parallel setting, we will shortly describe its inner workings.

The algorithm consists of three distinct stages: (a) The hunt-up where the IM is increased at a quadratically accelerating rate until collapse (the flatline) is reached, (b) the bracketing phase where runs are performed between the highest converged intensity level IM_{C} and the lowest non-converged level IM_{NC} in an attempt to pinpoint the flatline by iteratively dividing the interval $[IM_{\mathrm{C}}, IM_{\mathrm{NC}}]$ according to a 1/3-2/3 ratio (to improve our chances for a converged run), and (c) the fill-in phase where runs are performed by halving the largest gaps that were left between convergent IMs due to the accelerating steps of the hunt-up. An example of its application appears in Table 1 and Fig. 4 for a nine story building [23] where the 5%-damped first-mode spectral acceleration $S_a(T_1,5\%)$ is the IM of choice and the maximum interstory drift ratio $\theta_{\max}$ is the desired EDP-response. The pseudo-code of the algorithm is as follows:

Algorithm. SERIAL_HUNTFILL

```
1  // hunt-up
2  repeat
3      increase IM by the step
4      scale record, run analysis and extract EDP(s)
5      increase the step
6  until collapse is reached
7  // bracket
8  repeat
9      select IM that divides into 1/3-2/3 the interval $[IM_{\mathrm{C}}, IM_{\mathrm{NC}}]$
10     scale record, run analysis and extract EDP(s)
11 until gap between $IM_{\mathrm{C}}$ and $IM_{\mathrm{NC}}$ $<$ tolerance
12 // fill-in
13 repeat
14     select IM halving largest gap between converged IMs run
15     scale record, run analysis and extract EDP(s)
16 until largest gap in converged IMs $<$ tolerance
```

At least in the hunting-stage, the next step to be taken each time, i.e., the determination of the next run's IM-level, depends entirely upon the results of the previous run performed. Therefore, this algorithm cannot be implemented as is for a single-analysis parallelization scheme. Thus, comparing with the previous single-record scheme, we can see that scalability and IDA-tracing efficiency cannot be easily achieved at the same time.

5.3. Parallel hunt&fill for dynamic task generation

In order to achieve efficient task-partitioning of a single-record IDA by a slave-1 down to the single-analysis level, we need to work within the guts of the IDA tracing algorithm, the hunt&fill scheme. Run by a slave-1 processor, its function should be to dynamically add and remove single-analysis tasks from the work-pool based on the information available from analyses that have already been performed. For the rest of this section we will proceed by building upon the theoretically simpler case of a static assignment of CPUs. In other words, we assume that the CPUs assigned to a record will remain and work on it until tracing is finished. While we will modify this strategy later, it will serve well for our current development, as the critical test for our tracing algorithm will come when a very large number of CPUs are available to receive tasks from each record, a number that on average and across many records should be expected to remain relatively constant. In such scheduling-critical cases, having some CPUs jump from record to record will not appreciably change what we are going to discuss. So, let us now focus on each of the three stages of tracing.

Regarding the hunt-up stage and assuming we have M CPUs to apply, the obvious method would be to assign each to a single run in the hunt-up sequence, always facing the possibility that one of these M 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, only perhaps a desired upper limit on how many runs one might want to expend to trace a single record. Still,

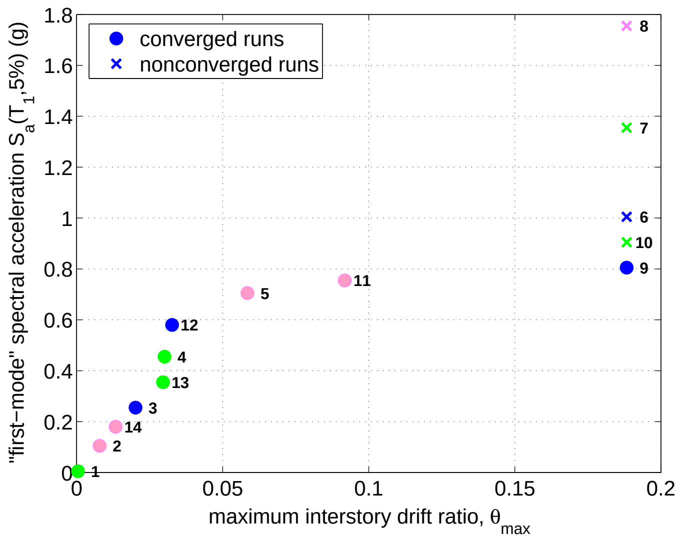


Figure 5: Tracing with three CPUs. Each of the three colors indicates a different CPU.

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

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	$0.105+0.10+1\times0.05$	0.255	2.02%
4	cpu3	$0.255+0.10+2\times0.05$	0.455	3.01%
5	cpu2	$0.455+0.10+3\times0.05$	0.705	5.85%
6	cpu1	$0.705+0.10+4\times0.05$	1.005	$+\infty$
7	cpu3	$1.005+0.10+5\times0.05$	1.355	$+\infty$
8	cpu2	$1.355+0.10+6\times0.05$	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%

due to the quadratically increasing steps in the hunt-up phase, it makes sense that the more CPUs we assign here and the more runs that have already been performed, the more we are likely to waste. This is an issue that will greatly impact the scheduling of slave-2's as we will discuss in the next section.

If on the other hand we have M CPUs to apply to the bracketing stage, an efficient way to do this would be to use as many of them as needed to achieve the desired accuracy and properly divide the space between the highest converged (C) run at level IM_{C} and the lowest non-converged (NC) run at IM_{NC}. This can be achieved by iteratively dividing the largest gap between the IM-levels of runs generated within the interval $[IM_{\mathrm{C}}, IM_{\mathrm{NC}}]$ by appropriately placing a run at the 1/3 distance point from the C level. When the distance between the successive IM_{C} and IM_{NC} points, as updated with each completed run, becomes lower than the required minimum dictated by the user-specified resolution, the bracketing stops. This has the excellent advantage that for $M=1$ CPU it conforms to the standard behavior of the serial hunt&fill algorithm, thus achieving excellent efficiency. For higher numbers of CPUs it is obviously prone to some higher percentage of wasted runs, still its performance remains quite acceptable.

The important question here is what is the maximum number of CPUs that can be applied to such a procedure for a given tolerance; any CPU beyond that will be clearly a waste. If we have defined the capacity resolution [23] by requesting that $(IM_{\mathrm{NC}} - IM_{\mathrm{C}}) \le a \cdot IM_{\mathrm{C}}$, where a is a reasonable percentage, e.g., 10%, of the level of the highest converged run, then there is an obvious limit to how many CPUs we should be willing to commit to such a scheme. It is easy to show that the worst case scenario happens when the flatline is very close to IM_{NC}, since successive divisions of the largest remaining segment in a 1/3-2/3 division will always cluster the larger 2/3-length segments in that direction. Correspondingly, the best scenario is for the flatline to be very close to IM_{C}. The number of runs that are going to be expended in these situations are:

$$i_{\text{best}} = \text{ceil}\left[\frac{\ln(IM_{\mathrm{NC}} - IM_{\mathrm{C}}) - \ln(a \cdot IM_{\mathrm{C}})}{\ln(3)}\right] \quad (1)$$

$$i_{\text{worst}} = \text{ceil}\left[\frac{\ln\left[(IM_{\mathrm{NC}} - IM_{\mathrm{C}})(a+1)\right] - \ln(a \cdot IM_{\mathrm{NC}})}{\ln(3/2)}\right], \quad (2)$$

where ceil(x) is the ceiling function that rounds any positive real x to its immediately higher integer. Obviously, we should not let the number of runs posted at any instant during bracketing exceed i_{worst}, or, conservatively thinking, not even i_{best}. As IM_{C} and IM_{NC} get closer with each run completed, i_{best} is updated (i.e., lowered) and the whole string of posted tasks is reevaluated. Whenever a CPU finishes a bracketing-run, the slave-1 should re-initialize the bracketing phase, removing all affected tasks that are still in the work-pool and assigning new ones to take advantage of the new positions of IM_{C} and IM_{NC}. Any excessive CPUs that cannot find a bracketing run due to the i_{best} limitation should simply be reassigned to filling-in below the C run. Thus, efficient bracketing is often intertwined with synchronous filling-in.

If, finally, we reach the fill-in stage, the sequence of runs is completely predictable, even in the rare case of a structural resurrection, i.e., a structural collapse occurring for a lower IM than the highest converged IM we obtained during hunt-up [1]. Thus, we can assign CPUs to fill-in up to the maximum number of runs we want to expend and still have no waste. We should be careful, though, with the mechanics of run-counting. We typically only want to allow a specified maximum number of runs [23] for the tracing record, thus assuring consistent treatment of all records. The problem is that if we count all runs performed by the spare CPUs during hunt-up or bracketing, there are going to be a lot of non-converged runs (especially in hunt-up) or many extra closely-spaced convergent runs in bracketing. If we sum all these, we may easily exceed our total allowance and still have a poorly-traced IDA, without any runs to spare for filling-in at the lower IMs. The only way to ensure consistent quality amongst all records, regardless of the number of CPUs available, is to count runs as if a standard single-CPU tracing was performed on each and every one of them. The close correspondence of the proposed algorithm with its serial version make this a trivial calculation.

It should be noted that since we never really know the number M of CPUs that will assign themselves to perform runs for this record, we can roughly approximate it by the number N_{idle} of CPUs that are idling at the moment plus the present slave-1 that is running the tracing. Conservatively, to make sure that other CPUs finishing their current tasks will have ample jobs to choose from, we will multiply it by a factor of, say, two. Thus, the resulting algorithm, meant to be run by a slave-1 processor, is designed to cater to multiple slave-2 CPUs by continually posting tasks while at the same time dynamically modifying its behavior by aggregating the results as they come back. In addition, it performs runs as a slave-2 itself instead of just waiting. The final parallel hunt&fill script is thus quite more complex than its serial counterpart:

Algorithm. PARALLEL_HUNTFILL

```
1  // hunt-up
2  repeat
3      for $i = 1$ to $2(N_{\text{idle}} + 1)$ do
4          increase IM by the step
5          post job in common memory
6          increase the step
7      end for
8      Pick a posted IM, run analysis, extract EDP(s)
9      Read EDP results from runs sent by other CPUs
10 until collapse is reached
11 delete posted jobs remaining
12 // bracket
13 repeat
14     $i_{\text{best}} \leftarrow \text{ceil}\{\ln[(IM_{\text{NC}} - IM_{\text{C}})/(a \cdot IM_{\text{C}})]/\ln(3)\}$
15     for $i = 1$ to $\min(2(N_{\text{idle}} + 1), i_{\text{best}})$ do
16         select IM that divides into 1/3-2/3 the largest gap in sorted
           list of posted IMs between $IM_{\text{C}}$ and $IM_{\text{NC}}$ inclusive
17         post job in common memory
18     end for
19     if $2(N + 1) > i_{\text{best}}$ then
20         while BracketJobsRunOrPosted < MaxRuns and largest gap
           in converged IMs < tolerance do
21             select IM halving largest gap between converged IMs
22             post job in common memory
23         end while
24     end if
25     Pick a posted IM, run analysis, extract EDP(s)
26     Read EDP results from runs sent by other CPUs
27     Delete all posted jobs
28 until gap between $IM_{\text{C}}$ and $IM_{\text{NC}}$ < tolerance
29 // fill-in
30 repeat
31     select IM halving largest gap between converged IMs
32     post job in common memory
33 until largest gap in converged IMs < tolerance
34 repeat
35     Pick a posted IM, run analysis, extract EDP(s)
36 until no jobs are left
```

An example of its application, using 3 CPUs for the same building and record as before, appears on Table 2 and Figure 5. Cpu1 was the designated coordinator, running as slave-1 in this case. Thus it only managed 4 analyses, while cpu2 and cpu3 run 5 each. Two runs were wasted during hunt-up by cpu2 and cpu3 as a lower run by cpu1 was found to be non-convergent. The rest of the phases happened to proceed in much the same way as for the serial hunt&fill. Thus, the final result was a total of 14 runs versus just 12 for the serial version, a waste percentage of $2/14 \approx 14.3\%$, a rather typical value for this setting. On the other hand, the total parallel tracing only lasted the time it takes to complete 5 runs, versus 12 for the serial case.

5.4. Self-scheduling via task prioritization

After the slave-1's have made sure that the work-pool contains an adequate number of tasks, it is the work of the slave-2's to sort them out and properly pick runs to perform. While in the previous section we based our discussion on a static model, considering only a single record with a given number of CPUs running analyses until its tracing is finished, our scheme becomes much more efficient if CPUs are allowed to reassign themselves across different records. This allows our algorithm to achieve proper load-balancing and minimize the amount of wasted runs. It can be efficiently implemented by appropriate prioritization of the tasks according to their chance of finding non-convergence. Hence, quantifying this chance for each task is fundamental and the best way to approach it is to consider each tracing phase separately.

Clearly, in the hunt-up stage it is not possible to predict where the next non-convergent run will be. Therefore, if we have more than one CPU performing hunt-up runs on the same record, there is a good chance that we might be wasting runs: One CPU may register an earlier collapse, thus invalidating all runs that are still being performed at higher IMs. Actually, there is no obvious limit to the number of CPUs one could apply to this phase, thus the potential waste can be enormous and reach up to the maximum number of runs allocated for a single record. Finding the flatline during bracketing is also an unpredictable process but quite less so compared to hunting-up, since we have an upper and a lower limit that we want to refine. While there is strong potential for many non-convergent runs to appear in this stage, this is valid for both the serial and the parallel tracing algorithms. In addition, there is an obvious limit to the total number of CPUs that can be assigned to bracketing as indicated by i_{best} and i_{worst}. This is why tasks in this stage are less prone to going to waste, although, again, the more CPUs that are performing bracketing runs simultaneously, the more we are likely to waste. Finally, if we are at the fill-in stage, the sequence of runs is completely predictable and practically always convergent: We can assign CPUs to fill-in up to the maximum number of runs we want to expend without any loss.

The priorities of the algorithm are now clear. First, it has to ensure that there is at least one CPU working on each available record to ensure that all records, even the ones currently in an "undesirable" phase, are proceeding smoothly. Otherwise, CPUs would be flocking to the one or two records that happen to be in the fill-in stage and provide guaranteed-to-converge runs, thus hurting their chances later on when they have to turn en masse to records in hunt-up. A simple way to ensure this is to have each slave-1 perform runs in its free time only for the record it currently traces. Then our scheme should channel any spare CPUs to perform runs for records that are in the fill-in phase. When no more fill-in tasks exist, CPUs should move to bracketing jobs and lastly to hunt-up. For these last two stages, preference should be given to jobs coming from records where fewer CPUs are already working. In addition, specifically for hunt-up, we should also take into account the number of runs that have already been performed, as it is an important indicator that the next run is closer to the flatline.

The proposed prioritization scheme cannot be implemented centrally in a simple and efficient way. This is why we propose to achieve it by self-scheduling, letting each slave-2 select jobs by itself using an appropriate scoring function to grade each task. Thus, for a given record i, let N^i_{CPU} be the number of CPUs currently working on it and let j denote the serial number assigned to the task (single-run) under question, according to its position in the sequence of runs as they have been posted by the tracing algorithm for record i. Then, the corresponding score S_{ij} can be estimated as:

$$S_{ij} = f(\text{stage}) + 5 \cdot N^i_{\text{CPU}} + g(j, \text{stage}) \tag{3}$$

where

$$f(\text{stage}) = \begin{cases} 10000, & \text{if stage = hunt-up} \\ 1000, & \text{if stage = bracket} \\ 0, & \text{if stage = fill-in} \end{cases} \tag{4}$$

$$g(j, \text{stage}) = \begin{cases} j, & \text{if stage = hunt-up} \\ 0, & \text{otherwise} \end{cases} \tag{5}$$

The tasks with the lowest score are deemed the least probable to prove non-convergent and thus receive the highest priority. In Eqs (3)–(5), the scoring weights have been selected to always favor filling-in over bracketing and bracketing over hunting-up and they have performed well for a variety of settings tested. Still, different strategies can be implemented by appropriately modifying the proposed constants.

The proposed technique guarantees excellent performance and a very low waste of runs (due to non-convergence) by directly minimizing the number of CPUs that work at the same time in jobs that have a high risk of collapse. The only minor problem is that it requires each CPU to update other CPUs on its status. In other words, each CPU needs to be informed of the case and record that all other CPUs are working on. This minimally raises the communication overhead but it is indeed a detail that we should keep in mind, especially when working with very simple systems (e.g., oscillators) where such delays can make a difference.

In order to apply the proposed single-run task partitioning, the master script does not have to change. It remains the same as for multiple IDA studies. The slave script though has to change dramatically. As 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, wasting no runs at all.

Algorithm. SLAVESCRIPT_MULTICASES2

```
 1 while jobs exist do
 2   if single-record job exists then
 3     Pick first available job (record)
 4     Run parallel hunt&fill on single record & postprocess
 5     Send results to common memory
 6   else
 7     Score posted single-run tasks, prioritizing fill-bracket-hunt
 8     Pick lowest-score run
 9     Run single dynamic analysis & postprocess
10     Send results to common memory
11   end if
12   If master node has called this procedure EXIT
13 end while
```

5.5. Further thoughts and improvements

Building upon the basic script developed, we can introduce further improvements that will enhance load-balancing and increase the speed of execution. These may complicate the details of the algorithms but they do not change the overall concepts presented.

First of all, as previously hinted, enormous advantage is to be gained by allowing some degree of interaction between the slave-1 CPU running the IDA tracing and the slave-2 CPUs performing the actual runs. When during the critical hunt-up and bracketing phases of the hunt&fill there are multiple CPUs working on the same record, then if any of them hits a non-convergence, this is to be immediately communicated to the slave-1 and the tracing strategy should be modified accordingly. In order to achieve that, each slave-1 should spawn a slave-2 process to run single analyses on the same CPU, while the parent process keeps checking the progress of all slave-2 nodes that are helping with the given record. If any of them hits a non-convergent IM, the parent process immediately deletes all jobs posted for higher IMs and it also sends a signal to the corresponding slave-2 CPUs (including its own spawned process) to terminate processing and start fresh with a new run. When single convergent dynamic runs may easily cost 30–90min of time, non-convergent ones may take substantially less time to complete, as numerical problems often happen in the middle of the record where the highest acceleration spikes occur. Therefore, the immediate exploitation of information resulting from non-convergence allows a rapid redeployment of computer resources, thus enhancing efficiency to the fullest. Specifically, for our example in Table 2, while it shows two runs wasted during the hunting phase, both of them were actually stopped before running their full length, thus considerably reducing the actual CPU time wasted.

Another variation that can result to a lower number of runs being wasted is based on the concept of performing fill-in even as we are hunting-up. While typically the fill-in phase corresponds to 20–30% of the total number of runs in any economically-traced IDA curve, when attempting to achieve excellent precision, something that is usually reserved for single-degree-of-freedom systems only, this percentage may rise to more than 60%. In such cases, while hunting-up with one processor, we can assign extra processors to fill in between the large hunt-up steps. This idea is only viable for high ratios of fill-in runs since the largest gaps are always left close to the flatline. Therefore, at least initially, we are going to be filling the smaller gaps that might need no filling at all. Of course we will not have wasted a single run in this way but the extra runs performed will probably not help to speed up the tracing if economy in runs is what we are after, having only improved resolution in the body of the IDA curve rather than close to collapse. Therefore this approach can only have a limited field of application for smaller models and high accuracy settings, but if these are what we are after it can help significantly.

6. Performance comparison

In order to test the algorithms presented we used the nine story building discussed earlier, subjected to 20 ground motion records [23]. Each dynamic analysis takes 8–15min on a Pentium IV CPU, the longer times usually needed for convergent runs close to collapse. We run this multi-record IDA study using (a) one CPU with the serial hunt&fill algorithm versus 2–26 identical CPUs with (b) the serial hunt&fill algorithm on single-record tasks, (c) the parallel hunt&fill algorithm with single-analysis tasks and (d) the stepping algorithm with single-analysis tasks. Serving as the basis for our comparison, the one-CPU case finished in approximately 40hrs. To better understand the capabilities of each algorithm, let us first compare this case in detail against using three CPUs only, before expanding to more.

When using the three CPUs with single-record tasks, the serial tracing algorithm managed to complete in 14hrs only. Due to the number of CPUs not being a divisor of the number of records, there cannot be a perfect load-balancing when tracing the last two records. Then, as seen in Table 3, cpu1 will remain idle while cpu2 and cpu3 keep working. Since we have many more records than CPUs, this is not a huge waste, simply under-using one CPU. Therefore we achieve a performance improvement, termed speedup [21], of $40/14 = 2.86$ over the single CPU case, resulting to a slightly sublinear efficiency of $2.86/3 = 95\%$.

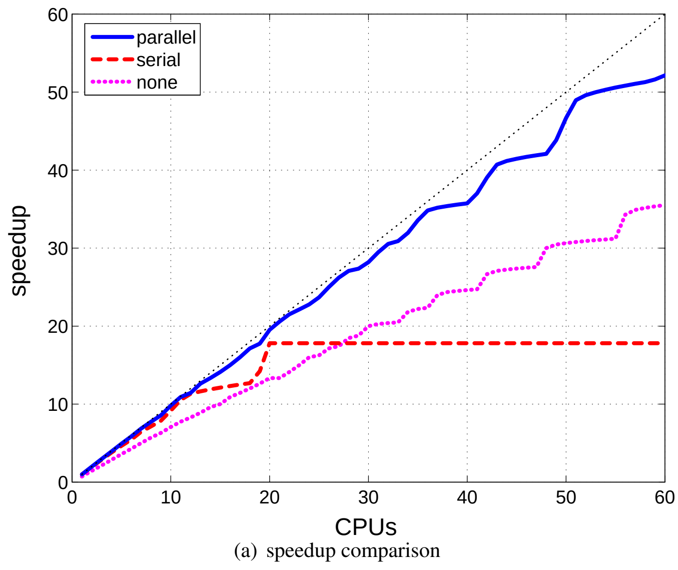
When using the parallel version of hunt&fill with single-analysis tasks, the analysis will proceed in much the same way as in the previous case until it hits the final two records. Then, the previously idle cpu1 will alternate between helping cpu2 and cpu3 with their workload. Therefore, the wasted runs will be less than what we observed in Table 2, where 3 CPUs were tracing a single record. Then, two runs were wasted for a single record, while now only one is wasted on average per record, resulting to 13 runs for each of the final two records. The speedup thus rises to 2.96 for 3 CPUs, for an almost 99% efficiency.

Table 3: Detailed performance comparison of stepping, serial IDA tracing and parallel tracing for a nine-story structure subjected to 20 records.

Tracing	CPUs	CPU loads	time (hrs)	speedup
serial	1	20x12 = 240runs cpu1: 20recs	40	1
serial	3	20x12 = 240runs cpu1: 6recs cpu2: 7recs cpu3: 7recs	14	2.86
parallel	3	20x12 + 2 = 242runs (2 wasted) cpu1: 6recs + 8runs cpu2: 6recs + 8runs cpu3: 6recs + 8runs	13.5	2.96
none	3	336 runs (2 wasted) cpu1: 111runs cpu2: 113runs cpu3: 112runs	18.7	2.14

To offer a different standard for comparison, we have also tested the simple stepping algorithm on three CPUs, marked as “None” (i.e., no tracing) in Table 3. This is essentially what can be applied with minimal programming through any platform supporting parallelization, e.g., OpenSees [25], via specifying a constant IM-step. We assume that at least some termination clause is provided to stop further runs at increased IMs for a given record when the first non-convergence is observed. Assuming that we have some knowledge of the expected flatline $S_a(T_1, 5\%)$-values, we have requested a minimum of 8 convergent runs per record. The reader is reminded here that the tracing algorithms manage 10–11 convergent runs everywhere, i.e., 25–30% higher quality in IDA curve representation. Consequently at least a 0.06g IM-step has to be used and the result is a staggering number of 334 runs. If we increase our programming overhead by at least using some clever task-assignment, i.e., by letting all CPUs run the lowest IM values across all records before proceeding with the highest ones, we can minimize the waste to only two runs for a 336 total. Still, the speedup drops to 2.14 for an efficiency rating of 71%. In other words, despite our having to settle for lower quality tracing, almost 30% of the supplied processing power was misused rather than just 1% with the parallel hunt&fill.

Fig. 6 shows the results for all the tests run on 2–26 CPUs, expanded by performing simulations on fitted process-models for up to 60 processors, showing the expected behavior of each algorithm. In general, for low numbers of CPUs per record, the serial and parallel hunt&fill will almost behave identically, with the latter having a slight edge in cases where the number of CPUs is not a divisor of the number of records, as explained earlier. They both maintain near-linear speedup until a total of 10–11 CPUs are used (Fig. 6(a)). In the same range, the stepping algorithm will achieve a nearly constant 70% efficiency (Fig. 6(b)). While one CPU per record, or twenty total, is



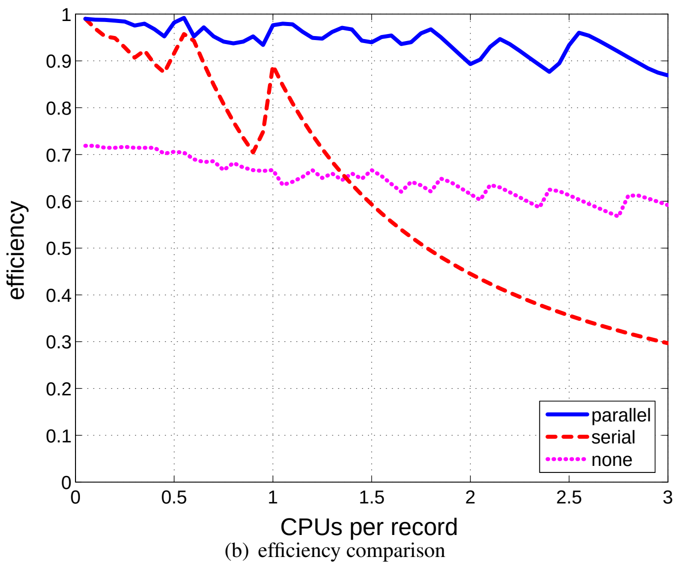


Figure 6: Multi-CPU performance comparison of the stepping, serial and parallel tracing algorithms for a nine-story structure subjected to 20 records. Data beyond 26 CPUs is based on simulations.

the limit of application for the serial tracing, 0.5–0.6 CPUs per record is actually a more reasonable practical limit as any additional processors will in general be kept idling without a record to process for the last cycle of task assignments. In such cases, the efficiency of serial hunt&fill will drop almost to the level of the stepping algorithm, up to when 20 processors are employed. That is where for an 89% efficiency the serial hunt&fill will manage its best processing time of about 2.3hrs, i.e., the time it takes to trace the worst (slowest to run) record with one CPU. The same case will be performed slightly faster by the parallel hunt&fill and single-analysis tasks. The reason is that the worst record will eventually be traced with the help of one or two more CPUs that have finished their own, easier cases. Thus, the processing time drops to less than 2.1hrs, indicating an efficiency of 97% compared to only 67% for the stepping algorithm.

If we use more than 20 CPUs, i.e., more than one per record, the serial tracing algorithm is no longer applicable. On the other hand, both the parallel hunt&fill and the stepping algorithm will roughly maintain their previous performance taking advantage of the finer partitioning of tasks that they are based on. In all tests, though, the parallel hunt&fill outperforms stepping by a wide margin of nearly 30% additional efficiency. As shown in Fig. 6(a), it maintains near-linear speedup, achieving efficiencies in the order of 90% or better in most situations (Fig. 6(b)).

Although this test only involved a single structure, the results can be used as indicative for other typical situations. The reason is that the number of CPUs per record has been found to be a reliable predictor of efficiency. For example, if we were to use 60 CPUs, i.e., 3 per record, without dynamic CPU self-scheduling, then we can conservatively approximate the algorithm's performance by using the case presented in Table 2 times twenty. 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 a speedup of 48 for 60 CPUs, or an efficiency of 48/60=80%. Actually, simulations of the actual process, where self-scheduling for slave-2 processors and early termination of wasted runs is implemented, show that the efficiency is slightly better, reaching 87%, as shown in Fig. 6(b). Of course, results will vary in each case, depending on the tracing settings and on the number of runs allowed per record, but the tests presented can still serve as a rough guide for typical MDOF IDA studies.

In any case, if the number of CPUs is further increased beyond three per record, the expected drop in efficiency will only worsen. Nevertheless, the proposed parallel tracing algorithm remains our best choice, if not the only one, before resulting to parallel finite element solution algorithms.

7. Conclusions

Two efficient algorithms to perform Incremental Dynamic Analysis using multiple CPUs in parallel have been presented and tested. The first represents a coarse-grained partitioning of tasks on the basis of single records and, while simple to program and implement, it is applicable only when the number of CPUs is equal to or less than the number of individual single-record IDA studies that will be performed. The second algorithm further partitions the computations by working at the level of single dynamic analyses, implementing a far more efficient, medium-grained scheme that necessitates some careful programming to be realized.

In this case, due to the unpredictable nature of IDA curves caused by the global dynamic instability of realistic structural models, the intensity level of the dynamic runs has to be decided on the fly. Therefore, to achieve efficient parallelization, we have exploited the idiosyncracies of IDA to offer a parallel version of the standard hunt&fill serial tracing algorithm. The end product is a dynamic load-balancing algorithm with a three-level hierarchy of master/slave-1/slave-2 roles for CPUs using dynamic task generation and self-scheduling to efficiently parallelize all tasks. Using a minimum of communication among the processes, it allows for rapid reassignment of CPUs based on currently available information to modify their behavior. It can achieve near-optimal load balancing that is scalable to a relatively large array of processors, the actual number depending on the size of the problem. Testing on a realistic analysis setting has shown that the parallel tracing algorithm with single-run task partitioning can thus achieve near-linear efficiency for high numbers of processors, making it suitable for all but the largest clusters available in typical engineering offices or research facilities.

8. Acknowledgments

The author wishes to dedicate this paper to the memory of Professor I. Vardoulakis, who first taught him that science is fun.

References

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

[2] Lee K, Foutch DA. Seismic performance evaluation of pre-northridge steel frame buildings with brittle connections. ASCE Journal of Structural Engineering 2002;128(4):546–55.

[3] Lee K, Foutch DA. Performance evaluation of new steel frame buildings for seismic loads. Earthquake Engineering and Structural Dynamics 2002;31(3):653–70.

[4] Yun SY, Hamburger RO, Cornell CA, Foutch DA. Seismic performance evaluation for steel moment frames. ASCE Journal of Structural Engineering 2002;128(4):534–45.

[5] Liao KW, Wen YK, Foutch DA. Evaluation of 3D steel moment frames under earthquake excitations. i: Modeling. ASCE Journal of Structural Engineering 2007;133(3):462–70.

[6] Tagawa H, MacRae G, Lowes L. Probabilistic evaluation of seismic performance of 3-story 3D one- and two-way steel moment-frame structures. Earthquake Engineering and Structural Dynamics 2008;37(5):681–96.

[7] Pinho R, Casarotti C, Antoniou S. A comparison of single-run pushover analysis techniques for seismic assessment of bridges. Earthquake Engineering and Structural Dynamics 2007;36(10):1347–62.

[8] Goulet CA, Haselton CB, Mitrani-Reiser J, Beck JL, Deierlein GG, Porter KA, et al. Evaluation of the seismic performance of a code-conforming reinforced-concrete frame building—from seismic hazard to collapse safety and economic losses. Earthquake Engineering and Structural Dynamics 2007;36(13):1973–97.

[9] Ibarra LF. Global collapse of frame structures under seismic excitations. PhD Dissertation; Department of Civil and Environmental Engineering, Stanford University; Stanford, CA; 2003.

[10] Vamvatsikos D, Cornell CA. Direct estimation of the seismic demand and capacity of oscillators with multi-linear static pushovers through incremental dynamic analysis. Earthquake Engineering and Structural Dynamics 2006;35(9):1097–117.

[11] ATC . Effects of strength and stiffness degradation on seismic response. Report No. FEMA-P440A; prepared for the Federal Emergency Management Agency; Washington, DC; 2008.

[12] Haselton CB. Assessing seismic collapse safety of modern reinforced concrete frame buildings. PhD Dissertation; Department of Civil and Environmental Engineering, Stanford University; Stanford, CA; 2006.

[13] Vamvatsikos D, Papadimitriou C. Optimal multi-objective design of a highway bridge under seismic loading through incremental dynamic analysis. In: International Conferenec on Structural Safety and Reliability, ICOSSAR 2005. Rome, Italy; 2005, p. 329–36.

[14] Liel AB, Haselton CB, Deierlein GG, Baker JW. Incorporating modeling uncertainties in the assessment of seismic collapse risk of buildings. Structural Safety 2009;31(2):197–211.

[15] Dolsek M. Incremental dynamic analysis with consideration of modelling uncertainties. Earthquake Engineering and Structural Dynamics 2009;38(6):805–25.

[16] Vamvatsikos D, Fragiadakis M. Incremental dynamic analysis for estimating seismic performance sensitivity and uncertainty. Earthquake Engineering and Structural Dynamics 2010;39(2):141–63.

[17] Vamvatsikos D, Cornell CA. Developing effcient scalar and vector intensity measures for IDA capacity estimation by incorporating elastic spectral shape information. Earthquake Engineering and Structural Dynamics 2005;34(13):1573–600.

[18] Luco N, Cornell CA. Structure-specific, scalar intensity measures for near-source and ordinary earthquake ground motions. Earthquake Spectra 2007;23(3):357–92.

[19] Baker JW, Cornell CA. Vector-valued intensity measures incorporating spectral shape for prediction of structural response. Journal of Earthquake Engineering 2008;12(4):534–54.

[20] Azarbakht A, Dolsek M. Prediction of the median IDA curve by employing a limited number of ground motion records. Earthquake Engineering and Structural Dynamics 2007;36(15):2401–21.

[21] Grama A, Gupta A, Karypis G, Kumar V. Introduction to Parallel Computing. Boston, MA: Addison-Wesley; 2nd ed.; 2003.

[22] McKenna F, Fenves G, Jeremic B, Scott M. Open system for earthquake engineering simulation. 2000. [May 2008]; URL http://opensees.berkeley.edu.

[23] Vamvatsikos D, Cornell CA. Applied incremental dynamic analysis. Earthquake Spectra 2004;20(2):523–53.

[24] Cornell CA. Engineering seismic risk analysis. Bulletin of the Seismological Society of America 1968;58(5):1583–606.

[25] McKenna F, Fenves G. Using the OpenSees interpreter on parallel computers. NEESit Report No. TN-2007-16; NEES Cyberinfrastructure Center; La Jolla, CA; 2007.

[26] MPI Forum . MPI: A message passing interface standard (ver. 1.1). 2003. [Mar 2008]; URL http://www.mpi-forum.org.

[27] Foster I. Designing and Building Parallel Programs. Boston, MA: Addison-Wesley; 1995. URL http://www.mcs.anl.gov/~itf/dbpp/.

[28] Seismosoft . Seismostruct - a computer program for static and dynamic nonlinear analysis of framed structures. 2007. [May 2008]; URL http://www.seismosoft.com.