How to Split Data to Speed Up MERT

Lane Schwartz
Air Force Research Laboratory
Human Effectiveness Directorate
Wright-Patterson AFB, OH USA

Abstract

Each iteration of minimum error rate training involves re-translating a development set. Distributing this work across computational nodes can speed up translation time, but in practice some parts may take much longer to complete than others, leading to computational slack time. To address this problem, we develop three novel algorithms for distributing translation tasks in a parallel computing environment, drawing on research in parallel machine scheduling. We present results showing a substantial speedup in overall decoding time.

1 Introduction

The task of translation involves translating a source language document $\mathbf{f}$ into target language $\mathbf{e}$. Most popular statistical translation techniques select the best translation $\hat{e}$ for source sentence f according to a linear combination of models $\boldsymbol{\phi}$ using a set of model weights $\boldsymbol{\lambda}$ (Och and Ney, 2002).

$$\hat{e} = \arg\max_{e} \sum_{i} \lambda_i \phi_i(e, f) \quad (1)$$

Values for $\boldsymbol{\lambda}$ are obtained by optimizing an objective function such as BLEU (Papineni et al., 2001) against a development set, most commonly using minimum error rate training (MERT) (Och, 2003). Each iteration of MERT requires this development set to be re-translated using a new set of $\boldsymbol{\lambda}$ weights. MERT is one of the slowest components in a typical machine translation training pipeline, and translating the development set is nearly always the slowest step in MERT. We now examine techniques for speeding up MERT by distributing translation jobs across a cluster of parallel computational nodes.

Ideally, all parts should take the same amount of time to translate. While naive splitting techniques reduce the time required for each translation iteration by splitting the work between p computational nodes, in practice some parts may take much longer to complete than others. This can lead to significant computational slack time. To address this problem, we develop three novel algorithms for splitting translation tasks in a parallel computing environment, drawing on research in parallel machine scheduling.

2 Related Work

While the models in Equation 1 could, in theory, condition on previously translated sentences, in practice virtually no widely used models do so. It is therefore very straightforward to split the data into p parts, and translate each part independently on p computational nodes. Scripts implemented in Moses (Koehn et al., 2007) do exactly that, simply splitting the data into p arbitrary parts such that each part contains the same number of lines (Algorithm 1).

Research into parallel machine scheduling problems constitutes a wide and well-studied field, ranging through various disciplines of engineering, manufacturing, and management in addition to computer science and applied mathematics (Cheng and Sin, 1999), spanning a wide range of scheduling techniques (Panwalkar and Iskander, 1977).

We now briefly examine the existing research most relevant to our task. Hu (1961) and Graham (1966; 1969) develop various list scheduling algorithms. This family of algorithms prioritizes jobs into a queue, then assigns jobs to machines in queue order. This approach attempts to evenly balance the

Algorithm 1 Split input text into n parts such that each part contains the same number of lines.

```
function NAIVE-SPLIT(n,input)
    ℓ ← n/ input.length
    for p ← 0 ... (n − 1) do
        i ← ℓ × p
        for j ← i ... (i + ℓ − 1) do
            if j < ℓ then
                output[p].append(input[j])
            else
                break
            end if
        end for
    end for
    return output
end function
```

load on each execution host (De and Morton, 1980; Cheng and Sin, 1999). Both Algorithm 1 and techniques we develop fall into this family of algorithms.

3 Better Splitting for Faster Results

To observe the effects of splitting algorithms on decoding speed, we translated Urdu-English data using Moses in a parallel computing cluster, distributing work using the Sun Grid Engine. We ran two decoding setups: a standard configuration using a 5-gram language model, and a much slower configuration that also used an incremental syntactic language model. Using Algorithm 1, the runtimes of the slowest of n translation jobs in each configuration is illustrated in Figure 1 for various values of n. Figure 2 shows that in all cases, there is a significant difference between the fastest and slowest translation job, leading to computational slack time (Figure 3).

In examining these results, we observe that the slack time results primarily from situations where some jobs are assigned a disproportionate number of short sentences, and thus finish much faster than jobs that are assigned many longer sentences. To remedy this imbalance, we propose Algorithm 2. This technique examines the word lengths of each sentence prior to splitting the data into jobs. Sentences are sorted according to length, then assigned in turns to jobs. This results in the sentence length histograms for each job being approximately equal. Figures 1a–3a fail to show improvment in speed for a standard Moses configuration for small values of n. But for values of $n > 8$, and for all values of n using the slow syntactic language model (Figures 1b–3b), Algorithm 2 results in significant decreases in total decoding time over the naive Algorithm 1.

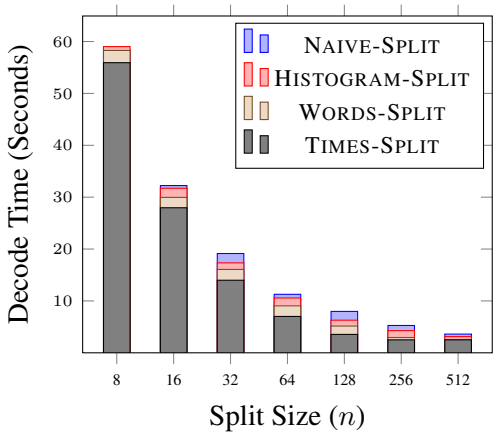
While Algorithm 2 balances short and long sentences across jobs, we may be able to improve runtimes by balancing the total number of words in each

Algorithm 2 Split input text into n parts to balance the histograms of line lengths for all parts.

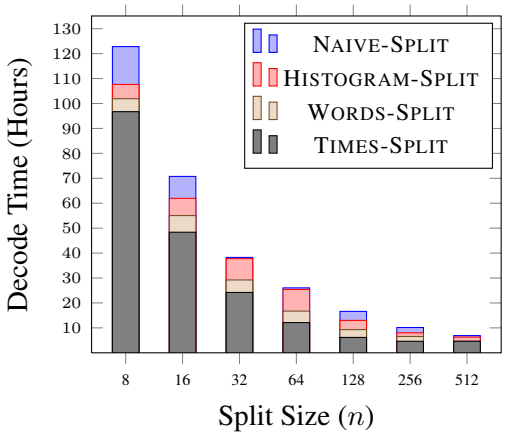
```
function HISTOGRAM-SPLIT(n,input)
    for i ← 0 ... ( input.length −1) do
        sentence[i].length ← input[i].length
        sentence[i].index ← i
    end for
    SORT(sentence) {|x, y| x.length ⇔ y.length}
                        ▷ Sort sentences by length
    p ← n
    for i ← 0 ... (input.length −1) do
        if p < n then
            p ← p + 1
        else
            p ← 0
        end if
        output[p].append(input[sentence[i].index])
    end for
    return output
end function
```

Algorithm 3 Split input text into n parts to balance the number of words for all parts.

```
function WORDS-SPLIT(n,input)
    for i ← 0 ... ( input.length −1) do
        sentence[i].length ← input[i].length
        sentence[i].index ← i
    end for
    SORT(sentence) {|x, y| y.length ⇔ x.length}
      ▷ Sort sentences by length, in reverse order
    for i ← 0 ... ( input.length −1) do
        p ← LEAST(words)
                ▷ Find partition with fewest words
        output[p].append(input[sentence[i].index])
        words[p] ←words[p] + sentence[i].length
    end for
    return output
end function
```



(a) Decoding times in **seconds** for standard decoder configured using a 5-gram language model.



(b) Decoding times in **hours** for decoder configured using a syntactic language model in addition to a 5-gram language model.

Figure 1: Decoding times for the slowest translation job in a translation task split into n decoding jobs using various splitting algorithms (NAIVE-SPLIT, HISTOGRAM-SPLIT, WORDS-SPLIT, and TIMES-SPLIT).

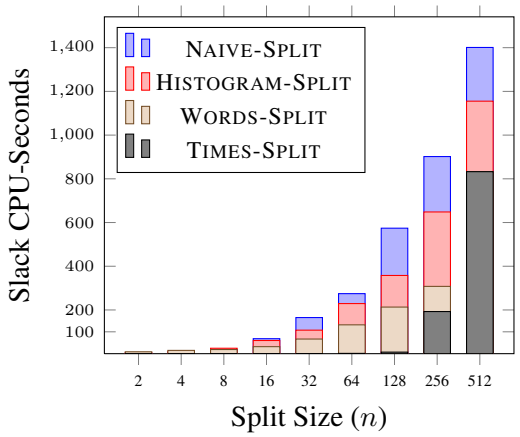
Split Size (n)	NAIVE-SPLIT		HISTOGRAM-SPLIT		WORDS-SPLIT		TIMES-SPLIT	
	Min	Max	Min	Max	Min	Max	Min	Max
2	222.9	224.4	221.5	225.8	219.3	228.0	*223.7*	***223.7***
4	109.2	113.7	110.0	114.6	108.7	115.6	*111.8*	***111.8***
8	51.4	58.4	52.2	59.0	53.2	58.3	*55.9*	***55.9***
16	24.9	32.2	25.0	31.7	25.3	30.0	27.9	**28.0**
32	11.3	19.1	11.9	17.3	11.7	16.1	*14.0*	***14.0***
64	5.4	11.3	5.4	10.6	5.7	9.1	*7.0*	***7.0***
128	1.3	8.0	2.2	6.3	2.3	5.2	*3.5*	***3.5***
256	0.3	5.3	0.7	4.3	0.8	3.0	1.7	**2.5**
512	0.0	3.6	0.2	3.1	0.3	**2.5**	0.6	**2.5**

(a) Decoding times in **seconds** for standard decoder configured using a 5-gram language model.

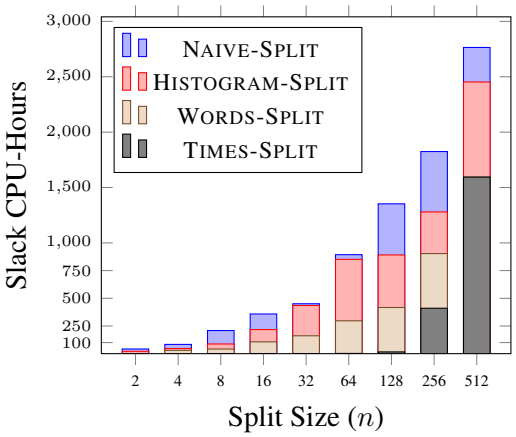
Split Size (n)	NAIVE-SPLIT		HISTOGRAM-SPLIT		WORDS-SPLIT		TIMES-SPLIT	
	Min	Max	Min	Max	Min	Max	Min	Max
2	365.7	408.0	376.3	397.5	386.1	387.6	*386.9*	***386.9***
4	176.1	214.4	186.8	205.0	184.6	200.9	*193.4*	***193.4***
8	84.6	122.8	84.8	107.6	88.4	101.9	*96.7*	***96.7***
16	40.8	70.8	40.5	62.0	45.3	55.0	*48.4*	***48.4***
32	19.2	38.3	18.8	37.8	20.5	29.2	*24.2*	***24.2***
64	9.2	26.1	9.2	25.4	9.4	16.7	*12.1*	***12.1***
128	2.7	16.6	4.1	13.0	3.7	9.3	5.9	**6.2**
256	0.7	10.2	1.3	8.0	1.4	6.6	2.9	**4.6**
512	0.0	6.9	0.3	6.3	0.6	**4.6**	1.1	**4.6**

(b) Decoding times in **hours** for decoder configured using a syntactic language model in addition to a 5-gram language model.

Figure 2: Decoding times for the fastest (min) and slowest (max) decoding jobs when a translation task is split into n decoding jobs. *Italics* indicate balanced task times. **Bold** indicates fastest max time at that split.



(a) Slack **CPU-Seconds** for standard decoder configured using a 5-gram language model.



(b) Slack **CPU-Hours** for decoder configured using a syntactic language model in addition to a 5-gram language model.

Figure 3: Cumulative slack CPU time for n processing cores when processing a parallel translation task split into n jobs using various splitting algorithms. Slack CPU time is caused when some jobs finish before others. Zero slack time indicates conditions where all jobs complete simultaneously.

Algorithm 4 Split input text into n parts to balance the estimated translation time of all parts.

```
function TIMES-SPLIT(n,input,estimate)
    for i ← 0 ... ( input.length −1) do
        sentence[i].time ← estimate[i]
        sentence[i].index ← i
    end for
    SORT(sentence) {|x, y| y.time ⇔ x.time}
        ▷ Sort sentences by time, in reverse order
    for i ← 0 ... ( input.length −1) do
        p ← LEAST(times)
                    ▷ Find partition with least time
        output[p].append(input[sentence[i].index])
        times[p] ← times[p] + sentence[i].time
    end for
    return output
end function
```

job. In Algorithm 3, sentences are sorted by length into a queue, with longest sentences at the head of the queue. Initially, no sentences have been assigned to any job. The longest sentence, at the head of the queue, is assigned first to a job. As each sentence is assigned to a job, the total number of words assigned to that job is recorded. Each subsequent sentence is removed from the queue and assigned to the job with the least work assigned to it, as measured by number of words. Results for Algorithm 3 show substantial speedups over Algorithms 1 and 2 when using the slower decoder configuration. Similar speedups are seen where $n > 8$ for the faster configuration.

When assigning sentences to jobs, we would ideally like to know how long each sentence will take to process. Algorithms 2 and 3 use the number of words in each sentence as a proxy for processing time. During MERT, the same set of development sentences are translated multiple times. Since each decoding process differs only by the $\boldsymbol{\lambda}$ weights used, it is reasonable to expect similar runtimes for each run. With this in mind, we record the time required to translate each sentence during the first iteration of MERT. In subsequent iterations, Algorithm 4 uses the time recorded to translate a sentence as an estimate of the time it will take to translate that sentence again. Algorithm 4 differs from Algorithm 3 by sorting using these times instead of sentence length. This technique results in speedups under all conditions (Figure 2). In nearly all conditions, the speedup is substantial over the baseline. Slack time is at or near zero in all cases where $n < 256$.

4 Conclusion

We have incorporated Algorithms 3 and 4 into the Moses decoding scripts, enabling substantial speedups in parallel decoding time at no additional cost.

References

T.C.E. Cheng and C.C.S. Sin. 1999. A state-of-the-art review of parallel-machine scheduling research. *European Journal of Operational Research*, 47:271–292.

Prabuddha De and Thomas E. Morton. 1980. Scheduling to minimum makespan on unequal parallel processors. *Decision Sciences*, 11(4):586–602, October.

Ron L. Graham. 1966. Bounds on certain multiprocessing timing anomalies. *The Bell Systems Technical Journal*, 45(9):1563–1581, November.

Ron L. Graham. 1969. Bounds on certain multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics*, 17(2):416–429, March.

T.C. Hu. 1961. Parallel sequencing and assembly line problems. *Operations Research*, 9(6):841–848.

Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Chris Dyer, Ondřej Bojar, Alexandra Constantin, and Evan Herbst. 2007. Moses: Open source toolkit for statistical machine translation. In *Proc. ACL 2007 Demo and Poster Sessions*.

Franz Och and Hermann Ney. 2002. Discriminative training and maximum entropy models for statistical machine translation. In *Proc. ACL*.

Franz Och. 2003. Minimum error rate training in statistical machine translation. In *Proc. ACL*, pages 160–167, Sapporo, Japan, July.

S.S. Panwalkar and Wafik Iskander. 1977. A survey of scheduling rules. *Operations Research*, 25(1):45–61.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2001. BLEU: a method for automatic evaluation of machine translation. In *Proc. ACL*, pages 311–318.

Opinions, interpretations, conclusions, and recommendations are those of the authors and are not necessarily endorsed by the United States Air Force. Cleared for public release on 19 Jan 2011. Originator reference number RH-11-107802. Case number 88ABW-2011-0185.