OpenMP to parallelise LSTM|

Abishek N

CS22B1092

Showing only parallelized part of the code:

```
root@DESKTOP-8P9HTVN:/mnt/c/Users/LENOVO/Desktop/recurrent-neural-net-master/src# gcc -o main *.c -lm
root@DESKTOP-8P9HTVN:/mnt/c/Users/LENOVO/Desktop/recurrent-neural-net-master/src# ./main input.txt
```

```c
void vector_set_to_zero(double* V, int L) {
  #pragma omp parallel for
  for (int l = 0; l < L; l++) {
      V[l] = 0.0;
  }
}
```

```c
void vectors_add(double* A, double* B, int L) {
    #pragma omp parallel for
    for (int l = 0; l < L; l++) {
        A[l] += B[l];
    }
}
```

```
void vectors_multiply(double* A, double* B, int L) {
    #pragma omp parallel for
    for (int l = 0; l < L; l++) {
        A[l] *= B[l];
    }
}
```

```
void copy_vector(double* A, double* B, int L) {
    #pragma omp parallel for
    for (int l = 0; l < L; l++) {
        A[l] = B[l];
    }
}
```

```
void vector_sqrt(double* A, int L) {
    #pragma omp parallel for simd
    for (int l = 0; l < L; l++) {
        A[l] = sqrt(A[l]);
    }
}
```

```
void vectors_multiply_scalar(double* A, double b, int L) {
    #pragma omp parallel for simd
    for (int l = 0; l < L; l++) {
        A[l] *= b;
    }
}
```

```c
void fully_connected_forward(double* Y, double* A, double* X, double* b, int R, i
    #pragma omp parallel for
    for (int i = 0; i < R; i++) {
        double sum = b[i];  // Initialize with bias
        for (int n = 0; n < C; n++) {
            sum += A[i * C + n] * X[n];
        }
        Y[i] = sum;  // Store the result
    }
}
```

```c
void fully_connected_backward(double* dldY, double* A, double* X, double* dldA,
                              double* dldX, double* dldb, int R, int C) {
    // Computing dldA
    #pragma omp parallel for collapse(2)
    for (int i = 0; i < R; i++) {
        for (int n = 0; n < C; n++) {
            dldA[i * C + n] = dldY[i] * X[n];
        }
    }

    // Computing dldb
    #pragma omp parallel for
    for (int i = 0; i < R; i++) {
        dldb[i] = dldY[i];
    }

    // Computing dldX
    #pragma omp parallel for
    for (int i = 0; i < C; i++) {
        double sum = 0.0;
        for (int n = 0; n < R; n++) {
            sum += A[n * C + i] * dldY[n];
        }
        dldX[i] = sum;
    }
}
```

```c
void sigmoid_forward(double* Y, double* X, int L) {
    #pragma omp parallel for
    for (int l = 0; l < L; l++) {
        Y[l] = 1.0 / (1.0 + exp(-X[l]));
    }
}

void sigmoid_backward(double* dldY, double* Y, double* dldX, int L) {
    #pragma omp parallel for
    for (int l = 0; l < L; l++) {
        dldX[l] = (1.0 - Y[l]) * Y[l] * dldY[l];
    }
}

void tanh_forward(double* Y, double* X, int L) {
    #pragma omp parallel for
    for (int l = 0; l < L; l++) {
        Y[l] = tanh(X[l]);
    }
}

void tanh_backward(double* dldY, double* Y, double* dldX, int L) {
    #pragma omp parallel for
    for (int l = 0; l < L; l++) {
        dldX[l] = (1.0 - Y[l] * Y[l]) * dldY[l];
    }
}
```

```c
void softmax_layers_forward(double* P, double* Y, int F, double temperature) {
    double sum = 0.0;

#ifdef WINDOWS
    double *cache = malloc(sizeof(double) * F);
    if (cache == NULL) {
        fprintf(stderr, "%s.%s.%d malloc(%zu) failed\r\n",
                __FILE__, __func__, __LINE__, sizeof(double) * F);
        exit(1);
    }
#else
    double cache[F]; // Stack allocation for non-Windows compilers
#endif

    // Compute exponentials in parallel
    #pragma omp parallel for reduction(+:sum)
    for (int f = 0; f < F; f++) {
        cache[f] = exp(Y[f] / temperature);
        sum += cache[f];
    }

    // Normalize in parallel
    #pragma omp parallel for
    for (int f = 0; f < F; f++) {
        P[f] = cache[f] / sum;
    }

#ifdef WINDOWS
    free(cache);
#endif
}
```

```c
void lstm_forward_propagate(lstm_model_t* model, double *input,
  lstm_values_cache_t* cache_in, lstm_values_cache_t* cache_out,
  int softmax)
{
  int N, Y, S, i = 0;
  double *h_old, *c_old, *X_one_hot;

  h_old = cache_in->h;
  c_old = cache_in->c;

  N = model->N;
  Y = model->Y;
  S = model->S;

#ifdef WINDOWS
  double *tmp;
  if (init_zero_vector(&tmp, N)) {
    fprintf(stderr, "%s.%s.%d init_zero_vector(.., %d) failed\r\n",
      __FILE__, __func__, __LINE__, N);
    exit(1);
  }
#else
  double tmp[N];
#endif

  copy_vector(cache_out->h_old, h_old, N);
  copy_vector(cache_out->c_old, c_old, N);

  X_one_hot = cache_out->X;

  // Parallelizing input one-hot encoding
  #pragma omp parallel for
  for (i = 0; i < S; i++) {
    X_one_hot[i] = (i < N) ? h_old[i] : input[i - N];
  }

  // Fully connected + activation layers (parallelized)
  fully_connected_forward(cache_out->hf, model->Wf, X_one_hot, model->bf, N, S);
  fully_connected_forward(cache_out->hi, model->Wi, X_one_hot, model->bi, N, S);
  fully_connected_forward(cache_out->ho, model->Wo, X_one_hot, model->bo, N, S);
  fully_connected_forward(cache_out->hc, model->Wc, X_one_hot, model->bc, N, S);

  #pragma omp parallel sections
  {
    #pragma omp section
    sigmoid_forward(cache_out->hf, cache_out->hf, N);

    #pragma omp section
    sigmoid_forward(cache_out->hi, cache_out->hi, N);

    #pragma omp section
    sigmoid_forward(cache_out->ho, cache_out->ho, N);

    #pragma omp section
```

```c
    #pragma omp section
    tanh_forward(cache_out->hc, cache_out->hc, N);
  }

  // Compute cell state: c = hf * c_old + hi * hc
  copy_vector(cache_out->c, cache_out->hf, N);
  vectors_multiply(cache_out->c, c_old, N);

  copy_vector(tmp, cache_out->hi, N);
  vectors_multiply(tmp, cache_out->hc, N);

  vectors_add(cache_out->c, tmp, N);

  // Compute hidden state: h = ho * tanh_c_cache
  tanh_forward(cache_out->tanh_c_cache, cache_out->c, N);
  copy_vector(cache_out->h, cache_out->ho, N);
  vectors_multiply(cache_out->h, cache_out->tanh_c_cache, N);

  // Compute softmax or alternative activations
  fully_connected_forward(cache_out->probs, model->Wy, cache_out->h, model->by, Y, N);

  if (softmax > 0) {
    softmax_layers_forward(cache_out->probs, cache_out->probs, Y, model->params->softmax_temp);
  }

#ifdef INTERLAYER_SIGMOID_ACTIVATION
  if (softmax <= 0) {
    sigmoid_forward(cache_out->probs, cache_out->probs, Y);
    copy_vector(cache_out->probs_before_sigma, cache_out->probs, Y);
  }
#endif

  copy_vector(cache_out->X, X_one_hot, S);

#ifdef WINDOWS
  free_vector(&tmp);
#endif
}
```

```c
  fp = fopen(argv[1], "r");
  if ( fp == NULL ) {
    printf("Could not open file: %s\n", argv[1]);
    return -1;
  }

  while ( ( c = fgetc(fp) ) != EOF ) {
    set_insert_symbol(&set, (char)c );
    ++file_size;
```

```
}

fclose(fp);

X_train = calloc(file_size+1, sizeof(int));
if ( X_train == NULL )
  return -1;

X_train[file_size] = X_train[0];

Y_train = &X_train[1];

fp = fopen(argv[1], "r");
while ( ( c = fgetc(fp) ) != EOF )
  X_train[sz++] = set_char_to_indx(&set,c);
fclose(fp);

if ( read_network != NULL ) {
  int FRead;
  int FReadNewAfterDataFile;

  initialize_set(&set);

  lstm_load(read_network, &set, &params, &model_layers);

  if ( seed == NULL ) {

    FRead = set_get_features(&set);

    // Read from datafile, see if new features appear

    fp = fopen(argv[1], "r");
    if ( fp == NULL ) {
      printf("Could not open file: %s\n", argv[1]);
      return -1;
    }

    while ( ( c = fgetc(fp) ) != EOF ) {
      set_insert_symbol(&set, (char)c );
    }

    fclose(fp);

    FReadNewAfterDataFile = set_get_features(&set);
```

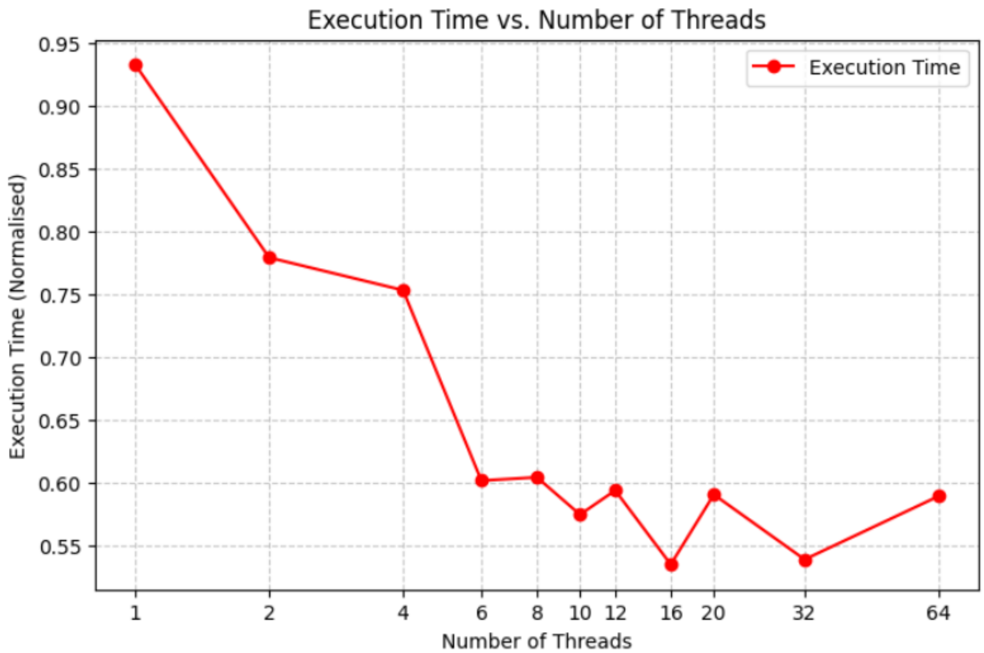
```
      if ( FReadNewAfterDataFile > FRead ) {
        // New features appeared. Must change
        // first and last layer.
        printf("New features detected in datafile.\nLoaded network worked with %d
features\
, now there is %d features in total.\n\
Reallocating space in network input and output layer to accommodate this new
feature set.\n",
          FRead, FReadNewAfterDataFile);

        lstm_reinit_model(
          model_layers,
          params.layers,
          FRead,
          FReadNewAfterDataFile
        );

      }

    }

    if ( seed == NULL )
      printf("Loaded the net: %s\n", read_network);
  } else {
    /* Allocating space for a new model */
    model_layers = calloc(params.layers, sizeof(lstm_model_t*));

    if ( model_layers == NULL ) {
      printf("Error in init!\n");
      exit(-1);
    }

    p = 0;
    while ( p < params.layers ) {
      // All layers have the same training parameters
      int X;
      int N = params.neurons;
      int Y;

      if ( params.layers == 1 ) {
        X = set_get_features(&set);
        Y = set_get_features(&set);
      } else {
        if ( p == 0 ) {
          Y = set_get_features(&set);
```

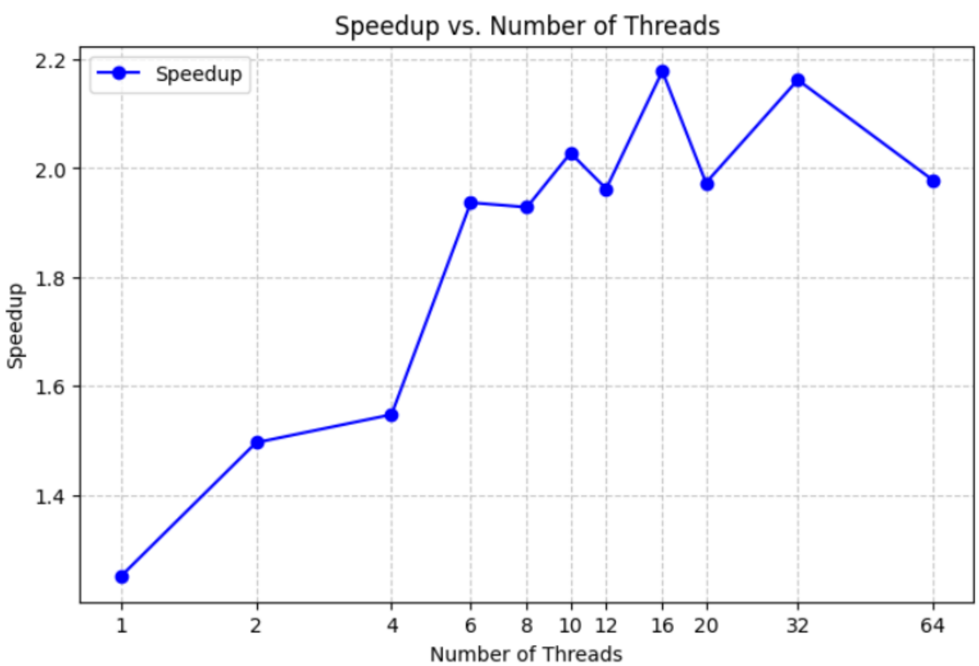
```c
          X = params.neurons;
        } else if ( p == params.layers - 1 ) {
          Y = params.neurons;
          X = set_get_features(&set);
        } else {
          Y = params.neurons;
          X = params.neurons;
        }
      }

      lstm_init_model(X, N, Y, &model_layers[p], 0, &params);

      ++p;
    }
  }

  if ( write_output_directly_bytes && read_network != NULL ) {

    lstm_output_string_layers(model_layers, &set, set_indx_to_char(&set, 0),
write_output_directly_bytes, params.layers);

    free(model_layers);
    free(X_train);
    return 0;
  } else if ( write_output_directly_bytes && read_network == NULL ) {
    usage(argv);
  }

  if ( seed != NULL ) {
    // output directly
    lstm_output_string_from_string(model_layers, &set, seed, params.layers, 256);

  } else {
    double loss;

    assert(params.layers > 0);

    printf("LSTM Neural net compiled: %s %s, %u Layers, ",
      __DATE__, __TIME__, params.layers);
    // Print neurons in each layer
    printf("Neurons: [");
    p = 0;
    while ( p < params.layers ) {
      printf("%s%d", (p>0?", ":""), model_layers[p]->N);
      ++p;
```

```
    }
    printf("], Features: %d.\n", model_layers[params.layers-1]->X);
    printf("Allocated bytes for the network: %s\n",
prettyPrintBytes(e_alloc_total()));
    printf("Training parameters: Backprop Through Time: %d, LR: %lf, Mo: %lf, LA:
%lf, LR-decrease: %lf.\n",
      MINI_BATCH_SIZE, params.learning_rate, params.momentum, params.lambda,
params.learning_rate_decrease);

    signal(SIGINT, store_the_net_layers);

    lstm_train(
      model_layers,
      &params,
      &set,
      file_size,
      X_train,
      Y_train,
      params.layers,
      &loss
    );

    if ( store_after_training ) {
      lstm_store(params.store_network_name_raw, &set,
      model_layers, params.layers);
      lstm_store_net_layers_as_json(model_layers, params.store_network_name_json,
        JSON_KEY_NAME_SET, &set, params.layers);
    }

    printf("Loss after training: %lf\n", loss);
  }

  free(model_layers);
  free(X_train);
```

```
vererYc.
mJfk9rhmya agoagIdmdM
-mtos 2ro
aa frrr fhao ge ind ther wJat
uiwte yo iehinid onliid mitd.
 9onUm aintedT Ceawsydsrsras dd bat vtoree aen eEficlte fhderdtRpste
=====================================================
18:09:43 Iteration: 600 (epoch: 14), Loss: 2.567078, record: 2.567078 (iteration: 600), LR: 0.001000
=====================================================
Iire wairn  band tesire
wnde ,en kuintodh e
vererYc.
 9onUm aintedT Ceawsydsrsras dd bat vtoree aen eEficlte fhderdtRpste
=====================================================
18:09:43 Iteration: 600 (epoch: 14), Loss: 2.567078, record: 2.567078 (iteration: 600), LR: 0.001000
=====================================================
Iire wairn  band tesire
wnde ,en kuintodh e
vererYc.
in bhirgiinty yigmere, ausat anc. o8erthne Irdu in
wnde ,en kuintodh e
vererYc.
in bhirgiinty yigmere, ausat anc. o8erthne Irdu in
in bhirgiinty yigmere, ausat anc. o8erthne Irdu in
r aude aud eand ficihe th.
ino dit iomlgochangI
=====================================================
18:41:26 Iteration: 700 (epoch: 17), Loss: 2.446119, record: 2.442453 (iteration: 695), LR: 0.001000
18:41:26 Iteration: 700 (epoch: 17), Loss: 2.446119, record: 2.442453 (iteration: 695), LR: 0.001000
=====================================================
d0V U?ua8gvett erou4II)gonte aad eamani
mwte inym24
mwte inym24
.oaam
.oaam
ailgas in5 hov.. , top oud. I 4 fthe. veecindas atales soe il irais uth al
peot filbe tetre
ailgas in5 hov.. , top oud. I 4 fthe. veecindas atales soe il irais uth al
peot filbe tetre
wases rakocr,coutg poneoH as .ferehh an atdem
=====================================================
19:13:11 Iteration: 800 (epoch: 19), Loss: 2.347986, record: 2.347986 (iteration: 800), LR: 0.001000
wases rakocr,coutg poneoH as .ferehh an atdem
=====================================================
19:13:11 Iteration: 800 (epoch: 19), Loss: 2.347986, record: 2.347986 (iteration: 800), LR: 0.001000
=====================================================
=====================================================
gin)
Fn1Ho vey lishm wose to hhin site ante ous waW :hhe, he ak the fol
wio thm ir ako Id iatn wfwim te caren sa
 rot hir,tig hine file at he ente s
=====================================================
19:45:00 Iteration: 900 (epoch: 21), Loss: 2.251413, record: 2.251413 (iteration: 900), LR: 0.001000
=====================================================
N?,eygdth Wtoit the wfxlee Yecacdecin.
mI hdes ntsom ead pafi as do tlos ind ha teal tou far in tha me tof lon lpan weitg. the ciuDto lo cands wenedM
=====================================================
```

PLOTS :





Inference on Normalized Execution Times for Parallelizing LSTM using OpenMP

Inference on Normalized Execution Times for Parallelizing LSTM using OpenMP

1.1 Initial Performance Gains (1 to 6 threads)

- Execution time **reduces significantly** from **0.9325 (1 thread) to 0.6017 (6 threads)**.
- This indicates that the **parallelization is effective in this range**, efficiently distributing computations across multiple threads.

1.2 Diminishing Returns (8 to 12 threads)

- Execution time remains almost **constant** between **6 and 12 threads**, with values fluctuating around **0.5747 to 0.5939**.
- This suggests that **parallel efficiency is decreasing**, and increasing threads further does not provide substantial benefits.
- Possible reasons:
    - **Synchronization overhead**
    - **Memory bandwidth bottlenecks**

1.3 Fluctuations & Saturation (16+ threads)

- At **16 threads**, execution time drops to **0.5350**, showing a slight improvement.
- However, beyond 16 threads (**20, 32, and 64 threads**), execution time fluctuates, increasing instead of decreasing.
    - **20 threads: 0.5906** (increase)
    - **32 threads: 0.5390** (small drop)
    - **64 threads: 0.5893** (increase again)
- This irregular pattern suggests that **overhead from excessive thread creation and memory contention limits performance gains.**
- At higher thread counts, **the cost of managing parallel execution outweighs the benefits of additional cores.**

---

2. Parallelization Fraction Calculation

Using Amdahl's Law:

S=Tseq/Tparallel

f=(1−(1/S))/(1−(1/p))

The parallelization fraction is approximately **57.7%**, for 16 threads.

---

3. Interpretation of Parallelization Fraction

- **57.7% of the workload is parallelizable**, while **42.3% remains sequential**.
- This suggests that the LSTM implementation has **significant sequential components** that limit performance scaling.
- **Beyond 16 threads, performance does not improve significantly** due to:
  - **Thread synchronization overhead**
  - **Memory contention**
  - **Load imbalance across threads**

5. Conclusion

- **LSTM parallelization provides noticeable speedup up to 6-8 threads.**
- **Performance gains diminish beyond 12 threads**, with fluctuations due to synchronization and memory access bottlenecks.
- The **parallelization fraction is 57.7%**, indicating that a significant portion of the workload remains sequential.
- **Further optimizations are required** to improve parallel efficiency, particularly in memory access and thread management.