#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <time.h>

#include <pthread.h>

struct Params

{

int \*start;

size\_t len;

int depth;

};

// only used for synchronizing stdout from overlap.

pthread\_mutex\_t mtx = PTHREAD\_MUTEX\_INITIALIZER;

// forward declare our thread proc

void \*merge\_sort\_thread(void \*pv);

// a simple merge algorithm. there are \*several\* more efficient ways

// of doing this, but the purpose of this exercise is to establish

// merge-threading, so we stick with simple for now.

void merge(int \*start, int \*mid, int \*end)

{

int \*res = malloc((end - start)\*sizeof(\*res));

int \*lhs = start, \*rhs = mid, \*dst = res;

while (lhs != mid && rhs != end)

\*dst++ = (\*lhs <= \*rhs) ? \*lhs++ : \*rhs++;

while (lhs != mid)

\*dst++ = \*lhs++;

while (rhs != end)

\*dst++ = \*rhs++;

// copy results

memcpy(start, res, (end - start)\*sizeof(\*res));

free(res);

}

// our multi-threaded entry point.

void merge\_sort\_mt(int \*start, size\_t len, int depth)

{

if (len < 2)

return;

if (depth <= 0 || len < 4)

{

merge\_sort\_mt(start, len/2, 0);

merge\_sort\_mt(start+len/2, len-len/2, 0);

}

else

{

struct Params params = { start, len/2, depth/2 };

pthread\_t thrd;

pthread\_mutex\_lock(&mtx);

printf("Starting subthread...\n");

pthread\_mutex\_unlock(&mtx);

// create our thread

pthread\_create(&thrd, NULL, merge\_sort\_thread, &params);

// recurse into our top-end parition

merge\_sort\_mt(start+len/2, len-len/2, depth/2);

// join on the launched thread

pthread\_join(thrd, NULL);

pthread\_mutex\_lock(&mtx);

printf("Finished subthread.\n");

pthread\_mutex\_unlock(&mtx);

}

// merge the paritions.

merge(start, start+len/2, start+len);

}

// our thread-proc that invokes merge\_sort. this just passes the

// given parameters off to our merge\_sort algorithm

void \*merge\_sort\_thread(void \*pv)

{

struct Params \*params = pv;

merge\_sort\_mt(params->start, params->len, params->depth);

return pv;

}

// public-facing api

void merge\_sort(int \*start, size\_t len)

{

merge\_sort\_mt(start, len, 4); // 4 is a nice number, will use 7 threads.

}

int main()

{

static const unsigned int N = 2048;

int \*data = malloc(N \* sizeof(\*data));

unsigned int i;

srand((unsigned)time(0));

for (i=0; i<N; ++i)

{

data[i] = rand() % 1024;

printf("%4d ", data[i]);

if ((i+1)%8 == 0)

printf("\n");

}

printf("\n");

// invoke our multi-threaded merge-sort

merge\_sort(data, N);

for (i=0; i<N; ++i)

{

printf("%4d ", data[i]);

if ((i+1)%8 == 0)

printf("\n");

}

printf("\n");

free(data);

return 0;

}

An interesting task would be performing this functionality using qsort() for sorting the smaller partitions or once the thread pool reaches exhaustion. qsort() is a pretty big hammer to bring to this party, and as such you're going to want to raise the minimum partition size to something respectful (in the example below, we use 256 elements).

So what would it take to integrate qsort() the the sub partitions rather than a hand-rolled merge-sort? Surprisingly, not much. Start with a qsort() compatible comparator:

// comparator for qsort

int cmp\_proc(const void \*arg1, const void\* arg2)

{

const int \*lhs = arg1;

const int \*rhs = arg2;

return (\*lhs < \*rhs) ? -1 : (\*rhs < \*lhs ? 1 : 0);

}

Pretty brain-dead. Now, modify the mt-wrapper to look something like this:

// our multi-threaded entry point.

void merge\_sort\_mt(int \*start, size\_t len, int depth)

{

if (len < 2)

return;

// invoke qsort on the partition. no need for merge

if (depth <= 0 || len <= 256)

{

qsort(start, len, sizeof(\*start), cmp\_proc);

return;

}

struct Params params = { start, len/2, depth/2 };

pthread\_t thrd;

pthread\_mutex\_lock(&mtx);

printf("Starting subthread...\n");

pthread\_mutex\_unlock(&mtx);

// create our thread

pthread\_create(&thrd, NULL, merge\_sort\_thread, &params);

// recurse into our top-end parition

merge\_sort\_mt(start+len/2, len-len/2, depth/2);

// join on the launched thread

pthread\_join(thrd, NULL);

pthread\_mutex\_lock(&mtx);

printf("Finished subthread.\n");

pthread\_mutex\_unlock(&mtx);

// merge the paritions.

merge(start, start+len/2, start+len);

}

Shell Scipt

#!/bin/bash

dirt=$1 #Get the first argument

RESULT= 'ls -a $dirt | sort' #sort all files lexicographically

SETD=$'\n' #files with spaces are recongnized

declare -a ARRAY1 #ARRAY1 holds files begnning with '.'

declare -a ARRAY2 #ARRAY2 holds other regular file

let countdot=0 # count for the ARRAY1

let count=0 #count for the ARRAY2

for FILE in $RESULT #loop will go through the output of the ls command

do

if [! -r "$dirt/$FILE" ] #if the file has no r

then

echo "You don't have permission to read the file $dirt/$FILE"

fi

if [[ ! -L "$dirt/$FILE" && -r "$dirt/$FILE" ]] #check the file is not symblic and readable

then

if [ -f "$dirt/$FILE" ] #check if it is regular file

then

if [ "${FILE:0:1}" == "." ] #check if the file starts with "." in ARRAY1

then

ARRAY1[$countdot]= "$dirt/$FILE"

let countdot=countdot+1

else

ARRAY[$count]= "$dirt/$FILE"

let count=count+1

fi

fi

fi

done

ARRAY1=("${ARRAY1[@]}" "${ARRAY2[@}")

let count=${#ARRAY1[@]} # let the overall count be the size of the ARRAY1

exec 2>/dev/null

for ((i=0; i<$count; i++))

do

for((j=$i+1; j<$count; j++))

do

if[[ -s "${ARRAY1[$i]}" && -s "${ARRAY1[$j]} || ! -s "${ARRAY1[$i]}" &&! -s "${ARRAY1[$j]}" ]]

then

compare=$(cmp -s -- "${ARRAY1[$I]}" "${ARRAY2[$j]}")

if [ ! $compare ]

then

ln -f -- "${ARRAY1[$i]}" "${ARRAY1[$j]}"

fi

fi

done

done