SMPL Programming Language Proposal

Andrei Papancea, Ajay Siva Santosh, Devashi Tandon
{alp2200, ac3647, dt2450} @columbia.edu

September 16, 2013

Motivation

Up until 2006, computer scientists benefited from what is referred to as the “free lunch”[1]. Programmers did not have to focus too much on writing very efficient code, since it was known that every two years processor speeds will increase and hence help their programs run faster. According to Moore’s Law[2] the number of transistors on integrated circuits doubles approximately every two years. This law still holds today, but it is not practical to increase processor speed anymore because of financial and physical constraints such as the high cost of dealing with excessive heat dissipation. Subsequently, in a multicore world, the computer scientists all over are looking at parallel programming languages as the solution.

Two issues with parallel programming are that it is not trivial given the current toolsets and that not every problem can be parallelized. Furthermore, most popular programming languages such as C, C++, Java, or Python either do not support parallel programming or they have a very complex implementation of it, since they were not built with that idea in mind. Due to the non-trivial syntax of most parallel programming -capable languages, the concept of parallel programming is rarely taught at the undergraduate level — and it is crucial to teach this concept at the undergraduate level in order to stir interest in the field and push for advancement. Hence, there is a need for programming languages with simpler syntax, that can better illustrate parallel programming concepts and that makes them more suitable for undergraduate teaching[3].

Language Description

SMPL is an approach to parallel programming that is based on a subset of C. Its major strength is that it has clearer and simpler syntax for doing parallel programming than programs written using OpenMP or POSIX threads.

A SMPL program consists of a sequence of function, variable, and array definitions, similar to C.

[1] Sutter, Herb. "The free lunch is over: A fundamental turn toward concurrency in software." *Dr. Dobb’s Journal* 30.3 (2005): 202-210.
[2] Schaller, Robert R. "Moore's law: past, present and future." *Spectrum, IEEE* 34.6 (1997): 52-59.
[3] Graf, Michael et al. "Selecting and using a parallel programming language." *Proceeding of the 44th ACM technical symposium on Computer science education* 6 Mar. 2013: 735-735.

Syntax Description

The syntax of a SMPL program is like in C. There are 4 parallel constructs in SMPL:

1. `pfor`
    a. **Syntax:** `pfor(num_threads; initializer; stop condition; updater) { statements }`
    b. **Definition:** breaks the work of the for loop into the specified number of threads
    c. **Features:** includes implicit thread exit and join
2. `spawn`
    a. **Syntax:** `spawn { statement }`
    b. **Definition:** spawns the specified statement into a separate thread, causing it to run asynchronously
3. `lock`
    a. **Syntax:** `lock { statement }`
    b. **Definition:** the C-equivalent of a *mutex* lock, which basically protects the operations inside it from thread concurrency
4. `barrier`
    a. **Syntax:** `barrier;`
    b. **Definition:** prevents code below the *barrier* from executing until the currently *spawn*-ed threads have finished

Syntax Examples

Example 1: *Sample Syntax to calculate the sum of first n numbers and checking if there are greater than max without using parallelization.*

```
void main(){

	/* This is a multi line
	   comment. Nesting is not
	   supported at this
	   point. */

	int i=0;
	int sum=0;
	int n;
	int max;

	printf(“Enter the value of n and max \n”);
	scanf(“%d,%d”, &n, &max);

	for(i=0;i<n;i++){
		sum += i;
	}

	if(sum>max){
		printf(“The sum is greater than max”);
	} else {
		printf(“The sum is not greater than max”);
	}

}

```

Example 2: *Sample Syntax to calculate the sum of first n numbers and checking if there are greater than max using functions without using parallelization.*

```
void main(){

      int i=0;
      int n;
      int max;

      printf("Enter the value of n and max \n");
      scanf("%d,%d", &n, &max);

      int sum=F1(n);

      if(sum>max){
            printf("The sum is greater than max");
      } else {
            printf("The sum is not greater than max");
      }

}

int F1(int n){

      int i=0;
      int sum=0;
      for(i=0;i<n;i++){
            sum+=i;
      }
      return sum;

}

```

Parallelism in SMPL

The program below computes the sum of the first *n* positive integers using *t* threads, where *n* and *t* are inputted by the user:

```
void main(){
     int i=0;
     int sum=0;
     int n;
     int t;

     printf(“Enter the number of threads t: ”);
     scanf(“%d”, &t);

     printf(“Enter the upper bound n: ”);
     scanf(“%d”, &n);

     pfor(t; i=0; i<n; i++)
       lock {
         sum += i;
       }
     }

     printf(“The sum of the first %d numbers is %d.\n”,
             limit,sum);
}
```

The C-equivalent of the SMPL program above:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *thread_sum(void *);
pthread_mutex_t m=PTHREAD_MUTEX_INITIALIZER;
int sum=0;

void main(){
  int i,j;
  int limit = 1000000;
  int num_threads;

  printf("Enter number of threads: ");
  scanf("%d",&num_threads);

  pthread_t threads[num_threads];
  int uppers[num_threads];
```

```c
  int lowers[num_threads];
  int args[num_threads][2];

  printf("Enter upper bound: ");
  scanf("%d",&limit);

  for(i=0;i<num_threads;i++){
    uppers[i] = (limit/num_threads)*(i+1);
    lowers[i] = (limit/num_threads)*i;
    args[i][0] = lowers[i];
    args[i][1] = uppers[i];
  }

  for(i=0;i<num_threads;i++){
    pthread_create(&threads[i],NULL,
                   thread_sum,(void *)&args[i]);
  }

  for(i=0;i<num_threads;i++){
    pthread_join(threads[i],NULL);
  }

  printf("The sum of the first %d numbers is %d.\n",limit,sum);
}

void *thread_sum(void *args){
  int i;
  int *p = (int *)args;
  int lower = p[0];
  int upper = p[1];

  for(i=lower+1;i<=upper;i++){
    pthread_mutex_lock(&m);
    sum = sum+i;
    pthread_mutex_unlock(&m);
  }

  pthread_exit(NULL);
}
```

The program below spawns 4 tasks into 3 different threads and then uses *barrier* to wait for them to finish.

```
void main(){
      spawn {
         /* work */
         printf("Task 1\n");
      }
      spawn {
         /* work */
         printf("Task 2\n");
      }
      spawn {
         /* work */
         printf("Task 3\n");
      }
      spawn {
         /* work */
         printf("Task 4\n");
      }

      barrier;

      printf("Done.\n");
}
```

The C-equivalent of the SMPL program above:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *thread_work(char *);

void main(){
  int i;
  int num_threads;

  printf("Enter number of threads: ");
  scanf("%d",&num_threads);

  pthread_t threads[num_threads];

  for(i=0;i<num_threads;i++){
    char str[7];
    strncpy(str,”Task “,5);
    strncat(str,(i+1),1);
    str[6] = ‘\0’;
    pthread_create(&threads[i],NULL,
                   thread_work,(char *)str);
  }

  for(i=0;i<num_threads;i++){
    pthread_join(threads[i],NULL);
  }

  printf("Done!”);
}

void *thread_work(char *str){
  printf(“%s”,str);
  pthread_exit(NULL);
}
```