[KISTI 기술문서] ISBN 978-89-294-1696-6

OpenMP/OpenACC 병렬 프로그래밍 API 활용

2024. 9. 1.

한국과학기술정보연구원
슈퍼컴퓨팅기술개발센터

저자소개

김상완

한국과학기술정보연구원
국가슈퍼컴퓨팅본부 슈퍼컴퓨팅기술개발센터
책임연구원
sangwan@kisti.re.kr

정기문

한국과학기술정보연구원
국가슈퍼컴퓨팅본부 슈퍼컴퓨팅기술개발센터
책임연구원
kmjeong@kisti.re.kr

이 기술보고서는 2024년도 한국과학기술정보연구원(KISTI)의
기본사업으로 수행된 연구입니다.
과제번호: (KISTI) K24-L02-C06-S01
과제명: 초고성능컴퓨팅 공동활용을 위한 통합 환경 개발 및 구축

목 차

1. 개요

- 프로그래밍 모델[1]은 하드웨어의 복잡성을 추상화하여 소프트웨어 개발자가 프로그램을 작성할 수 있게 하는 프로그램 방법론이다. 컴퓨터의 구조가 발전되어 감에 따라 프로그래밍 모델도 변화되고 있다.
- 초기의 프로그래밍 모델은 주로 순차적 프로그래밍에 기반한다. 포트란(Fortran)과 같은 초기 고급 프로그래밍 언어는 정적 메모리 할당과 같은 효율적인 실행이 주목적이었다. 다중처리(multi-processing)가 가능하게 되어, 여러 프로그램이 동시에 실행될 수 있게 됨에 따라 비동기 프로그램이 일반화 되었다. 멀티 코어 CPU가 등장하게 되자, 스레드와 같이 메모리 공간을 공유하는 공유 메모리 모델이 등장하였다. 클러스터와 같은 분산 컴퓨팅 환경에서는 여러 노드(컴퓨터)가 마치 하나의 메모리 공간을 사용하는 것처럼 작동하는 분산 공유 메모리 프로그래밍 모델이 필요하게 되었다. 이와 같이 프로그래밍 모델은 하드웨어의 발전과 함께 변화되어 가고 있다.
- 특히 고성능 컴퓨팅을 위한 병렬 프로그램과 관련하여 하드웨어의 성능을 최대한 활용하는 것이 중요하다. 그러나 새로운 컴퓨터 구조를 위해 새로운 언어를 개발하는 것은 맞지 않기 때문에 API를 통하여 병렬 실행 모델을 구현하는 것이 일반적 이다.
- 병렬 처리 방식의 종류 :

* 데이터 병렬(data parallelism)은 동일한 작업이 여러개 있을 때 이를 분산하여 동시에 수행하는 방식. 벡터처리, 행렬연산 등이 이에 해당. 서로 다른 작업끼리는 의존성없이 독립적으로 실행가능. 데이터는 서로 다르지만 동일한 처리가 수행되므로 SIMD(single instruction, multiple data)에 해당한다.
* 태스크 병렬(task parallelism)은 서로 다른 작업들을 동시에 수행하는 방법, 각 태스크는 독립적으로 수행가능. 웹서버가 여러 클라이언트의 요청을 동시에 처리하는 것, CPU코어가 서로 다른 프로그램을 동시에 수행하는 것이 이에 해당함. 데이터도 다르고 이를 처리하는 방법도 상이하므로 MIMD(multiple instruction, multiple data)에 해당함.
* 파이프라인 병렬(pipeline parallelism)은 순차적인 작업을 여러 단계로 나누고, 각 단계가 서로 다른 프로세서에서 동시에 실행되도록 처리하는 방식. 이전 단계의 출력은 다음 단계의 입력으로 연결. 실시간 스트리밍 데이터의 처리, 이미지 처리와 같은 연속적인 데이터 처리에 효과적임.

- 병렬 처리에서는 단위작업 크기가 전체 성능에 영향을 미칠 수 있음:

* 잘개 나눔(find-grained) : 프로세서 간에 균등하게 작업을 분배할 수 있음. 작업을 나누고 합치고 동기화 하는 과정에서 오버헤드가 증가함.
* 크게 나눔(coarse-grained) : 작업을 나누고 합치고 동기화 하는 비용이 작음. 큰 작업 단위로 인해 로드 밸런싱이 어려워 질 수 있음

- 본 기술문서에서는 pthread를 이용하 스레드 프로그래밍, OpenMP 및 OpenACC API에 대해 소개하고 예제 프로그램을 위주로 설명한다.
- 본 문서에서 사용자 예제코드는 다음 깃허브 주소에서 찾을 수 있다:
  `https://github.com/swkim85/linux-drill`

1) https://en.wikipedia.org/wiki/Programming_model

2. 스레드 프로그래밍

- 스레드(thread)는 보통 경량 프로세스라고 불리운다. 스레드는 한 프로세스내에서 주소 영역과 다른 자원들을 공유한다. 글로벌 메모리(heap)은 공유하지만, 각 스레드는 자기 자신의 스택(stack)을 유지함.
- POSIX[2] Threads(Pthreads) 는 1995년 IEEE 표준 POSIX.1c-1995 로 정의됨. [3][4]
- pthreads는 여러 Unix-like POSIX OS에서 구현됨. (FreeBSD, NetBSD, OpenBSD, Linux, macOS, Android, Solaris, Redoc 등). **libpthread** 라는 라이브러리로 제공된다.
- C언어 라이브러리 헤더파일인 **pthread.h** 에는 라이브러리에서 제공하는 type, function 등을 정의하고 있다. 약 100개 이상의 함수가 존재하며 이들은 다음과 같은 그룹으로 나뉜다:[5]
  - 스레드 관리: create, exit, join
  - 뮤텍스(mutex) : 스레드간 상호배제(mutual exclusion)를 위한 동기화
  - 조건 변수(conditional variables) : 스레드간 동기화와 통신을 위한 용도
  - 스레드간 동기화 : read/write locks, barriers
  - 스핀락(spinlock) : 임계영역에 진입하려는 스레드가 락을 획득할 때까지 계속 확인(spin)함.

※ LinuxThreads 는 리눅스용 POSIX Threads의 초기 구현 버전임. NPTL(Native POSIX Thread Library)는 리눅스에서 pthreads 구현을 효율화한 것으로 성능과 확장성 측면에서 우수하다. NPTL은 glibc 2.3.2 이후에 적용되었고, 리눅스 커널 2.6 버전 이상이 요구됨.[6]

2.1. 스레드 관리

- 스레드 프로그래밍 예제1: pthread_create() 호출을 통해 스레드를 만듦(프로세스의 fork()와 같은 기능). pthread_join()은 스레드가 종료될때까지 대기한다.

```
형식:  int pthread_create ( pthread_t *newthread, const pthread_attr_t *attr,
                      void *(start_routine)(void *),
                      void *arg );
       int pthread_join ( pthread_t thread, void **retval);
```

```
// thread1.c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void *MyThreadFunc(void* arg){
  printf("Hello message from a thread\n");
  return NULL;
}
int main() {
  pthread_t aThread;
  pthread_create(&aThread, NULL, MyThreadFunc, NULL);
  pthread_join(aThread, NULL);  // 스레드가 종료될때까지 대기
  printf("main program exit.\n");  // 스레드가 종료된 이후 실행됨
  return 0;
}
```

```
$ gcc -g  -o thread1  thread1.c   -pthread
$ ./thread1
```

2) Portable Operating System Interface https://ko.wikipedia.org/wiki/POSIX
3) https://standards.ieee.org/ieee/1003.1c/1393/
4) man 7 pthreads 참고
5) https://www.man7.org/linux/man-pages/man0/pthread.h.0p.html
6) man 7 nptl 참고

```
Hello message from a thread
main program exit.
```

- 만일 main thread가 pthread_join 을 호출하지 않고 프로세스를 종료한다면 해당 프로세스에 속한 모든 스레드는 자동으로 종료되고 정리된다.

```
// thread2.c  (thread1.c 에서 수정)
[...]
int main() {
  pthread_t aThread;
  pthread_create(&aThread, NULL, MyThreadFunc, NULL);
  // pthread_join(aThread, NULL); // join을 하지 않고 프로세스를 종료
  printf("main program exit.\n");
  return 0;
}
```

```
$ gcc -g  -o  thread2  thread2.c  -pthread
$ ./thread2
main program exit.
Hello message from a thread  <-- main thread 가 종료된 이후에는 출력되지 않음
```

- 자식 스레드를 부모 스레드와 분리(detach) 시킬 수 있다. 분리된 스레드는 더 이상 joinable thread가 아니며, 부모 스레드는 종료된 쓰레드를 join할 필요가 없게된다. joinable 하지 않은 스레드를 join 하려고 하면 EINVAL 값을 반환한다.
- 자식 스레드를 부모 스레드와 분리 시킬 때 장점은 ① 자원자동해제: 스레드가 종료될 때 자동으로 자원(스레드 스택, 스레드 제어 블록 등)이 해제됨 ② 비차단(non-blocking): 부모 스레드는 종료를 기다릴 필요없이 바로 다음 작업을 수행할 수 있음 ③ 독립적 운용: 스레드의 실행결과를 돌려 받아야 하는 경우가 아니라면 조인을 할 필요가 없음. 많은 스레드를 생성하고 관리하는 경우에 유리함

```
// thread3.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
void *MyThreadFunc(void* arg){
  usleep(10000);
  printf("thread exit.\n");
  return NULL;
}
int main() {
  pthread_t aThread;
  int ret;
  for (int i = 0; ; i++) {
    ret = pthread_create(&aThread, NULL, MyThreadFunc, NULL);
    if (ret) { printf(" pthread_create failed \n"); break; }
#ifdef DETACH
    printf("detach thread.\n");
    pthread_detach (aThread);
#endif
#ifdef JOIN
    ret = pthread_join(aThread, NULL);
    if (ret) { printf(" pthread_join failed \n"); }
```

```
#endif
    printf("iter = %d \n", i);
    usleep(10000);
  }
  printf("main program exit.\n");
  return 0;
}
```

```
// 메모리 오버커밋(memory overcommit) 기능을 끄고 테스트한다.
$ echo 2 ¦ sudo tee /proc/sys/vm/overcommit_memory
2
$ gcc -g  -o  thread3  thread3.c  -pthread
$ ./thread3
[...]
thread exit.
iter = 211
thread exit.
 pthread_create failed   // 스레드 join 을 하지 않으면 생성할 수 있는 스레드 개수에 제한있음
main program exit.
// 스레드 join을 하거나, detach 을 하면, 스레드 종료시 리소스는 자동으로 해제된다.
$ gcc -g  -o  thread3  thread3.c  -pthread -DJOIN
$ gcc -g  -o  thread3  thread3.c  -pthread -DDETACH
```

※ 프로세스가 생성할 수 있는 최대 스레드 개수는 시스템 자원과 운영체제 설정에 따라 달라질 수 있음. 일반적으로 리눅스에서 스택사이즈는 기본적으로 8MB임. 스레드는 각자 고유의 스택을 가지므로, 스레드 생성이 많을 수록 메모리 사용량이 증가함. 보다 많은 수의 스레드를 생성하기 위해서는 기본 스택의 크키를 줄일 수 있는데, 스텍의 크기를 줄이거나 늘리기 위해서는 **pthread_attr_setstacksize** 함수를 사용하거나, **ulimit (ulimit -s)** 설정을 통해서 조정이 가능함.

※ ps 명령으로 스레드를 모니터링 할 수 있다. ps -L 옵션을 이용하면 LWP(light-weight process)정보를 확인할 수 있음. NLWP 컬럼은 스레드의 개수를 의미.

```
$ ps -eLf  ¦ egrep "UID¦thread3"
ps -e -o pid,tid,lwp,rss,vsz,cmd ¦ egrep "PID¦thread"
UID          PID    PPID     LWP  C NLWP STIME TTY          TIME CMD
ubuntu   1750152    1404 1750152  0    2 06:20 pts/1    00:00:00 ./thread3
ubuntu   1750152    1404 1750569  0    2 06:20 pts/1    00:00:00 ./thread3
```

※ **pmap** 명령으로 프로세스의 메모리 영역을 조회할 수 있다. **anon** 으로 표시되는 부분은 익명 메모리(anonymous memory)를 의미함.

```
$ pmap -x  $(ps ax ¦ grep thread3 ¦ head -n 1 ¦ awk '{print $1}')
35152:   ./thread3
Address           Kbytes     RSS   Dirty Mode  Mapping
00006148e24e0000       4       4       0 r---- thread3
00006148e24e1000       4       4       0 r-x-- thread3
00006148e24e2000       4       4       0 r---- thread3
[...]
000073ce4b9ff000       4       0       0 -----   [ anon ]
000073ce4ba00000    8192       8       8 rw---   [ anon ]
000073ce4c200000     160     160       0 r---- libc.so.6
[...]
00007ffe9b978000     132      12      12 rw---   [ stack ]
```

```
00007ffe9b9d2000      16       0       0 r----   [ anon ]
00007ffe9b9d6000       8       4       0 r-x--   [ anon ]
ffffffffff600000       4       0       0 --x--   [ anon ]
---------------- ------- ------- -------
total kB          150308    1940     244
```

glibc 2.34 에서 libpthread를 포함한 몇개의 라이브러리가 glibc에 통합됨.[7)]
glibc 버전 확인은 so.6 파일을 직접 실행하면 됨

```
$ ldd -v thread1
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x0000757634c00000)
[...]
$ /lib/x86_64-linux-gnu/libc.so.6 --version
GNU C Library (Ubuntu GLIBC 2.35-0ubuntu3.8) stable release version 2.35.
[...]
```

2.2. 뮤텍스 동기화

- 뮤텍스(mutex, mutual execlusion, 상호배제)는 멀티스레드 프로그래밍에서 동기화를 위해 사용된다. 여러 스레드가 공유 자원에 동시에 접근하는 것(race condition)을 방지하여 데이터의 일관성을 유지하게 함. 임계영역(critical section)에 대한 독점적 접근을 보장함. 한 스레드가 뮤텍스를 획득하면 해제될 때까지 다른 스레드는 접근할 수 없게 됨.
- 뮤텍스를 사용하지 않아 경쟁 조건을 발생시키는 코드 예제

```
// mutex1.c
[...]
int counter = 0;
void *threadCount(void *arg) {
  int i;
  char* name = (char*)arg;
  counter = 0;
  for (i = 0; i < 3; i++) {
    printf("%s counter: %d\n", name, counter);
    usleep(50);
    counter++;    // race condition
  }
}
int main() {
  pthread_t thread1;
  pthread_t thread2;
  pthread_create(&thread1, NULL, threadCount, (void *)"thread1");
  pthread_create(&thread2, NULL, threadCount, (void *)"thread2");
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
  return 0;
}
   // pthread 가 glibc 에 통합되었기 때문에 -pthread 옵션은 없어도됨
gcc -g  -o  mutex1  mutex1.c -pthread
$ ./mutex1
thread1 counter: 0
thread2 counter: 0
```

7) https://developers.redhat.com/articles/2021/12/17/why-glibc-234-removed-libpthread#

```
thread2 counter: 1
thread1 counter: 2
thread1 counter: 3
thread2 counter: 4
```

- 뮤텍스를 사용하여 두개의 스레드가 상호 배타적으로 동작.

```
// mutex2.c
[...]
pthread_mutex_t  mutex = PTHREAD_MUTEX_INITIALIZER;
int counter = 0;
void *threadCount(void *arg) {
  int i;
  char* name = (char*)arg;
  pthread_mutex_lock(&mutex);
  counter = 0;
  for (i = 0; i < 3; i++) {
    printf("%s counter: %d\n", name, counter);
    usleep(50);
    counter++;
  }
  pthread_mutex_unlock(&mutex);
}
int main() {
  pthread_t thread1;
  pthread_t thread2;
  pthread_mutex_init(&mutex, NULL);
  pthread_create(&thread1, NULL, threadCount, (void *)"thread1");
  pthread_create(&thread2, NULL, threadCount, (void *)"thread2");
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
  pthread_mutex_destroy(&mutex);
  return 0;
}
$ gcc -g  -o  mutex2  mutex2.c -pthread
$ ./mutex2
thread1 counter: 0
thread1 counter: 1
thread1 counter: 2
thread2 counter: 0
thread2 counter: 1
thread2 counter: 2
```

※ pthread_mutex_t 타입은 pthread 관련 헤더 파일에 아래와 같이 정의되어 있다. 구조체 내부에는 다음과 같은 속성값들이 들어 있다: 잠금 상태를 나태나는 정수 값, 소유자의 스레드 번호, 뮤텍스 속성을 나타내는 플래그, 대기중인 스레드의 큐를 관리하기 위한 데이터 구조

```
 /usr/include/bits/struct_mutex.h
struct __pthread_mutex_s {
  int __lock;
  unsigned int __count;
  int __owner;
#ifdef __x86_64__
  unsigned int __nusers;
```

```
#endif
[...]
};
```

```
 /usr/include/bits/pthreadtypes.h
typedef union
{
  struct __pthread_mutex_s __data;
  char __size[__SIZEOF_PTHREAD_MUTEX_T];
  long int __align;
} pthread_mutex_t;
```

※ 데드락(deadlock, 교착상태)이란 여러 프로세스나 스레드가 자원을 기다리며 무한히 대기하는 상태를 의미함. 다음의 코드는 데드락을 발생시키는 전형적인 코드이다. 2개의 스레드가 각각 자신의 자원(뮤텍스)을 점유하고 있는 상태에서 상태방의 자원을 기다리고 있는 상황임. 데드락을 피하기 위해서 trylock을 사용하는 방법이 있다.

```
// deadlock.c
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;
void *thread1_function(void *arg) {
  pthread_mutex_lock(&mutex1);
  printf("thread1 locked mutex1 ...\n");
  sleep(1);
  pthread_mutex_lock(&mutex2);
  printf("thread1 locked mutex2 ...\n");
  printf("thread1 working...\n"); // 임계 영역 코드
  pthread_mutex_unlock(&mutex2);
  pthread_mutex_unlock(&mutex1);
  return NULL;
}
void *thread2_function(void *arg) {
  pthread_mutex_lock(&mutex2);
  printf("thread2 locked mutex2 ...\n");
  sleep(1);
  pthread_mutex_lock(&mutex1);
  printf("thread2 locked mutex1 ...\n");
  printf("thread2 working...\n"); // 임계 영역 코드
  pthread_mutex_unlock(&mutex1);
  pthread_mutex_unlock(&mutex2);
  return NULL;
}
int main() {
  pthread_t thread1, thread2;
  pthread_create(&thread1, NULL, thread1_function, NULL);
  pthread_create(&thread2, NULL, thread2_function, NULL);
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
  return 0;
}
```

```
$ gcc -g  -o  deadlock  deadlock.c -pthread
$ ./deadlock
```

```
thread1 locked mutex1 ...
thread2 locked mutex2 ...
^C    // 데드락 상태. 프로그램 종료
// mutex3.c (데드락을 피하기 위해 trylock을 사용)
[...]
void *thread1_function(void *arg) {
  pthread_mutex_lock(&mutex1);
  printf("thread1 locked mutex1 ...\n");
  sleep(1);
  if (pthread_mutex_trylock(&mutex2) == 0) {
    printf("thread1 locked mutex2 ...\n");
    printf("thread1 working...\n"); // 임계 영역 코드
    pthread_mutex_unlock(&mutex2);
    printf("thread1 unlocked mutex2 ...\n");
  }
  pthread_mutex_unlock(&mutex1);
  printf("thread1 unlocked mutex1 ...\n");
  return NULL;
}

void *thread2_function(void *arg) {
  pthread_mutex_lock(&mutex2);
  printf("thread2 locked mutex2 ...\n");
  sleep(1);
  if (pthread_mutex_trylock(&mutex1) == 0) {
    printf("thread2 locked mutex1 ...\n");
    printf("thread2 working...\n"); // 임계 영역 코드
    pthread_mutex_unlock(&mutex1);
    printf("thread2 unlocked mutex1 ...\n");
  }
  pthread_mutex_unlock(&mutex2);
  printf("thread2 unlocked mutex2 ...\n");
  return NULL;
}
[...]
$ gcc -g  -o  mutex3  mutex3.c -pthread
$ ./mutex3
thread1 locked mutex1 ...
thread2 locked mutex2 ...
thread1 unlocked mutex1 ...
thread2 locked mutex1 ...
thread2 working...
thread2 unlocked mutex1 ...
thread2 unlocked mutex2 ...
 시간순으로 표시하면 다음과 같다.
   thread1    잠금1획득 ... 잠금1해제
   thread2    잠금2획득              잠금1획득 .. 잠금1해제   잠금2해제
```

- pthread_mutex_trylock의 실용적인 예제[8]를 참고한다. 다중 스레드 환경에서 작업 큐가 있고, 각 스레드는 큐에서 작업을 가져와 처리한다고 가정하자. 큐가 다른 스레드에 의해 사용 중일 때 해당 스레드는 큐가 해제될 때까지 기다리며 멈추지 않고, 다른 일을 할 수 있어야 한다. 이와 같은 시나리오에서 사용 될 수 있다.

8) https://www.youtube.com/watch?v=UrTU7ss3LDc

※ pthread 프로그램을 gdb로 디버깅 해보기 위해 아래와 같이 작성한다.

```
// thsleep.c
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
typedef struct {
  int tid;  int s1;  int s2;
} ThreadData;
pthread_mutex_t  mutex = PTHREAD_MUTEX_INITIALIZER;
void* thsleep(void* arg) {
  ThreadData *data = (ThreadData *)arg;
  sleep(data->s1);  // s1 만큼 시간이 지난 이후에 mutex lock을 한다.
  pthread_mutex_lock(&mutex); printf("thread %d lock mutex for %d seconds. \n", data->tid, data
->s2);
  sleep(data->s2); // s2 시간 이후에 unlock 한다.
  pthread_mutex_unlock(&mutex); printf("thread %d unlock mutex \n", data->tid);
  return NULL;
}
int main() {
  pthread_t threads[3];
  ThreadData thread_data[3];
  // 스레드0,1,2는 각각 15초,10초,5초 이후에 mutex lock을 걸로 10초 후에 unlock 한다.
  // 따라서 lock을 획득하는 순서는 스레드 2 -> 1 -> 0 순서임
  thread_data[0].s1 = 15; thread_data[1].s1 = 10; thread_data[2].s1 = 5;
  thread_data[0].s2 = 10; thread_data[1].s2 = 10; thread_data[2].s2 = 10;
  for (int i = 0; i < 3; ++i) {
    thread_data[i].tid = i;
    pthread_create(&threads[i], NULL, thsleep, (void *)&thread_data[i]);
  }
  for (int i = 0; i < 3; ++i) { pthread_join(threads[i], NULL); }
  pthread_mutex_destroy(&mutex);
  return 0;
}
```

```
$ gcc -g -o thsleep  thsleep.c  -pthread
$ ./thsleep
thread 2 lock mutex for 10 seconds.
thread 2 unlock mutex
thread 1 lock mutex for 10 seconds.
thread 1 unlock mutex
thread 0 lock mutex for 10 seconds.
thread 0 unlock mutex
```

```
$ gdb -ex "break main" -ex "run" -ex "set pagination off"  ./thsleep
(gdb) disas thsleep
Dump of assembler code for function thsleep:
[...]
   0x000055555555524a <+33>:    call   0x555555555120 <sleep@plt>
   0x000055555555524f <+38>:    lea    0x2dea(%rip),%rax        # 0x555555558040 <mutex>
   0x0000555555555256 <+45>:    mov    %rax,%rdi
   0x0000555555555259 <+48>:    call   0x555555555130 <pthread_mutex_lock@plt>
   0x000055555555525e <+53>:    mov    -0x8(%rbp),%rax
   0x0000555555555262 <+57>:    mov    0x8(%rax),%edx
[...]
```

```
(gdb) b *0x000055555555525e          pthread_mutex_lock() 직후에 중단점을 설정
Breakpoint 2 at 0x55555555525e: file thsleep.c, line 17.

(gdb) continue
Continuing.
[New Thread 0x7ffff7bff640 (LWP 40047)]
[New Thread 0x7ffff73fe640 (LWP 40048)]
[New Thread 0x7ffff6bfd640 (LWP 40049)]
[Switching to Thread 0x7ffff6bfd640 (LWP 40049)]
    가장 먼저 lock을 획득한 마지막 스레드(Thread 4)가 중단점에 도달함
Thread 4 "thsleep" hit Breakpoint 2, thsleep (arg=0x7fffffffe308) at thsleep.c:17
17        pthread_mutex_lock(&mutex); printf("thread %d lock mutex for %d seconds. \n", data->t
id, data->s2);

(gdb) info thread          스레드 정보를 표시
  Id   Target Id                                       Frame
  1    Thread 0x7ffff7fa6740 (LWP 40042) "thsleep" __futex_abstimed_wait_common64 (private=128,
cancel=true, abstime=0x0, op=265, expected=40047, futex_word=0x7ffff7bff910) at ./nptl/futex-in
ternal.c:57
  2    Thread 0x7ffff7bff640 (LWP 40047) "thsleep" 0x00007ffff7ce57f8 in __GI___clock_nanosleep
(clock_id=clock_id@entry=0, flags=flags@entry=0, req=req@entry=0x7ffff7bfedf0, rem=rem@entry=0x
7ffff7bfedf0) at ../sysdeps/unix/sysv/linux/clock_nanosleep.c:78
  3    Thread 0x7ffff73fe640 (LWP 40048) "thsleep" 0x00007ffff7ce57f8 in __GI___clock_nanosleep
(clock_id=clock_id@entry=0, flags=flags@entry=0, req=req@entry=0x7ffff73fddf0, rem=rem@entry=0x
7ffff73fddf0) at ../sysdeps/unix/sysv/linux/clock_nanosleep.c:78
* 4    Thread 0x7ffff6bfd640 (LWP 40049) "thsleep" thsleep (arg=0x7fffffffe308) at thsleep.c:17

(gdb) continue       다음으로 lock을 획득한 스레드(Thread 3)이 중단점에 도달
Continuing.
thread 2 lock mutex for 10 seconds.
thread 2 unlock mutex
[Thread 0x7ffff6bfd640 (LWP 40049) exited]
[Switching to Thread 0x7ffff73fe640 (LWP 40048)]
[...]
(gdb) backtrace     백트레이스 스택 정보를 확인한다.
#0  thsleep (arg=0x7fffffffe2fc) at thsleep.c:17
#1  0x00007ffff7c94ac3 in start_thread (arg=<optimized out>) at ./nptl/pthread_create.c:442
#2  0x00007ffff7d26850 in clone3 () at ../sysdeps/unix/sysv/linux/x86_64/clone3.S:81

(gdb) bt full    ; local 변수를 포함하여 자세한 정보를 출력
#0  thsleep (arg=0x7fffffffe2fc) at thsleep.c:17
        data = 0x7fffffffe2fc
#1  0x00007ffff7c94ac3 in start_thread (arg=<optimized out>) at ./nptl/pthread_create.c:442
        ret = <optimized out>
        pd = <optimized out>
        out = <optimized out>
[...]
No locals.
; gdb에서는 종료되지 않은 함수를 하나의 frame이라고 함. 각 프레임은 스택에 쌓여있는 함수, 아직
종료되지 않은 함수이며, 각 frame은 caller와 callee의 관계를 구성
(gdb) info frame    ; 프레임 정보를 출력
Stack level 0, frame at 0x7ffff73fde60:
 rip = 0x55555555525e in thsleep (thsleep.c:17); saved rip = 0x7ffff7c94ac3
 called by frame at 0x7ffff73fdf00
```

```
 source language c.
 Arglist at 0x7ffff73fde50, args: arg=0x7fffffffe2fc
 Locals at 0x7ffff73fde50, Previous frame's sp is 0x7ffff73fde60
 Saved registers:
  rbp at 0x7ffff73fde50, rip at 0x7ffff73fde58
(gdb) continue    마지막으로 lock을 획득한 스레드(Thread 2)가 중단점에 도달
Continuing.
thread 1 lock mutex for 10 seconds.
thread 1 unlock mutex
[Thread 0x7ffff73fe640 (LWP 40048) exited]
[Switching to Thread 0x7ffff7bff640 (LWP 40047)]
[...]
(gdb) continue
Continuing.
thread 0 lock mutex for 10 seconds.
thread 0 unlock mutex
[Thread 0x7ffff7bff640 (LWP 40047) exited]
[Inferior 1 (process 40042) exited normally]
```

※ clang 컴파일러 이용. 클랭(clang) 은 LLVM을 백앤드로 사용하는 컴파일러 프론트엔드로 C, C++, Objective-C/C++ 을 지원한다. GCC를 대체하기 위한 오픈소스 프로젝트이다.

```
sudo apt install clang
clang -g -o thread_example thread_example.c -pthread
```

2.3. 조건변수

- 여러 스레드들 간의 실행 순서 결정하거나, 특정 조건을 만족할 때까지 대기하도록 하려면 조건변수(condition variable)을 사용해야 함. 조건변수는 스레드가 변경사항을 지속적으로 모니터링 하는 바쁜 감시(busy-waiting 또는 spinning)가 아닌 조건을 만족할 때까지 대기하도록 만든다.
- pthread_cond_wait 함수에서 mutex가 함께 사용되는 이유는 스레드 간의 동기화와 데드락을 방지한다. 조건이 만족될 때까지 현재 스레드를 블록 상태로 두고, 동시에 mutex를 해제하여 다른 스레드가 공유 자원에 접근할 수 있도록 한다.

```
// signalwait.c
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cond  = PTHREAD_COND_INITIALIZER;
int shared_data = 0;
void* waiter(void* arg) {
  pthread_mutex_lock(&mutex);  // 뮤텍스 잠금
  while (shared_data == 0) {
    printf("waiting...\n");
    // 조건 변수 대기, 뮤텍스 잠금 해제 후 대기, 신호를 받으면 다시 잠금
    pthread_cond_wait(&cond, &mutex);
  }
  printf("OK: shared_data = %d\n", shared_data);
  pthread_mutex_unlock(&mutex);  // 뮤텍스 잠금 해제
  return NULL;
}
void* signaler(void* arg) {
```

```
  sleep(1);
  pthread_mutex_lock(&mutex);  // 뮤텍스 잠금
  shared_data = 1;  // 조건 충족
  pthread_cond_signal(&cond);  // 조건 변수 신호
  pthread_mutex_unlock(&mutex);  // 뮤텍스 잠금 해제
  return NULL;
}
int main() {
  pthread_t thread1, thread2;
  pthread_create(&thread1, NULL, waiter, NULL);
  pthread_create(&thread2, NULL, signaler, NULL);
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);  return 0;
}
$ gcc -g -o signalwait  signalwait.c -pthread
$ ./signalwait
waiting...
OK: shared_data = 1
```

- 조건 변수는 스레드간의 동기화된 실행 순서를 조정할 수 있기 때문에 대표적으로 생산자-소비자(producer-consumer) 모델의 상황에서 유용하게 사용될 수 있다. 데이터를 큐에 입력하는 생산자 스레드와 큐에서 데이터를 가져와 처리하는 소비자 스레드로 구성되는 다양한 사용패턴에 적용 가능하다.

```
// procon.c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define BUFFER_SIZE 5
int buffer[BUFFER_SIZE];
int count = 0;
int in = 0;
int out = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  full = PTHREAD_COND_INITIALIZER;  // 2개의 조건 변수는 버퍼의 2가지 상태
pthread_cond_t  empty = PTHREAD_COND_INITIALIZER;
typedef struct {
  int consid; char *name;
} threadData;
void *producer(void *arg) {
  int item;
  while (1) {
    item = rand() % 100;  // 생산할 아이템
    pthread_mutex_lock(&mutex);  // 버퍼가 가득찼다면 대기
    while (count == BUFFER_SIZE) pthread_cond_wait(&empty, &mutex);

    buffer[in] = item;  // 링버퍼에 입력
    in = (in + 1) % BUFFER_SIZE;
    count++;
    printf("produce item=%d, count=%d\n", item, count);

    pthread_cond_signal(&full);
    pthread_mutex_unlock(&mutex);
    sleep(1);  // 생산자는 1초 간격으로 동작
```

```
  }
  return NULL;
}
void *consumer(void *arg) {
  int item;
  threadData *data = (threadData *)arg;
  while (1) {
    pthread_mutex_lock(&mutex);
    while (count == 0) pthread_cond_wait(&full, &mutex); // 버퍼가 비어 있다면 대기

    item = buffer[out];  // 링버퍼에서 출력
    out = (out + 1) % BUFFER_SIZE;
    count--;
    printf("%s consum item=%d count=%d\n", data->name, item, count);

    pthread_cond_signal(&empty);
    pthread_mutex_unlock(&mutex);
    sleep(4);  // 소비자는 4초 간격으로 동작
  }
  return NULL;
}
int main() {
  pthread_t prod_thread, cons_thread1, cons_thread2;
  threadData thread_data[2];
  thread_data[0].name = "consumer 1";
  thread_data[1].name = "consumer 2"; // 2개의 소비자 스레드를 준비

  pthread_create(&prod_thread, NULL, producer, NULL);
  pthread_create(&cons_thread1, NULL, consumer, (void *)&thread_data[0]);
  pthread_create(&cons_thread2, NULL, consumer, (void *)&thread_data[1]);
  pthread_join(prod_thread, NULL);
  pthread_join(cons_thread1, NULL);
  pthread_join(cons_thread2, NULL);
  return 0;
}
$ gcc -g -o procon    procon.c -pthread
$ ./procon    생산자는 1초간격, 소비자는 4초간격 이므로 어느 순간 버퍼가 가득차게된다.
produce item=83, count=1
consumer 2 consum item=83 count=0
produce item=86, count=1
consumer 1 consum item=86 count=0
produce item=77, count=1
produce item=15, count=2
consumer 2 consum item=77 count=1
produce item=93, count=2
consumer 1 consum item=15 count=1
produce item=35, count=2
produce item=86, count=3
produce item=92, count=4
consumer 2 consum item=93 count=3
produce item=49, count=4
consumer 1 consum item=35 count=3
produce item=21, count=4
produce item=62, count=5 <------ buffer full
consumer 2 consum item=86 count=4
produce item=27, count=5 <------ buffer full
```

```
consumer 1 consum item=92 count=4
produce item=90, count=5 <------ buffer full
consumer 2 consum item=49 count=4
[...]
```

- 조건변수에 관한 구조체는 아래 헤더파일에 정의 되어 있다.

```
// bits/pthreadtypes.h
typedef union  {
  struct __pthread_cond_s __data;
  char __size[__SIZEOF_PTHREAD_COND_T];
  __extension__ long long int __align;
} pthread_cond_t;
// bits/thread-shared-types.h
struct __pthread_cond_s  {
  __atomic_wide_counter __wseq;    대기중인 스레드의 시퀀스 번호를 추적
  __atomic_wide_counter __g1_start;  다음 대기자 그룹의 시작 지점을 추적하는데 사용
  unsigned int __g_refs[2] __LOCK_ALIGNMENT;
  unsigned int __g_size[2];
  unsigned int __g1_orig_size;
  unsigned int __wrefs;        현재 대기 중인 스레드의 수를 나타냄. 대기 참조 수
  unsigned int __g_signals[2];    전송된 신호의 수를 추적
};
```

- 조건변수에 관한 것을 gdb로 관찰

```
$ gdb -ex "break main" -ex "run" -ex "set pagination off"  signalwait
(gdb) disas waiter
Dump of assembler code for function waiter:
[...]
   0x00005555555552a3 <+58>:    lea    0x2dd6(%rip),%rax        # 0x555555558080 <cond>
   0x00005555555552aa <+65>:    mov    %rax,%rdi
   0x00005555555552ad <+68>:    call   0x5555555550f0 <pthread_cond_wait@plt>
   0x00005555555552b2 <+73>:    mov    0x2df8(%rip),%eax        # 0x5555555580b0 <shared_data>
   0x00005555555552b8 <+79>:    test   %eax,%eax
   0x00005555555552ba <+81>:    je     0x55555555528a <waiter+33>
   0x00005555555552bc <+83>:    mov    0x2dee(%rip),%eax        # 0x5555555580b0 <shared_data>
   0x00005555555552c2 <+89>:    mov    %eax,%esi
   0x00005555555552c4 <+91>:    lea    0xd44(%rip),%rax        # 0x55555555600f
[...]
End of assembler dump.
(gdb) break  *0x00005555555552ad        pthread_cond_wait 전에 중단점 설정
Breakpoint 2 at 0x5555555552ad: file signalwait.c, line 15.
(gdb) break  *0x00005555555552bc        while 문 밖에서 중단점 설정
Breakpoint 3 at 0x5555555552b2: file signalwait.c, line 12.
(gdb) continue
Continuing.
[New Thread 0x7ffff7bff640 (LWP 44438)]
[New Thread 0x7ffff73fe640 (LWP 44439)]
[Switching to Thread 0x7ffff7bff640 (LWP 44438)]
Thread 2 "signalwait" hit Breakpoint 3, waiter (arg=0x0) at signalwait.c:12
12        while (shared_data == 0) {
(gdb) print cond       조건변수를 출력
$1 = {__data = {__wseq = {__value64 = 0, __value32 = {__low = 0, __high = 0}}, __g1_start = {__
value64 = 0, __value32 = {__low = 0, __high = 0}}, __g_refs = {0, 0}, __g_size = {0, 0}, __g1_o
```

```
rig_size = 0, __wrefs = 0, __g_signals = {0, 0}}, __size = '\000' <repeats 47 times>, __align =
0}
(gdb) continue
Continuing.
[Thread 0x7ffff73fe640 (LWP 44529) exited]

Thread 2 "signalwait" hit Breakpoint 3, waiter (arg=0x0) at signalwait.c:17
17        printf("OK: shared_data = %d\n", shared_data);
(gdb) print cond
$2 = {__data = {__wseq = {__value64 = 3, __value32 = {__low = 3, __high = 0}}, __g1_start = {__
value64 = 1, __value32 = {__low = 1, __high = 0}}, __g_refs = {0, 0}, __g_size = {0, 0}, __g1_o
rig_size = 4, __wrefs = 0, __g_signals = {0, 0}}, __size = "\003\000\000\000\000\000\000\00
1", '\000' <repeats 23 times>, "\004", '\000' <repeats 14 times>, __align = 3}
```

2.4. 읽기/쓰기 잠금

- 여러 스레드가 공유 데이터에 접근할 때 읽기와 쓰기 작업을 효율적으로 관리하기 위한 방법을 제공함. 한번에 하나의 스레드만 데이터 쓰기 작업을 수행할 수 있으며, 쓰기 작업이 진행 중일때는 모든 읽기 작업이 차단됨. 반대로 읽기 작업의 경우 여러 스레드가 동시에 데이터를 읽을 수 있음.

```c
// rwlock.c
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#define NUM_READERS 3  // 읽기 스레드를 3개 실행
pthread_rwlock_t rwlock;
int shared_data = 0;
void *reader_thread(void *arg) {
  int thread_id = *((int*)arg);
  while (1) {
    pthread_rwlock_rdlock(&rwlock);    // 읽기 락을 설정
    printf("Reader %d read data %d\n", thread_id, shared_data);
    pthread_rwlock_unlock(&rwlock);
    sleep(1);  // 1초에 한번씩 값을 읽는다
  }
  return NULL;
}
void *writer_thread(void *arg) {
  while (1) {
    pthread_rwlock_wrlock(&rwlock);  // 쓰기 락을 설정
    shared_data++; // 2초 마다 1씩 값을 증가 시킨다
    printf("Writer thread: %d\n", shared_data);
    pthread_rwlock_unlock(&rwlock);
    sleep(2);
  }
  return NULL;
}
int main() {
  pthread_t readers[NUM_READERS], writer;
  int thread_ids[NUM_READERS];
  pthread_rwlock_init(&rwlock, NULL); // 락을 초기화
  for (int i = 0; i < NUM_READERS; i++) {
    thread_ids[i] = i;
```

```
    pthread_create(&readers[i], NULL, reader_thread, &thread_ids[i]);
  }
  pthread_create(&writer, NULL, writer_thread, NULL);
  for (int i = 0; i < NUM_READERS; i++) {
    pthread_join(readers[i], NULL);
  }
  pthread_join(writer, NULL);
  pthread_rwlock_destroy(&rwlock);
  return 0;
}
$ gcc -g -o rwlock      rwlock.c -pthread
$ ./rwlock
Reader 0 read data 0
Reader 2 read data 0
Reader 1 read data 0
Writer thread: 1
Reader 2 read data 1
Reader 0 read data 1
Reader 1 read data 1
Writer thread: 2
Reader 0 read data 2
Reader 1 read data 2
Reader 2 read data 2
Reader 1 read data 2
Reader 0 read data 2
Reader 2 read data 2
[...]
```

3. OpenMP 프로그래밍

- OpenMP(Open Multi-Processing, 오픈MP)[9]는 공유 메모리 다중 처리를 위한 프로그래밍 언어 확장 API로, C, C++, Fortran 언어와, 유닉스 및 마이크로소프트 윈도우 플랫폼을 비롯한 여러 플랫폼을 지원함
- OpenMP 아키텍처 리뷰 보드(ARB)[10]는 최초의 API 규격인 포트란 1.0용 OpenMP를 1997년 10월 발표. C/C++용 OpenMP는 1998년 10월에 공개. 2000년 11월에 포트란 버전으로 2.0이 나온 다음 2002년 3월에 C/C++ 규격으로 2.0 버전을 발표. 2005년 5월에 발표된 버전 2.5부터는 C/C++/포트란 규격이 통합됨.
- OpenMP 버전별 차이점

버전	설명
OpenMP 2.5 (2005) GCC 4.2	- 기본 병렬화 기능: 병렬 영역, 작업 분할, 작업 공유, 데이터 환경, 동기화 및 상호 배제 등
OpenMP 3.0 (2008) GCC 4.4	- 동적 작업 생성 및 관리 지원
OpenMP 4.0 (2013) GCC 4.9	- GPU 및 기타 가속기 장치를 위한 디렉티브 추가
OpenMP 4.5 (2015) GCC 6	- Device 메모리 모델 개선: GPU 및 다른 가속기 장치의 메모리 모델 및 관리 개선
OpenMP 5.0 (2018) GCC 9이상	- Task Dependency: 작업 간 의존성 관리 개선
OpenMP 5.2 (2021) GCC 12이상	- Enhanced directives: scan, order, loop 디렉티브의 확장 및 개선.

- OpenMP 핵심 기능 요소

기능	설명
parallel control structures	- 프로그램의 실행 흐름을 제거
work sharing	- 작업을 스레드에 분산시킴
data environment	- 변수의 공유 규칙
synchronization	- 스레드 실행을 조정함 `critical`, `atomic`, `barrier` 지시어
runtime function, env. variables	- 런타임 실행 환경을 제공. `omp_set_num_threads()`, `omp_get_thread_num()`, `OMP_NUM_THREADS`, `OMP_SCHEDULE` 등

9) https://en.wikipedia.org/wiki/OpenMP
10) https://www.openmp.org/about/about-us/

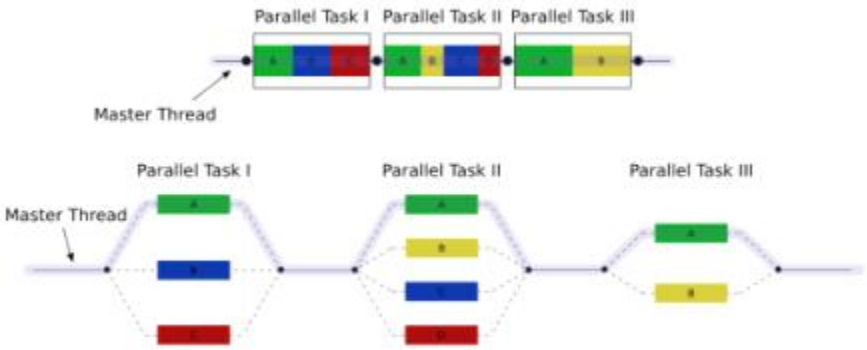


그림 1 Multithreading parallel 실행 개념도

※ GNU OMP 소스코드는 GCC 컴파일러 소스에 포함되어 있다.

```
wget https://ftp.gnu.org/gnu/gcc/gcc-11.3.0/gcc-11.3.0.tar.gz
tar zxvf gcc-11.3.0.tar.gz
cd gcc-11.3.0/libgomp
```

3.1 OpenMP 시작

- GNU의 OpenMP 구현체인 GOMP(GNU Offloading and Multi Processing) 라이브러리[11)]
- GNU 컴파일러 컬랙션 C, C++, Fortran 컴파일러를 위한 OpenMP 구현. 병렬 프로그래밍을 간소화함. GCC 4.2(2007년)에서 OpenMP 2.5가 지원되기 시작
- OpenACC에 대한 지원이 추가되면서 “GNU OpenMP“를 나타내는 대신 “GNU Offloading and Multi-Processing"으로 변경됨.

- GCC 에서 OpenMP를 이용하여 위해서는 컴파일러 옵션 `-fopenmp` 이 필요함

```
// hello_omp.c
#include <stdio.h>
#include <omp.h>

int main() {
  int tid, total_threads;
  #pragma omp parallel        // preprocessor directive
  //_Pragma("omp parallel")   // preprocessor operator
  {
    tid = omp_get_thread_num();
    total_threads = omp_get_num_threads();
    printf("Hello from thread %d/%d\n", tid, total_threads);
  }
  return 0;
}
```

```
$ gcc -g -fopenmp -o hello_omp   hello_omp.c
$ OMP_NUM_THREADS=3   ./hello_omp
Hello from thread 1/3
Hello from thread 2/3
Hello from thread 0/3
```

```
$ ldd hello_omp
        linux-vdso.so.1 (0x00007ffdebe88000)
```

11) https://gcc.gnu.org/projects/gomp/

```
        libgomp.so.1 => /lib/x86_64-linux-gnu/libgomp.so.1 (0x00007e85cc480000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007e85cc200000)
        /lib64/ld-linux-x86-64.so.2 (0x00007e85cc4e3000)
```

※ OMP_NUM_THREADS 를 지정하지 않으면 프로세서의 코어 수를 기본값으로 사용함

※ 지시어(directive) 형식은 다음과 같다 (C, C++, Fortran 언어에 따라서 다를 수 있음)

```
#pragma omp directive-specification [clause [[,] clause]] new-line
또는
_Pragma("omp directive-specification")
```

※ 컴파일된 바이너리는 libgomp.so 공유 라이브러리를 참조하고 있음. [12]

- libgomp를 포함하여 컴파일 하려면 libgomp.a 가 필요하다.

```
$ ldd  hello_omp
        linux-vdso.so.1 (0x00007ffca7590000)
        libgomp.so.1 => /lib/x86_64-linux-gnu/libgomp.so.1 (0x00007e29771a6000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007e2976e00000)
        /lib64/ld-linux-x86-64.so.2 (0x00007e2977209000)
$ gcc -g -fopenmp -o hello_omp.o -c  hello_omp.c
$ gcc  -Wall -Werror -pedantic hello_omp.o  -o hello_omp_with_libgomp  \
  /usr/lib/gcc/x86_64-linux-gnu/11/libgomp.a
$ ldd  hello_omp_with_libgomp
        linux-vdso.so.1 (0x00007ffefefa4000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007e75e2c00000)
        /lib64/ld-linux-x86-64.so.2 (0x00007e75e2ed1000)
```

※ OpenMP버전 확인은 _OPENMP 매크로를 이용하면 알 수 있음

```
// gomp_version.c
#include <stdio.h>
#include <omp.h>
int main() {
  printf("OpenMP version: %d\n", _OPENMP);
  return 0;
}
$ gcc -g -fopenmp -o  gomp_version  gomp_version.c
$ ./gomp_version
OpenMP version: 201511   // OpenMP 4.5를 의미
```

3.2 데이터 공유 구문

- private 구문(clause)[13]은 병렬 영역 내에서 각 스레드가 독립적인 변수를 가지도록 함. 즉, private로 지정된 변수는 각 스레드에서 개별적으로 초기화되며, 다른 스레드와 공유되지 않는다.

```
// private.c
#include <stdio.h>
#include <omp.h>
int main() {
  int tid, total_threads;
  int i;
```

12) https://gcc.gnu.org/onlinedocs/libgomp/

13) https://www.openmp.org/spec-html/5.2/openmpsu37.html#x72-740005.4.3

```
  omp_set_num_threads(2);   //  2개의 스레드가 실행됨
  #pragma omp parallel  private(tid)   // tid는 private 로 선언
  {
    tid = omp_get_thread_num();
    total_threads = omp_get_num_threads();
    for (i = 0; i < 5; i++) {   // 각 스레드가 5번의 printf를 수행
      printf("Hello from thread %d of %d. i=%d\n", tid, total_threads, i);
    }
  }
  printf("program terminated.\n");
  return 0;
}
```

```
$ gcc -g -fopenmp -o private  private.c
$ ./private
Hello from thread 0 of 2. i=0
Hello from thread 0 of 2. i=1
Hello from thread 0 of 2. i=2
Hello from thread 0 of 2. i=3
Hello from thread 0 of 2. i=4
Hello from thread 1 of 2. i=0
program terminated.
 ; 총 10번의 printf 출력을 기대하였으나, thread 1은 i=0만 출력하고 종료됨
  변수 i는 private이 아니며, 공유변수이기 때문
```

```
// private.c
[...]
  #pragma omp parallel private(tid, i)  // i 를 private 로 선언
[...]
```

```
 재컴파일하고 실행하여 결과를 확인. 의도했던 결과를 확인
$ gcc -g -fopenmp -o private  private.c  ; ./private
Hello from thread 1 of 2. i=0
Hello from thread 1 of 2. i=1
Hello from thread 1 of 2. i=2
Hello from thread 1 of 2. i=3
Hello from thread 1 of 2. i=4
Hello from thread 0 of 2. i=0
Hello from thread 0 of 2. i=1
Hello from thread 0 of 2. i=2
Hello from thread 0 of 2. i=3
Hello from thread 0 of 2. i=4
program terminated.
```

※ private이 아닌 변수는 기본적으로 모든 스레드에 공유된다. 그러나, 혼선이 없이 하기 위해 명시적으로 shared **구문**을 사용하는 것이 좋음.

```
#pragma omp parallel  shared(variable_name)
```

※ default 구문을 사용하면 기본 데이터 공유 속성을 설정할 수 있다.

```
default(shared ¦ none)
```

- firstprivate [14]구문은 private와 비슷하지만, 병렬 영역이 시작될 때 변수의 초기값을 복사함. 원본 변수의 초기값이 각 스레드에 복사됨

14) https://www.openmp.org/spec-html/5.2/openmpsu38.html#x73-750005.4.4

```
// private2.c
#include <stdio.h>
#include <omp.h>
int main() {
  int tid, total_threads;
  int i = 10;
  omp_set_num_threads(2);
  #pragma omp parallel private(tid) firstprivate(i)
  {
    tid = omp_get_thread_num();
    total_threads = omp_get_num_threads();
    for (; i < 13; i++) {   // i의 초기값(i=10)이 각 스레드로 복사됨
      printf("Hello from thread %d of %d. i=%d\n", tid, total_threads, i);
    }
  }
  printf("program terminated.\n");
  return 0;
}
$ gcc -g -fopenmp -o private2  private2.c ; ./private2
Hello from thread 0 of 2. i=10
Hello from thread 0 of 2. i=11
Hello from thread 0 of 2. i=12
Hello from thread 1 of 2. i=10
Hello from thread 1 of 2. i=11
Hello from thread 1 of 2. i=12
program terminated.
```

- threadprivate [15]구문은 전역 변수를 각 스레드가 독립적으로 사용할 수 있도록 함. threadprivate 로 선언된 변수는 스레드 간에 독립적인 복사본이 유지되며, 스레드가 종료될 때까지 값이 유지됨.

```
// private3.c
#include <stdio.h>
#include <omp.h>
int total = 10; // total은 전역 변수
#pragma omp threadprivate(total)   // total은 전역변수이지만 각 스레드에 복사본이 유지됨
int main() {
  int tid, total_threads;
  omp_set_num_threads(2);
  #pragma omp parallel private(tid)
  {
    tid = omp_get_thread_num();
    total_threads = omp_get_num_threads();
    for (; total > 0; total--) {
      printf("Hello from thread %d of %d. total=%d(%p)\n", tid, total_threads, total, &total);
    }
  }
  printf("program terminated. tid=%d total=%d(%p)\n", tid, total, &total);
  return 0;
}
$ gcc -g -fopenmp -o private3  private3.c
$ ./private3
Hello from thread 1 of 2. total=10(0x729ba5bff63c) // 스레드1에서 10,9,8,...,1 출력
Hello from thread 1 of 2. total=9(0x729ba5bff63c)
```

15) https://www.openmp.org/spec-html/5.2/openmpse24.html#x67-690005.2

```
Hello from thread 1 of 2. total=8(0x729ba5bff63c)
[...]
Hello from thread 1 of 2. total=1(0x729ba5bff63c)
Hello from thread 0 of 2. total=10(0x729ba5feb7bc) // 스레드0 에서 10,9,8,...,1 출력
Hello from thread 0 of 2. total=9(0x729ba5feb7bc)  // 스레드 0,1에서 변수의 주소가 다름
Hello from thread 0 of 2. total=8(0x729ba5feb7bc)
[...]
Hello from thread 0 of 2. total=1(0x729ba5feb7bc)
program terminated. tid=0 total=0(0x729ba5feb7bc)
```

- lastprivate 지시어[16]는 병렬 영역에서 마지막으로 실행된 스레드의 값을 변수에 반영함. 병렬 영역 내에서 변수의 최종 상태를 병렬 영역 외부에서 유지하고자 할 때 사용

```
// private4.c
#include <stdio.h>
#include <omp.h>
int main() {
  int tid, total_threads;
  int i, last_value = 0;
  #pragma omp parallel private(tid, i)
  {
    tid = omp_get_thread_num();
    total_threads = omp_get_num_threads();
    #pragma omp lastprivate(last_value)
    for (i = 0; i < 10; i++) {
      printf("Hello from thread %d of %d. i=%d\n", tid, total_threads, i);
      last_value = tid * 100 + i;
        //  스레드 0 : last_value = 9
        //  스레드 1 : last_value = 109
    }
  }
  printf("program terminated. tid=%d last_value=%d\n", tid, last_value);
  return 0;
}
```

```
$ gcc -g -fopenmp -o private4  private4.c
$ ./private4
Hello from thread 1 of 2. i=0
[...]
Hello from thread 0 of 2. i=9
program terminated. tid=0 last_value=9
 ; last_value의 최종 값은 스레드 0,1 의 종료 순서에 따라 다르며, 9 또는 109 가 될 수 있음
```

3.3 작업 공유 지시어

- section 지시어[17]는 sections 지시어와 함께 사용되며, 코드 블럭을 여러 스레드로 나누어 병렬로 실행할 수 있도록 함. 각 섹션 블록은 독립적으로 실행된다. sections 블록이 끝나면 암시적인 동기화가 이루어 진다. 모든 section 블록이 완료될 때까지 다른 스레드는 대기하게 된다.

```
// share1.c
#include <stdio.h>
#include <omp.h>
```

16) https://www.openmp.org/spec-html/5.2/openmpsu39.html#x74-760005.4.5
17) https://www.openmp.org/spec-html/5.0/openmpsu37.html

```
int main() {
  int a = 0;
  #pragma omp parallel
  {
    if (omp_get_thread_num() == 0) {
      int total_threads = omp_get_num_threads();
      printf("total threads %d\n", total_threads);
    }
    #pragma omp sections firstprivate(a) lastprivate(a)
    {                   // 3개의 섹션이므로 3개의 스레드가 필요함
      #pragma omp section
      {
        printf("section 1: a 초기값 = %d\n", a);
        a = 1;
        printf("section 1: a 수정값 = %d\n", a);
      }
      #pragma omp section
      {
        printf("section 2: a 초기값 = %d\n", a);
        a = 2;
        printf("section 2: a 수정값 = %d\n", a);
      }
      #pragma omp section
      {
        printf("section 3: a 초기값 = %d\n", a);
        a = 3;
        printf("section 3: a 수정값 = %d\n", a);
      }
    }  // sections 끝 부분에서 동기화가 이루어진다.
  }
  printf("a 최종값 = %d\n", a);
  return 0;
}
```

```
$ gcc -g -fopenmp -o share1  share1.c
$ ./share1
total threads 2  <-- section 개수는 3개이나 스레드는 2개 뿐임
section 2: a 초기값 = 0
section 2: a 수정값 = 2
section 3: a 초기값 = 2 <---- section 3은 section 2가 종료된 이후 실행되었다
section 3: a 수정값 = 3
section 1: a 초기값 = 0
section 1: a 수정값 = 1
a 최종값 = 3  <-- 마지막으로 실행된 스레드의 값이 반영됨 (lastprivate 효과)
```

※ sections 지시어에 nowait 구문을 사용하면 암시적인 동기화는 사용되지 않는다.

- single 지시어는 특정 코드 블럭을 단일 스레드에서만 실행한다. 코드 블럭을 실행하는 스레드가 반드시 마스터 스레드일 필요는 없다. 블록의 끝 부분에 암시적 장벽(implicit barrier)가 있어 동기화가 이루어 진다(단, nowait 가 명시되지 않은 경우).

```
// share2.c
#include <stdio.h>
#include <omp.h>
```

```
int main() {
  omp_set_num_threads(3);  // 3개의 스레드를 실행
  #pragma omp parallel
  {
    int thread_id = omp_get_thread_num();
    printf("This is executed by all threads. Thread ID: %d\n", thread_id);
    #pragma omp single nowait
    {      // 하나의 스레드만 이 블럭을 실행
      for (int i ; i < 10000000; i++) ;
      int total_threads = omp_get_num_threads(); // 총 스레드 개수를 출력
      printf("total threads %d. Thread ID=%d\n", total_threads, thread_id);
    }
    printf("done. Thread ID: %d\n", thread_id);
  }
  return 0;
}
```

```
$ ./share2
This is executed by all threads. Thread ID: 2
This is executed by all threads. Thread ID: 1 <-- 0번과 1번 스레드는 barrier 없이 먼저 종료됨
done. Thread ID: 1                                                    (nowait 구문)
This is executed by all threads. Thread ID: 0
done. Thread ID: 0
total threads 3. Thread ID=2  <---- 마스터가 아닌 2번 스레드에서 single 블럭이 실행됨
done. Thread ID: 2
```

- 작업공유 지시어는 블록의 실행을 별도의 작업 단위로 나누고 팀의 스레드가 작업을 나누어 실행하도록 함. 포트란에서는 workshare 지시어를 사용하고 C에서는 for 지시어를 사용함

```
// share3.c
#include <stdio.h>
#include <omp.h>
#define N 10
int main() {
  int i;
  int array[N], result[N];
  for (i = 0; i < N; i++) {   array[i] = i + 1;  }
  #pragma omp parallel shared(array, result) private(i)
  {
    int tid = omp_get_thread_num();
    #pragma omp for     // for 블럭은 반드시 parallel 블럭 안에서 사용해야 함
    for (i = 3; i < N; i++) {
      printf("tid=%d i=%d\n", tid, i);
      result[i] = array[i] * array[i];
      //if (i > 5) break; // for loop 안에서 break 문은 사용이 금지된다.
    }
  }
  printf("Squared elements of array:\n");
  for (i = 0; i < N; i++) {    printf("%d ", result[i]);  }
  printf("\n");
  return 0;
}
```

```
$ ./share3
tid=0 i=3    <-- 스레드0. 루프는 2개의 스레드에 나뉘어 실행됨
tid=0 i=4
```

```
tid=0 i=5
tid=0 i=6
tid=1 i=7   <-- 스레드 1
tid=1 i=8
tid=1 i=9
Squared elements of array:
0 0 0 16 25 36 49 64 81 100
```

※ for 블럭 안에서 break 문은 사용이 금지된다. 컴파일시 에러 발생함.

```
$ gcc -g -fopenmp -o share3 share3.c
share3.c: In function ‘main’:
share3.c:19:18: error: break statement used with OpenMP for loop
   19 |       if (i > 5) break;
      |                  ^~~~~
```

3.4 병렬 실행 제어

- parallel 지시어는 중첩되어 사용될 수 있지만, 기본적으로는 비활성화 되어 있음. 중첩된 병렬 영역을 활성화 하려면 omp_set_nested 함수를 사용할 수 있다.

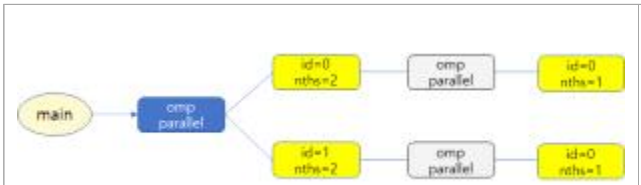
```
// par1.c
#include <stdio.h>
#include <omp.h>
int main (void) {
#ifdef NESTED
  omp_set_nested(1);
#endif
  #pragma omp parallel
  {
    printf ("thread ID=%d num_thds=%d level=%d\n",
      omp_get_thread_num(), omp_get_num_threads(), omp_get_level());
    #pragma omp parallel
    {
      printf ("thread ID=%d num_thds=%d level=%d\n",
        omp_get_thread_num(), omp_get_num_threads(), omp_get_level());
    }
  }
  return 0;
}
```

```
$ gcc -g -fopenmp -o par1 par1.c
$ gcc -g -fopenmp -o par1_nested par1.c -DNESTED
```

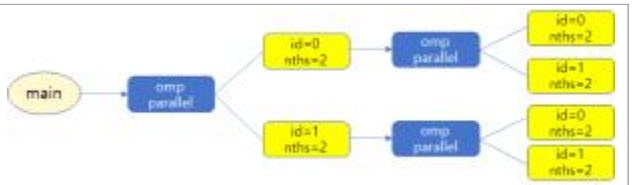
```
$ OMP_NUM_THREADS=2  ./par1
thread ID=0 num_thds=2 level=1  내부에 중첩된 parallel 블럭은 1개 스레드만 실행됨
thread ID=0 num_thds=1 level=2
thread ID=1 num_thds=2 level=1
thread ID=0 num_thds=1 level=2
$ OMP_NESTED=true OMP_NUM_THREADS=2  ./par1
thread ID=0 num_thds=2 level=1
thread ID=0 num_thds=2 level=2
thread ID=1 num_thds=2 level=2  내부에 중첩된 parallel 블럭도 2개씩 실행됨
thread ID=1 num_thds=2 level=1
thread ID=0 num_thds=2 level=2
thread ID=1 num_thds=2 level=2
```

```
$ OMP_NUM_THREADS=2  ./par1_nested
thread ID=0 num_thds=2 level=1  OMP_NESTED 환경변수없이 omp_set_nested(1)로 처리. 같은 결과
thread ID=0 num_thds=2 level=2
thread ID=1 num_thds=2 level=2
thread ID=1 num_thds=2 level=1
thread ID=0 num_thds=2 level=2
thread ID=1 num_thds=2 level=2
```

※ 위 결과를 도식화 하면 아래와 같다.



중첩을 허용하지 않음



중첩을 허용함

※ OMP_NESTED 는 OpenMP의 ICV(internal control variables, 내부 제어 변수)[18] 중의 하나임

※ omp_set_max_active_levels() 함수로 최대 중첩될 수 있는 parallel 블럭의 수를 제한할 수 있음

- barrier 지시어는 모든 스레드가 해당 지점에 도달할 때까지 대기하도록 한다. critical 지시어는 해당 코드 블럭을 한 번에 하나의 스레드만 실행할 수 있도록 한다.

```
// par2.c
#include <stdio.h>
#include <unistd.h>
#include <omp.h>
#define N 10
int main() {
  int i;
  int array[N], result[N];
  for (i = 0; i < N; i++) { array[i] = i + 1; result[i] = 0; } // 1,2,3,4,...,10
  #pragma omp parallel shared(array, result) private(i)
  {
    #pragma omp for nowait      // nowait 로 처리함에 주의
    for (i = 0; i < N; i++) {
      result[i] = array[i] * array[i];  // 1,4,9,16,25,..,100
    }
    #pragma omp barrier   // 모든 스레드가 여기서 대기
    #pragma omp critical
    {  // 출력 내용이 스레드간에 섞이지 않게 하기 위해 critial 로 지정함
      for (i = 0; i < N; i++) printf("%d ", result[i]);
      printf(" <== tid=%d\n", omp_get_thread_num());
    }
    #pragma omp for     // i는 1부터 시작, 이웃한 요소(i-1)와 더하기
    for (i = 1; i < N; i++) {      result[i] += result[i-1];    }
  }
  for (i = 0; i < N; i++) printf("%d ", result[i]);  // 최종 결과를 출력
  printf(" <= result \n");
  return 0;
}
$ ./par2
```

18) https://www.openmp.org/spec-html/5.0/openmpse13.html

```
1 4 9 16 25 36 49 64 81 100  <== tid=0
1 5 14 30 55 91 49 64 81 100  <== tid=1
1 5 14 30 55 91 140 204 285 385  <= result
 또는
$ ./par2
1 4 9 16 25 36 49 64 81 100  <== tid=1
1 4 9 16 25 36 85 149 230 330  <== tid=0
1 5 14 30 55 91 85 149 230 330  <= result
```

- barrier 까지는 동기화가 되어 문제가 없지만, 두번째 omp for 블럭은 스레드 실행순서에 따라서 결과가 달라짐. 그림 참조:

- reduction(축소) 구문[19] 여러 스레드가 각각의 결과를 계산한 후 이를 하나의 결과로 합치는 데 사용. 배열의 합, 최대, 최소 값을 구하는 코드는 다음과 같다. reduction-identiier 는 +, -, *, &, ¦, ^, &&, ¦¦ 등이 가능함

```
// reduction.c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <limits.h>
int main() {
  const int size = 1000;
  int *arr = (int *)malloc(size * sizeof(int));
  int sum = 0;
  int min_value = INT_MAX;
  int max_value = INT_MIN;

  for (int i = 0; i < size; i++) {
    arr[i] = i % 99;
  }

  #pragma omp parallel for reduction(+:sum) reduction(min:min_value) reduction(max:max_value)
  for (int i = 0; i < size; ++i) {
    sum += arr[i];
    if (arr[i] < min_value) min_value = arr[i];
    if (arr[i] > max_value) max_value = arr[i];
  }
```

19) https://www.openmp.org/spec-html/5.0/openmpsu107.html#x140-5800002.19.5

```
  printf("sum=%d\n", sum);
  printf("min=%d\n", min_value);
  printf("max=%d\n", max_value);
  free(arr);
  return 0;
}
$ gcc -g -fopenmp -o reduction reduction.c ; ./reduction
sum=48555
min=0
max=98
```

※ OpenMP가 내부적으로 리덕션을 처리하는 원리는 다음과 같다: ① 리덕션이 적용된 변수는 각 스레드마다 로컬 복사본이 생성됨. ② 각 스레드는 독립적으로 작업을 수행함. ③ 병렬 영역이 끝날때 지정된 연산자에 따라 모든 스레드의 로컬 결과를 하나의 공유 변수로 취합함. 이 과정에서 암시적인 동기화가 이루어 짐.

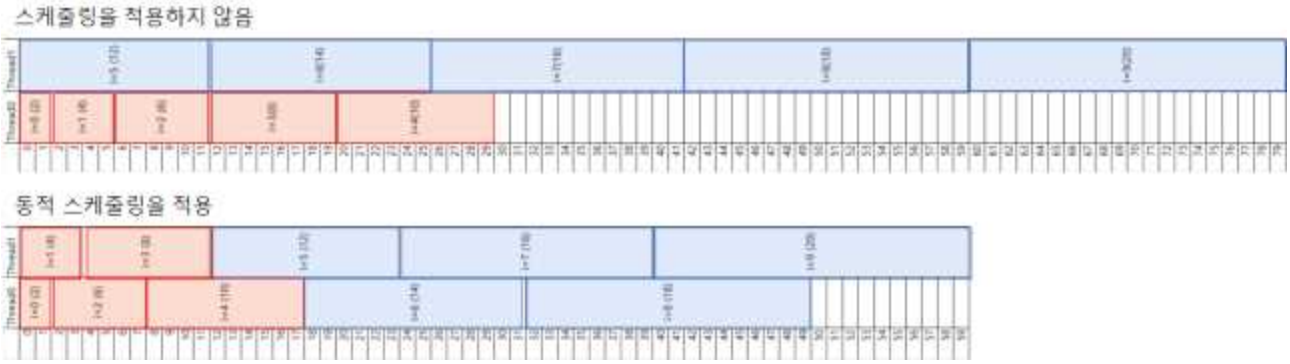
- schedule 구문[20]은 작업을 스레드에 동적으로 분산시키는 방법을 설정한다. 작업을 일정한 크기의 청크(chunk)로 나누고 각 스레드가 청크를 완료할 때마다 새로운 청크를 할당 받도록 함. 작업량이 고르지 않은 경우에 유용하며, 스레드간 부하 불균형을 줄임.

```
// sched1.c
#include <stdio.h>
#include <omp.h>
#include <unistd.h>
#include <time.h>
#define N 10
#define CHUNKSIZE 1
int main() {
  int i;
  time_t start_time = time(NULL);
  #pragma omp parallel  private(i)
  {
    int sec;
    #pragma omp for schedule(dynamic, CHUNKSIZE)
    for (i = 0; i < N; i++) {
      time_t t = time(NULL);
      sec = (i+1) * 2; // 2,4,6,8,...
    printf("t=%ld i=%d thread=%d worktime=%d \n", t-start_time, i, omp_get_thread_num(), sec);
      sleep(sec);
    }
  }
  time_t t = time(NULL);
  printf("t=%ld done. \n", t-start_time);
  return 0;
}
$ ./sched1   스케줄인을 적용하지 않음          $ ./sched1  스케쥴링을 적용
t=0 i=0 thread=0 worktime=2                  t=0 i=0 thread=0 worktime=2
t=0 i=5 thread=1 worktime=12                 t=0 i=1 thread=1 worktime=4
t=2 i=1 thread=0 worktime=4                  t=2 i=2 thread=0 worktime=6
t=6 i=2 thread=0 worktime=6                  t=4 i=3 thread=1 worktime=8
t=12 i=6 thread=1 worktime=14                t=8 i=4 thread=0 worktime=10
```

20) https://www.openmp.org/spec-html/5.0/openmpse49.html#x288-20520006.1

```
t=12 i=3 thread=0 worktime=8                    t=12 i=5 thread=1 worktime=12
t=20 i=4 thread=0 worktime=10                   t=18 i=6 thread=0 worktime=14
t=26 i=7 thread=1 worktime=16                   t=24 i=7 thread=1 worktime=16
t=42 i=8 thread=1 worktime=18                   t=32 i=8 thread=0 worktime=18
t=60 i=9 thread=1 worktime=20                   t=40 i=9 thread=1 worktime=20
t=80 done.  총 실행시간 80초                    t=60 done.  총 실행시간 60초
```

- 위 차이를 그림으로 나타내면 다음과 같다. 스케쥴링을 적용하지 않은 경우 0-4번 작업은 0번 스레드에, 5~9번 작업은 1번 스레드에 고정적으로 할당되므로 불균형이 발생한다.



※ 스케쥴링 종류(kind)를 runtime으로 지정할 경우 런타임에 OMP_SCHEDULE 환경변수에 의해 스케쥴링 방법을 변경할수 있다. omp_set_schedule() 호출을 통해서도 변경할 수 있음.

```
// sched2.c
...
    #pragma omp for schedule(runtime)
    for (i = 0; i < N; i++) {
...
```

```
$ gcc -g -fopenmp -o sched2 sched2.c
$ OMP_SCHEDULE="static" ./sched2
$ OMP_SCHEDULE="dynamic,1" ./sched2
$ OMP_SCHEDULE="dynamic,2" ./sched2
```

```
형식: void omp_set_schedule(omp_sched_t kind, int chunk_size);
```

4. OpenACC

4.1 OpenACC 개요

- OpenACC(open accelerators)[21]는 가속기를 위한 프로그래밍 모델로 Cray, CAPS Enterprise, NVIDIA, PGI가 개발함(2011년 발표). 이 표준은 이기종 CPU/GPU 시스템의 병렬 프로그래밍을 단순하게 만들기 위해 설계된 것[22]
- OpenMP와 유사하게 지시문(`#pragma acc`)을 사용하여 코드를 병렬화함. 병렬화 코드를 지시하는 `kernels`문과 데이터 이동을 지시하는 `data copyin/copyout` 을 사용,
- 병렬화 수준에 따라, gang, worker, vector 수준을 지원. ① Gang-level은 가장 상위 수준의 병렬화로 특정 루프를 gang 구문으로 지정하면 분산된 자원(GPU에서는 블럭의 개념)에 분배된다. 분산된 gang 작업은 독립적으로 수행되며, 총 실행할 gang의 개수는 동적으로 결정된다. ② Worker-level 병렬성은 gang 내부에서 발생하며, gang내의 worker들 간에 루프 반복 작업을 분배함 ③ Vector-level 병렬성은 worker내에서 SIMD 수행을 위한 것임.

4.2 GNU GCC와 OpenACC

- GNU GCC 컴파일러에서도 GPU와 같은 가속기 장치를 프로그래밍 하기 위한 표준으로 OpenACC를 지원함. Nvidia PTX(nvptx)[23]와 AMD의 GCN[24], CDNA[25]와 같은 장치를 지원함. GCC 10~13는 OpenACC 2.6을 지원함. GCC 14부터는 OpenACC 2.7일부 기능을 지원.[26]

- GNU 컴파일러에서 OpenACC를 사용하기 위해서는 `-fopenacc` 옵션을 사용함. Nvidia PTX 지원을 위해서 패키지를 설치해 준다.

```
$ sudo apt-get install  nvptx-tools  gcc-offload-nvptx
$ dpkg-query  -L  nvptx-tools  gcc-offload-nvptx    설치된 파일 확인
```

- 테스트를 위한 샘플 프로그램. 참고[27]

```
// gv1.c
#include <stdio.h>
#include <openacc.h>
#define N 1000
int main() {
  float a[N], b[N], c[N];
  for (int i = 0; i < N; i++) {
    a[i] = i * 1.0f;
    b[i] = i * 2.0f;
  }
  #pragma acc parallel loop
  for (int i = 0; i < N; i++) {
    c[i] = a[i] + b[i];
  }
  for (int i = 0; i < 10; i++) {
    printf("c[%d] = %f\n", i, c[i]);
```

21) https://www.openacc.org/
22) https://en.wikipedia.org/wiki/OpenACC
23) https://en.wikipedia.org/wiki/Parallel_Thread_Execution
24) https://en.wikipedia.org/wiki/Graphics_Core_Next
25) https://en.wikipedia.org/wiki/CDNA_(microarchitecture)
26) https://gcc.gnu.org/wiki/OpenACC
27) https://gcc.gnu.org/wiki/OpenACC/Quick%20Reference%20Guide

```
  }
  return 0;
}

  '-fcf-protection=full' is not supported  관련 에러를 방지하기 위해
$ gcc -fopenacc -foffload=nvptx-none -fcf-protection=none -o gv1  gv1.c
  다음과 같이 환경변수를 설정하고 테스트
$ export ACC_DEVICE_TYPE=nvidia
$ export ACC_DEVICE_NUM=0
$ export GOMP_DEBUG=1
$ ./gv1
OpenACC version: 201711
GOACC_parallel_keyed: mapnum=3, hostaddrs=0x7ffef87af550, size=0x59a93d184010, kinds=0x59a93d18
4028
libgomp: device type nvidia not supported
$ export ACC_DEVICE_TYPE=host  ./gv1
```

4.3 CUDA 드라이버 설치

- 테스트를 위해 Amazon EC2에서 Tesla V100이 장착된 서버를 사용함
- CUDA 드라이버를 설치

```
$ sudo apt update
$ sudo apt install alsa-utils  ubuntu-drivers-common

$ lspci -v ¦ grep -i nvidia
00:1e.0 3D controller: NVIDIA Corporation GV100GL [Tesla V100 SXM2 16GB] (rev a1)
        Subsystem: NVIDIA Corporation GV100GL [Tesla V100 SXM2 16GB]
        Kernel modules: nvidiafb
$ ubuntu-drivers  devices
== /sys/devices/pci0000:00/0000:00:1e.0 ==
modalias : pci:v000010DEd00001DB1sv000010DEsd00001212bc03sc02i00
vendor   : NVIDIA Corporation
model    : GV100GL [Tesla V100 SXM2 16GB]
driver   : nvidia-driver-470 - distro non-free
driver   : nvidia-driver-545 - distro non-free
driver   : nvidia-driver-535 - distro non-free recommended <------- recommended 를 설치
driver   : nvidia-driver-470-server - distro non-free
driver   : nvidia-driver-535-server - distro non-free
driver   : nvidia-driver-390 - distro non-free
driver   : nvidia-driver-418-server - distro non-free
driver   : nvidia-driver-450-server - distro non-free
driver   : xserver-xorg-video-nouveau - distro free builtin
$ sudo apt install nvidia-driver-535
[...]
$ apt list --installed ¦ grep -i nvidia     설치된 패키지 목록확인
$ lsmod ¦ grep nvidia        관련 커널 모듈 확인
$ nvidia-smi
Thu Aug  1 00:54:46 2024
+-----------------------------------------------------------------------------------------+
¦ NVIDIA-SMI 535.183.01             Driver Version: 535.183.01   CUDA Version: 12.2     ¦
¦-----------------------------------------+----------------------+----------------------+
¦ GPU  Name                 Persistence-M ¦ Bus-Id        Disp.A ¦ Volatile Uncorr. ECC ¦
```

```
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla V100-SXM2-16GB           Off | 00000000:00:1E.0 Off |                    0 |
| N/A   33C    P0              39W / 300W |      0MiB / 16384MiB |      1%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+
[...]
```

- CUDA 드라이버를 설치후 위에서 GCC로 컴파일한 프로그램을 실행하여 확인한다.

```
$ export ACC_DEVICE_TYPE=nvidia
$ export ACC_DEVICE_NUM=0
$ export GOMP_DEBUG=1
$ ./gv1
OpenACC version: 201711
GOACC_parallel_keyed: mapnum=3, hostaddrs=0x7ffc04d84470, size=0x5de0b51c9010, kinds=0x5de0b51c
9028
GOMP_NVPTX_JIT: <Not defined>
Loading:
---
// BEGIN PREAMBLE
.version 3.1
.target sm_35
.address_size 64
[...]
---
Linking
Link complete: 0.000000ms
Link log info    : 0 bytes gmem
info    : Function properties for 'main$_omp_fn$0':
info    : used 22 registers, 32 stack, 0 bytes smem, 360 bytes cmem[0], 0 bytes lmem
  GOMP_OFFLOAD_openacc_exec: prepare mappings
 warp_size=32, block_size=1024, dev_size=80, cpu_size=2048
 default dimensions [160,32,32]
cuOccupancyMaxPotentialBlockSize: grid = 2560, block = 32
  nvptx_exec: kernel main$_omp_fn$0: launch gangs=5120, workers=1, vectors=32
  nvptx_exec: kernel main$_omp_fn$0: finished
c[0] = 0.000000
[...]
```

4.4 NVIDIA HPC SDK 설치

- NVIDIA HPC SDK 를 설치한다. 참조[28]. CUDA 최신 버전 12.5와 함께 설치

```
  Linux x86_64 Ubuntu (apt)를 선택.  CUDA version 12.5
$ curl https://developer.download.nvidia.com/hpc-sdk/ubuntu/DEB-GPG-KEY-NVIDIA-HPC-SDK | sudo g
  pg --dearmor -o /usr/share/keyrings/nvidia-hpcsdk-archive-keyring.gpg
$ echo 'deb [signed-by=/usr/share/keyrings/nvidia-hpcsdk-archive-keyring.gpg] https://develope
  r.download.nvidia.com/hpc-sdk/ubuntu/amd64 /' | sudo tee /etc/apt/sources.list.d/nvhpc.list
$ sudo apt-get update -y
$ sudo apt-get install -y nvhpc-24-7   버전 2024.7월을 의미
```

28) https://docs.nvidia.com/hpc-sdk/archive/

```
$ dpkg-query -L nvhpc-24-7     설치된 파일을 확인   /opt/nvidia/hpc_sdk/
  OpenACC 샘플 코드는 /opt/nvidia/hpc_sdk/Linux_x86_64/24.7/examples/OpenACC/samples/ 에 존재
환경 변수를 설정 하기
export NVARCH=`uname -s`_`uname -m`
export NVCOMPILERS=/opt/nvidia/hpc_sdk
export MANPATH=$MANPATH:$NVCOMPILERS/$NVARCH/24.7/compilers/man
export PATH=$NVCOMPILERS/$NVARCH/24.7/compilers/bin:$PATH
$ nvc --version        컴파일러 버전을 확인
nvc 24.7-0 64-bit target on x86-64 Linux -tp broadwell
NVIDIA Compilers and Tools
Copyright (c) 2024, NVIDIA CORPORATION & AFFILIATES.  All rights reserved.
$ ncv --help     도움말
$ nvprof --version
nvprof: NVIDIA (R) Cuda command line profiler
Copyright (c) 2012 - 2024 NVIDIA Corporation
Release version 12.5.82 (21)
$ nvaccelinfo   디바이스 정보 확인
CUDA Driver Version:           12020
NVRM version:                  NVIDIA UNIX x86_64 Kernel Module  535.183.01  Sun May 12 19:39:1
5 UTC 2024
Device Number:                 0
Device Name:                   Tesla V100-SXM2-16GB
Device Revision Number:        7.0
Global Memory Size:            16935682048
Number of Multiprocessors:     80
[...]
```

- 설치 확인용 테스트 프로그램

```
// testnvc.c
#include <stdio.h>
#define N 1000
int array[N];
int main() {
#pragma acc parallel loop copy(array[0:N])
   for(int i = 0; i < N; i++) {
      array[i] = 3.0;
   }
   printf("Success!\n");
}
$ nvc -acc -gpu=cc70 testnvc.c  -o testnvc
$ ./test
Success!
$ ldd testnvc
    linux-vdso.so.1 (0x00007ffef5187000)
    libacchost.so => /opt/nvidia/hpc_sdk/Linux_x86_64/24.7/compilers/lib/libacchost.so (0x0000751904400000)
    libaccdevaux.so => /opt/nvidia/hpc_sdk/Linux_x86_64/24.7/compilers/lib/libaccdevaux.so (0x0000751904000000)
    libaccdevice.so => /opt/nvidia/hpc_sdk/Linux_x86_64/24.7/compilers/lib/libaccdevice.so (0x0000751903c00000)
    libcudadevice.so => /opt/nvidia/hpc_sdk/Linux_x86_64/24.7/compilers/lib/libcudadevice.so (0x0000751903800000)
    libnvomp.so => /opt/nvidia/hpc_sdk/Linux_x86_64/24.7/compilers/lib/libnvomp.so (0x0000751902600000)
    libnvcpumath.so => /opt/nvidia/hpc_sdk/Linux_x86_64/24.7/compilers/lib/libnvcpumath.so (0x0000751902000000)
    libnvc.so => /opt/nvidia/hpc_sdk/Linux_x86_64/24.7/compilers/lib/libnvc.so (0x0000751901c00000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x0000751901800000)
    libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007519046ef000)
    libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x0000751904319000)
    /lib64/ld-linux-x86-64.so.2 (0x000075190471b000)
    libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007519046ea000)
```

```
    libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007519046e3000)
```

4.5 백터 덧셈 예제

- GPU를 이용한 백터 덧셈 프로그램. OpenACC Getting Started Guide[29]를 참고한다.

```
// vecadd.c
#include <stdio.h>
#include <stdlib.h>
void vecaddgpu( float *restrict r, float *a, float *b, int n ){
  #pragma acc kernels loop copyin(a[0:n],b[0:n]) copyout(r[0:n])
  for ( int i = 0; i < n; ++i ) r[i] = a[i] + b[i];
}
int main ( int argc, char* argv[] ) {
  int n; // vector length
  float * a; // input vector 1
  float * b; // input vector 2
  float * r; // output vector
  float * e; // expected output values
  int i, errs;
  if ( argc > 1 ) n = atoi( argv[1] ); else n = 100000; // default vector length
  if ( n <= 0 ) n = 100000;
  a = (float*)malloc( n*sizeof(float) );
  b = (float*)malloc( n*sizeof(float) );
  r = (float*)malloc( n*sizeof(float) );
  e = (float*)malloc( n*sizeof(float) );
  for (i = 0; i < n; ++i) {
    a[i] = (float)(i+1);
    b[i] = (float)(1000*i);
  }
  vecaddgpu( r, a, b, n ); // GPU에서 계산
  for( i = 0; i < n; ++i ) e[i] = a[i] + b[i]; // 호스트에서 계산
  errs = 0;
  for (i = 0; i < n; ++i) { // 두가지 결과를 비교
    if (r[i] != e[i]) ++errs;
  }
  printf( "%d errors found\n", errs );
  return errs;
}
```

```
$ nvc -acc vecadd.c -o vecadd
$ nvc -acc -fast -Minfo  vecadd.c -o vecadd
$ ./vecadd
0 errors found
```

```
  GPU에서 커널이 실행될 때 메시지를 출력하기  NVCOMPILER_ACC_NOTIFY=1
$ NVCOMPILER_ACC_NOTIFY=1 ./vecadd
launch CUDA kernel  file=/home/ubuntu/linux-drill/nvhpc/vecadd.c function=vecaddgpu line=6 devi
ce=0 threadid=1 num_gangs=782 num_workers=1 vector_length=128 grid=782 block=128
 그리드 사이즈는 782, 스레드 블록의 크기는 128임을 알수 있다.
0 errors found

  NVCOMPILER_ACC_NOTIFY=3 으로 설정하여 데이터 전송관련 정보도 함께 표시
```

29) https://docs.nvidia.com/hpc-sdk/compilers/openacc-gs/index.html

```
$ NVCOMPILER_ACC_NOTIFY=3 ./vecadd
upload CUDA data  file=/home/ubuntu/linux-drill/nvhpc/vecadd.c function=vecaddgpu line=4 device
=0 threadid=1 variable=a[:n] bytes=400000
upload CUDA data  file=/home/ubuntu/linux-drill/nvhpc/vecadd.c function=vecaddgpu line=4 device
=0 threadid=1 variable=b[:n] bytes=400000
launch CUDA kernel  file=/home/ubuntu/linux-drill/nvhpc/vecadd.c function=vecaddgpu line=6 devi
ce=0 threadid=1 num_gangs=782 num_workers=1 vector_length=128 grid=782 block=128
download CUDA data  file=/home/ubuntu/linux-drill/nvhpc/vecadd.c function=vecaddgpu line=6 devi
ce=0 threadid=1 variable=r[:n] bytes=400000
0 errors found
```

```
  NVCOMPILER_ACC_TIME=1 로 설정하여 데이터 전송과 계산에 소요된 시간을 파악한다.
$ export LD_LIBRARY_PATH=$NVCOMPILERS/Linux_x86_64/24.7/cuda/12.5/extras/CUPTI/lib64
$ NVCOMPILER_ACC_TIME=1  ./vecadd
0 errors found

Accelerator Kernel Timing data
/home/ubuntu/linux-drill/nvhpc/vecadd.c
  vecaddgpu  NVIDIA  devicenum=0
    time(us): 608     총 608 microseconds를 사용
    4: compute region reached 1 time
        6: kernel launched 1 time
            grid: [782]  block: [128]
            elapsed time(us): total=35 max=35 min=35 avg=35
    4: data region reached 2 times
        4: data copyin transfers: 2   2번의 데이터 전송 host to device
             device time(us): total=219 max=110 min=109 avg=109
        6: data copyout transfers: 1   1번의 데이터 전송 device to host
             device time(us): total=389 max=389 min=389 avg=389
```

- kernels loop 과 data copyin/copyout을 분리하여 다음과 같이 작성가능

```
// vecadd2.c
#include <stdio.h>
#include <stdlib.h>
void vecaddgpu( float *restrict r, float *a, float *b, int n ){
  #pragma acc kernels loop present(r, a, b)
  for ( int i = 0; i < n; ++i ) r[i] = a[i] + b[i];
}
int main ( int argc, char* argv[] ) {
  int n; // vector length
[...]
  #pragma acc data copyin(a[0:n],b[0:n]) copyout(r[0:n])
  {
    vecaddgpu( r, a, b, n ); // GPU에서 계산
  }
  for( i = 0; i < n; ++i ) e[i] = a[i] + b[i]; // 호스트에서 계산
  errs = 0;
  for (i = 0; i < n; ++i) { // 두가지 결과를 비교
    if (r[i] != e[i]) ++errs;
  }
  printf( "%d errors found\n", errs );
  return errs;
}
```

```
$ nvc -acc -fast -Minfo  vecadd2.c -o vecadd2
$ NVCOMPILER_ACC_TIME=1  ./vecadd2
0 errors found
Accelerator Kernel Timing data
/home/ubuntu/linux-drill/nvhpc/vecadd2.c
  vecaddgpu  NVIDIA  devicenum=0
    time(us): 5    계산에 걸린 시간
    4: compute region reached 1 time
        6: kernel launched 1 time
            grid: [782]  block: [128]
             device time(us): total=5 max=5 min=5 avg=5
            elapsed time(us): total=722 max=722 min=722 avg=722
    4: data region reached 2 times
/home/ubuntu/linux-drill/nvhpc/vecadd2.c
  main  NVIDIA  devicenum=0
    time(us): 568   데이터 전송에 걸린 시간
    27: data region reached 2 times
        27: data copyin transfers: 2
             device time(us): total=196 max=103 min=93 avg=98
        29: data copyout transfers: 1
             device time(us): total=372 max=372 min=372 avg=372
```

5. 벤치마크 응용 프로그램

5.1 FFT

- FFTW(Fastest Fourier Transform in the West)[30]는 C언어로 작성된 고속 푸리에 변환(FFT) 라이브러리임.
- 주파수 성분이 혼합된 신호를 FFT 변환을 수행하는 코드 예제. 푸리에 변환의 입력 및 출력을 각각 실수부와 허수부를 출력하고, 이것을 gnuplot을 이용하여 그래프 이미지를 생성한다.

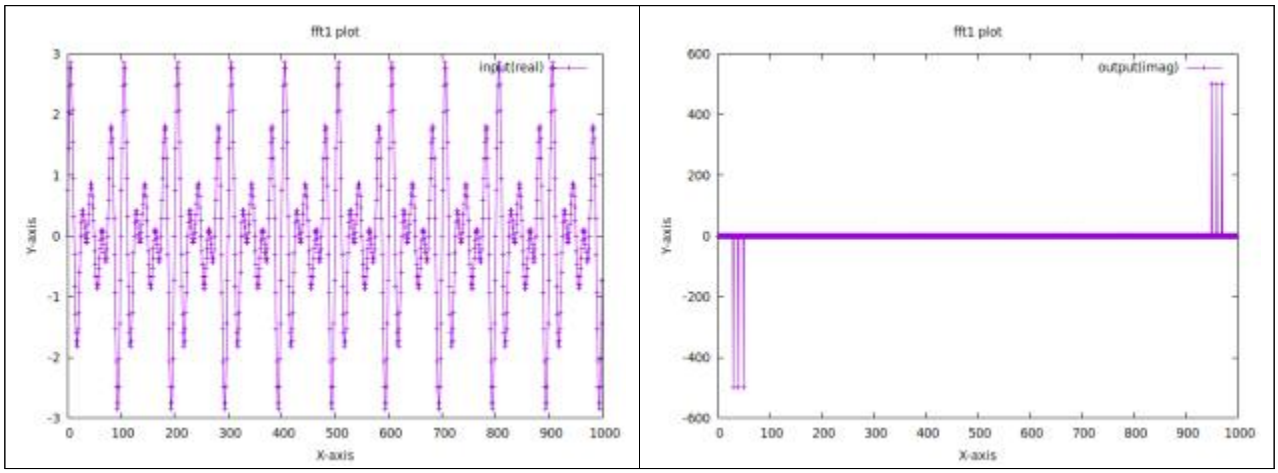
```
// fft1.c
#include <stdio.h>
#include <math.h>
#include <complex.h>
#include <fftw3.h>

#define N 1000  // 데이터 포인트 수
#define PI 3.14159265358979323846
void  initialize(fftw_complex *vec, int n) {
    // 주파수 성분 설정
    double freq1 = 50.0;  // 첫 번째 주파수 (Hz)
    double freq2 = 30.0; // 두 번째 주파수 (Hz)
    double freq3 = 40.0; // 세 번째 주파수 (Hz)
    double sample_rate = 1000.0; // sampling rate
    for (int i = 0; i < N; i++) {
        double t = (double)i / sample_rate;
        double signal = sin(2*PI * freq1 * t) + sin(2*PI * freq2 * t) + sin(2*PI * freq3 * t);
        vec[i] = signal + 0.0 * I;
    }
}
int main() {
    fftw_complex *in, *out;
    fftw_plan p;
    // 입력과 출력을 위한 메모리 할당
    in = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * N);
    out = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * N);
    initialize(in, N);
    // FFTW 계획 생성
    p = fftw_plan_dft_1d(N, in, out, FFTW_FORWARD, FFTW_ESTIMATE);
    fftw_execute(p); // FFT 실행
    for (int i = 0; i < N; i++) {
        printf("%d\t%f\t%f\t%f\t%f\n", i, creal(in[i]), cimag(in[i]), creal(out[i]), cimag(out
[i]) );
    }
    fftw_destroy_plan(p);
    fftw_free(in);
    fftw_free(out);
    return 0;
}
$ gcc -c fft1.c -I/usr/local/fftw/include
$ gcc -o fft1 fft1.o -lfftw3 -lm -L/usr/local/fftw/lib
$ ./fft1      출력은 순서대로 입력의 실수부, 허수부, 출력의 실수부 허수부
```

30) https://www.fftw.org/

```
0   0.000000   0.000000   0.000000   0.000000
1   0.745088   0.000000   -0.000000  -0.000000
2   1.437663   0.000000   -0.000000  0.000000
3   2.029391   0.000000   0.000000   -0.000000
[...]
998 -1.437663  0.000000   -0.000000  -0.000000
999 -0.745088  0.000000   -0.000000  0.000000
```

```
# fft1.plot
set terminal pngcairo enhanced color font 'Arial,10'
set output 'fft1.png'
set title 'fft1 plot'
set xlabel 'X-axis'
set ylabel 'Y-axis'
plot 'fft1.txt' using 1:2 with linespoints title 'Column 2', \
  '' using 1:5 with linespoints title 'Column 3'
```

```
$ gnuplot fft1.plot
```



- FFTW 라이브러리를 사용하지 않고 DFT(discrete Fourier transform)를 수행하는 코드

```
// fft2mp.c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
#define PI 3.14159265358979323846
#ifndef SIZE
#define  SIZE  100000
#endif
void DFT(double *in_real, double *in_imag, double *out_real, double *out_imag, int N) {
  #pragma omp parallel
  {
    #pragma omp for
    for (int k = 0; k < N; k++) {
      out_real[k] = 0;
      out_imag[k] = 0;
      for (int n = 0; n < N; n++) {
        double angle = 2 * PI * k * n / N;
        out_real[k] += in_real[n] * cos(angle) + in_imag[n] * sin(angle);
        out_imag[k] += -in_real[n] * sin(angle) + in_imag[n] * cos(angle);
      }
```

```
    }
  }
}
void initialize(double *real, double *imag, int n) {
  // 주파수 성분 설정
  double freq1 = 20.0;
  double freq2 = 30.0;
  double freq3 = 40.0;
  double sample_rate = 1000.0; // sampling rate

  // 입력 데이터 초기화 (3개의 주파수 성분 혼합)
  for (int i = 0; i < n; i++) {
    double t = (double)i / sample_rate;
    double signal = sin(2*PI * freq1 * t) + sin(2*PI * freq2 * t) + sin(2*PI * freq3 * t);
    real[i] = signal; // real part
    imag[i] = 0; // imag is zero
  }
}
int main() {
  int N = SIZE;
  double *in_real, *in_imag, *out_real, *out_imag;
  double start_time, end_time;
  in_real = (double *)malloc(SIZE * sizeof(double));
  in_imag = (double *)malloc(SIZE * sizeof(double));
  out_real = (double *)malloc(SIZE * sizeof(double));
  out_imag = (double *)malloc(SIZE * sizeof(double));
  initialize(in_real, in_imag, N);

  start_time = omp_get_wtime();
  DFT(in_real, in_imag, out_real, out_imag, N);
  end_time = omp_get_wtime();
  fprintf(stderr, "Elapsed time: %f seconds\n", end_time - start_time);
  for (int i = 0; i < N; i++) {
    printf("%d\t%f\t%f\t%f\t%f\n", i, in_real[i], in_imag[i], out_real[i], out_imag[i]);
  }
  return 0;
}
```

스레드 개수를 변경하면서 실행시간을 측정

```
$ gcc -g -fopenmp -o fft2mp  fft2mp.c -lm -DSIZE=10000
$ OMP_NUM_THREADS=1  ./fft2mp > out1
Elapsed time: 9.011626 seconds
$ OMP_NUM_THREADS=2  ./fft2mp > out2
Elapsed time: 4.533218 seconds
$ OMP_NUM_THREADS=4  ./fft2mp > out4
Elapsed time: 2.259968 seconds
$ OMP_NUM_THREADS=8  ./fft2mp > out8
Elapsed time: 1.883881 seconds
```

- OpenACC 버전

```
// fft2acc.c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
```

```
#define PI 3.14159265358979323846
#ifndef SIZE
#define  SIZE  100000
#endif
void DFT(double *in_real, double *in_imag, double *out_real, double *out_imag, int N) {
    #pragma acc parallel loop
    for (int k = 0; k < N; k++) {
      out_real[k] = 0;
      out_imag[k] = 0;
      for (int n = 0; n < N; n++) {
        double angle = 2 * PI * k * n / N;
        out_real[k] += in_real[n] * cos(angle) + in_imag[n] * sin(angle);
        out_imag[k] += -in_real[n] * sin(angle) + in_imag[n] * cos(angle);
      }
    }
}
[...]
```

```
$ nvc -acc -fast -Minfo   fft2acc.c  -o fft2acc   -DSIZE=10000
$ NVCOMPILER_ACC_TIME=3   ./fft2acc  > out
Elapsed time: 0.175378 seconds

Accelerator Kernel Timing data
/home/ubuntu/linux-drill/openacc/fft2acc.c
  DFT  NVIDIA  devicenum=0
    time(us): 5,569
    12: compute region reached 1 time
        12: kernel launched 1 time
            grid: [79]  block: [128]
             device time(us): total=5,338 max=5,338 min=5,338 avg=5,338
            elapsed time(us): total=5,386 max=5,386 min=5,386 avg=5,386
    12: data region reached 2 times
        12: data copyin transfers: 2
             device time(us): total=62 max=35 min=27 avg=31
        22: data copyout transfers: 2
             device time(us): total=169 max=91 min=78 avg=84
```

5.2 매트릭스 곱셈

- 매트릭스 곱셈을 위한 예제 코드

```
// matmul.c
#include <stdio.h>
#include <omp.h>
#ifndef N
#define N 100
#endif
int A[N][N];
int B[N][N];
int C[N][N];
void matmul(int a[N][N], int b[N][N], int result[N][N]) {
  #pragma omp parallel for
  for (int i = 0; i < N; i++) {
```

```
    for (int j = 0; j < N; j++) {
      result[i][j] = 0;
      for (int k = 0; k < N; k++) {
        result[i][j] += a[i][k] * b[k][j];
      }
    }
  }
}
void printMatrix(int matrix[N][N]) {
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      printf("%d ", matrix[i][j]);
    }
    printf("\n");
  }
}
void initialize(int mat[N][N]) {
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      mat[i][j] = 1;
    }
  }
}
int main() {
  double start_time, end_time;
  initialize(A);  initialize(B);
  start_time = omp_get_wtime(); // 시간측정
  matmul(A, B, C);
  end_time = omp_get_wtime(); // 시간측정
  fprintf(stderr, "Elapsed time: %f seconds\n", end_time - start_time);
  printf("Result matrix is:\n");
  printMatrix(C);
  return 0;
}
```

```
gcc -g -fopenmp -o matmul  matmul.c -DN=1000
    스레드 개수를 변경하면서 실행시간을 측정
 OMP_NUM_THREADS=1   ./matmul > out
Elapsed time: 5.392456 seconds
$ OMP_NUM_THREADS=2   ./matmul > out
Elapsed time: 2.718656 seconds
$ OMP_NUM_THREADS=4   ./matmul > out
Elapsed time: 1.363754 seconds
$ OMP_NUM_THREADS=8   ./matmul > out
Elapsed time: 1.516795 seconds
```

- OpenACC 버전

```
// matmulacc.c
#include <stdio.h>
#include <omp.h>
#ifndef N
#define N 100
#endif
int A[N][N];
```

```
int B[N][N];
int C[N][N];
void matmul(int a[N][N], int b[N][N], int result[N][N]) {
  #pragma acc parallel loop
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      result[i][j] = 0;
      for (int k = 0; k < N; k++) {
        result[i][j] += a[i][k] * b[k][j];
      }
    }
  }
}
[...]
```

```
nvc -acc -fast -Minfo  -o matmulacc  matmulacc.c -DN=1000
NVCOMPILER_ACC_TIME=1  ./matmulacc > out
Elapsed time: 0.502468 seconds

Accelerator Kernel Timing data
/home/ubuntu/linux-drill/openacc/matmulacc.c
  matmul  NVIDIA  devicenum=0
    time(us): 337,968
    10: compute region reached 1 time
        10: kernel launched 1 time
            grid: [8]  block: [128]
             device time(us): total=333,442 max=333,442 min=333,442 avg=333,442
            elapsed time(us): total=333,497 max=333,497 min=333,497 avg=333,497
    10: data region reached 2 times
        10: data copyin transfers: 2
             device time(us): total=1,229 max=616 min=613 avg=614
        19: data copyout transfers: 1
             device time(us): total=3,297 max=3,297 min=3,297 avg=3,297
```

5.3 야코비 반복법

- Jacobi Iteration[31]은 선형대수학에서 연립 일차 방정식을 푸는 알고리즘 중 하나임. 선형 방정식 Ax=b 에서 A는 n×n 행렬이고, x와 b는 각각 n차원의 백터임.

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \qquad \begin{aligned} a_{11}x_1 + a_{12}x_2 &= b_1 \\ a_{21}x_1 + a_{22}x_2 &= b_2 \end{aligned} \qquad x_1 = \frac{b_1 - a_{12}x_2}{a_{11}},\ x_2 = \frac{b_2 - a_{21}x_1}{a_{22}}$$

  여기서 벡터 (x1, x2)의 초기값을 임의로 설정하고, 위 점화식을 이용하여 벡터의 수열을 계산해 나가서 일정 오차 범위내로 수렴할때까지 반복한다.

$$\begin{aligned} Ax &= b \\ A &= L + D + U \\ (L + D + U)x &= b \\ Dx + (L + U)x &= b \end{aligned} \qquad \begin{aligned} Dx &= -(L + U)x + b \\ D^{-1}Dx &= -D^{-1}(L + U)x + D^{-1}b \\ x^{(k+1)} &= -D^{-1}(L + U)x^{(k)} + D^{-1}b \end{aligned}$$

31) https://en.wikipedia.org/wiki/Jacobi_method

- 야코비 반복법을 수행하는 프로그램을 파이썬으로 작성함. numpy의 행렬 연산 함수를 활용

```
# jacobi1.py
import numpy as np
ITERATION_LIMIT = 1000

# initialize the matrix
A = np.array([[10., -1., 2., 0.],
              [-1., 11., -1., 3.],
              [2., -1., 10., -1.],
              [0.0, 3., -1., 8.]])
# initialize the RHS vector
b = np.array([6., 25., -11., 15.])

# prints the system
print("System:")
for i in range(A.shape[0]):
    row = [f"{A[i, j]}*x{j + 1}" for j in range(A.shape[1])]
    print(f'{" + ".join(row)} = {b[i]}')
print()

L = np.tril(A, k=-1)
U = np.triu(A, k=1)
diagaonal_elements = 1/np.diag(A)
Dinv = np.diag(diagaonal_elements)
DinvLU = np.dot(Dinv, L+U)

def jacobi_step(x, b):
  result = np.dot(DinvLU, x) + np.dot(Dinv, b)
  return result

x_new = x = np.zeros_like(b)
for it_count in range(ITERATION_LIMIT):
  print(f"Iteration {it_count}: {x}")

  x_new = jacobi_step(x, b) // iternation을 반복

  if np.allclose(x, x_new, atol=1e-10, rtol=0.):  // 오차 범위 이내가 되면 stop
    break

  x = x_new

print("Solution: ")
print(x)
error = np.dot(A, x) - b
print("Error:")
print(error)
```

```
$ sudo apt install python3-pip
$ pip3 install numpy
$ python3 jacobi1.py
System:
10.0*x1 + -1.0*x2 + 2.0*x3 + 0.0*x4 = 6.0
-1.0*x1 + 11.0*x2 + -1.0*x3 + 3.0*x4 = 25.0
2.0*x1 + -1.0*x2 + 10.0*x3 + -1.0*x4 = -11.0
0.0*x1 + 3.0*x2 + -1.0*x3 + 8.0*x4 = 15.0
```

```
Iteration 0: [0. 0. 0. 0.]
Iteration 1: [ 0.6         2.27272727 -1.1         1.875     ]
Iteration 2: [ 1.04727273  1.71590909 -0.80522727  0.88522727]
Iteration 3: [ 0.93263636  2.05330579 -1.04934091  1.13088068]
Iteration 4: [ 1.01519876  1.95369576 -0.96810863  0.97384272]
Iteration 5: [ 0.9889913   2.01141473 -1.0102859   1.02135051]
Iteration 6: [ 1.00319865  1.99224126 -0.99452174  0.99443374]
Iteration 7: [ 0.99812847  2.00230688 -1.00197223  1.00359431]
Iteration 8: [ 1.00062513  1.9986703  -0.99903558  0.99888839]
Iteration 9: [ 0.99967415  2.00044767 -1.00036916  1.00061919]
Iteration 10: [ 1.0001186   1.99976795 -0.99982814  0.99978598]
Iteration 11: [ 0.99994242  2.00008477 -1.00006833  1.0001085 ]
Iteration 12: [ 1.00002214  1.99995896 -0.99996916  0.99995967]
Iteration 13: [ 0.99998973  2.00001582 -1.00001257  1.00001924]
Iteration 14: [ 1.00000409  1.99999268 -0.99999444  0.9999925 ]
Iteration 15: [ 0.99999816  2.00000292 -1.0000023   1.00000344]
Iteration 16: [ 1.00000075  1.99999868 -0.99999899  0.99999862]
Iteration 17: [ 0.99999967  2.00000054 -1.00000042  1.00000062]
Iteration 18: [ 1.00000014  1.99999976 -0.99999982  0.99999975]
Iteration 19: [ 0.99999994  2.0000001  -1.00000008  1.00000011]
Iteration 20: [ 1.00000003  1.99999996 -0.99999997  0.99999995]
Iteration 21: [ 0.99999999  2.00000002 -1.00000001  1.00000002]
Iteration 22: [ 1.          1.99999999 -0.99999999  0.99999999]
Iteration 23: [ 1.  2. -1.  1.]
Iteration 24: [ 1.  2. -1.  1.]
Iteration 25: [ 1.  2. -1.  1.]
Iteration 26: [ 1.  2. -1.  1.]
Iteration 27: [ 1.  2. -1.  1.]
Iteration 28: [ 1.  2. -1.  1.]
Solution:
[ 1.  2. -1.  1.]  // =x 이고, 실제로 Ax=b임을 확인 할 수 있다
Error:
[ 3.95797173e-10 -7.29649230e-10  5.13315612e-10 -5.86034332e-10]
```

※ NVIDIA에서 제공하는 code sample은 아래 주소에서 확인할 수 있다.

```
git clone https://github.com/NVIDIA-developer-blog/code-samples
cd posts/002-openacc-example
```

- 야코비 반복법을 C언어로 작성한 예제는 다음과 같다.

```
// jacobi2.c
#include <stdio.h>
#include <math.h>

#define N 4
#define ITERATION_LIMIT 1000
void triangle(float in[N][N], float out[N][N], int upper) { // L,U 성분을 구하기
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      if (upper)
        if (j > i) out[i][j] = in[i][j]; else out[i][j] = 0;
      else
        if (j < i) out[i][j] = in[i][j]; else out[i][j] = 0;
```

```
    }
  }
}

void inv_diag(float in[N][N], float out[N][N]) {  // 대각행열의 역수 (inverse diagonal)
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      if (j == i) out[i][j] = 1/in[i][j]; else out[i][j] = 0;
    }
  }
}

void matrix_vector_multiply(float result[N], float matrix[N][N], float vector[N]) {
  for (int i = 0; i < N; i++) {  // matrix x vector
    result[i] = 0;
    for (int j = 0; j < N; j++) {
      result[i] += matrix[i][j] * vector[j];
    }
  }
}

void vector_zero(float a[N]) {
  for (int i = 0; i < N; i++) a[i] = 0;
}
void copy_vector(float dst[N], float src[N]) {
  for (int i = 0; i < N; i++) {
    dst[i] = src[i];
  }
}
void matrix_add(float result[N][N], float a[N][N], float b[N][N]) {
  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
      result[i][j] = a[i][j] + b[i][j];
}
void matrix_change_sign(float out[N][N]) {
  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
      out[i][j] = -out[i][j];
}
void matrix_matrix_multiply(float result[N][N], float a[N][N], float b[N][N]) {
  for (int i = 0; i < N; i++) {     // 매트릭스 곱셉
    for (int j = 0; j < N; j++) {
      result[i][j] = 0;
      for (int k = 0; k < N; k++) {
        result[i][j] += a[i][k] * b[k][j];
      }
    }
  }
}
void vector_add(float result[N], float a[N], float b[N]) {
  for (int i = 0; i < N; i++) {
    result[i] = a[i] + b[i];
  }
}
```

```
int are_vectors_close(float *vector1, float *vector2, int size, float rtol, float atol) {
  for (int i = 0; i < size; i++) {   // 두 백차의 오차를 계산
    float diff = fabs(vector1[i] - vector2[i]);
    float max_tol = atol + rtol * fabs(vector2[i]);
    if (diff > max_tol) {
      return 0;
    }
  }
  return 1;
}

void jacobi_step(float result[N], float x[N], float b[N], float DinvLU[N][N], float Dinv[N][N])
{
  float tmp1[N], tmp2[N], tmp3[N];
  matrix_vector_multiply(tmp1, DinvLU, x);  // - D^-1 (L+U) x
  matrix_vector_multiply(tmp2, Dinv, b);    //   D^-1 b
  vector_add(result, tmp1, tmp2);
}

void printMatrix(float matrix[N][N]) {
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      printf("%f ", matrix[i][j]);
    }
    printf("\n");
  }
}
void printVector(float vec[N]) {
  printf("[ ");
  for (int i = 0; i < N; i++) {
    printf("%f ", vec[i]);
  }
  printf("] \n");
}

int main() {
  float A[N][N] = {{10., -1., 2., 0.},
              {-1., 11., -1., 3.},
              {2., -1., 10., -1.},
              {0.0, 3., -1., 8.}};
  float b[N] = {6., 25., -11., 15.};

  float B[N][N];
  float C[N][N];
  float L[N][N];
  float U[N][N];
  float LU[N][N];
  float Dinv[N][N];
  float DinvLU[N][N];
  float x[N];
  float x_new[N];

  triangle(A, U, 1);
  triangle(A, L, 0);
```

```
  inv_diag(A, Dinv);
  matrix_add(LU, U, L);
  matrix_matrix_multiply(DinvLU, Dinv, LU);
  matrix_change_sign(DinvLU);

  printf("Dinv : \n"); printMatrix( Dinv );
  printf("DinvLU : \n"); printMatrix( DinvLU );

  vector_zero(x);
  printf("x : "); printVector( x );

  for (int i = 0; i < ITERATION_LIMIT; i++) {
    printf("i=%d x", i); printVector( x );
    jacobi_step(x_new, x, b, DinvLU, Dinv);

    if ( are_vectors_close(x, x_new, N, 0., 1e-10) ) break;

    copy_vector(x, x_new);
  }
  printf("solution: "); printVector( x_new );

  return 0;
}
```

```
$ gcc -o jacobi2  jacobi2.c
$  ./jacobi2
Dinv :
0.100000 0.000000 0.000000 0.000000
0.000000 0.090909 0.000000 0.000000
0.000000 0.000000 0.100000 0.000000
0.000000 0.000000 0.000000 0.125000
DinvLU :
-0.000000 0.100000 -0.200000 -0.000000
0.090909 -0.000000 0.090909 -0.272727
-0.200000 0.100000 -0.000000 0.100000
-0.000000 -0.375000 0.125000 -0.000000
x : [ 0.000000 0.000000 0.000000 0.000000 ]
i=0 x[ 0.000000 0.000000 0.000000 0.000000 ]
i=1 x[ 0.600000 2.272727 -1.100000 1.875000 ]
i=2 x[ 1.047273 1.715909 -0.805227 0.885227 ]
i=3 x[ 0.932636 2.053306 -1.049341 1.130881 ]
i=4 x[ 1.015199 1.953696 -0.968109 0.973843 ]
i=5 x[ 0.988991 2.011415 -1.010286 1.021351 ]
i=6 x[ 1.003199 1.992241 -0.994522 0.994434 ]
i=7 x[ 0.998129 2.002307 -1.001972 1.003594 ]
i=8 x[ 1.000625 1.998670 -0.999036 0.998888 ]
i=9 x[ 0.999674 2.000448 -1.000369 1.000619 ]
i=10 x[ 1.000119 1.999768 -0.999828 0.999786 ]
i=11 x[ 0.999942 2.000085 -1.000068 1.000108 ]
i=12 x[ 1.000022 1.999959 -0.999969 0.999960 ]
i=13 x[ 0.999990 2.000016 -1.000013 1.000019 ]
i=14 x[ 1.000004 1.999993 -0.999994 0.999992 ]
i=15 x[ 0.999998 2.000003 -1.000002 1.000003 ]
```

```
i=16 x[ 1.000001 1.999999 -0.999999 0.999999 ]
i=17 x[ 1.000000 2.000000 -1.000000 1.000001 ]
i=18 x[ 1.000000 2.000000 -1.000000 1.000000 ]
i=19 x[ 1.000000 2.000000 -1.000000 1.000000 ]
i=20 x[ 1.000000 2.000000 -1.000000 1.000000 ]  // 결과는 파이썬으로 작성한 경우와 같다.
solution: [ 1.000000 2.000000 -1.000000 1.000000 ]
```

[참고자료]

- GNU Offloading and Multi-Processing Project (GOMP) https://gcc.gnu.org/projects/gomp/
- OpenMP tutorials from LLNL https://hpc-tutorials.llnl.gov/openmp/
- OpenMP API version 5.2, Nov. 2021 https://www.openmp.org/spec-html/5.2/openmp.html
- The GNU Debugger : 3. Stack 정보 분석하기 https://sonseungha.tistory.com/454
- Compiler Explorer https://godbolt.org/
- OpenMP - Tutorials & Articles https://www.openmp.org/resources/tutorials-articles/
- Introduction to OpenMP, Youtube https://www.youtube.com/watch?v=nE-xN4Bf8XI&list=PLLX-Q6B8xqZ8n8bwjGdzBJ25X2utwnoEG&index=1
- Programming with POSIX Threads, David R. Butenhof(1997) https://www.informit.com/store/programming-with-posix-threads-9780201633924
- OpenMP API Examples Version 4.5.0, Nov. 2016 https://www.openmp.org/wp-content/uploads/openmp-examples-4.5.0.pdf
- OpenACC Programming and Best Practices Guide https://openacc-best-practices-guide.readthedocs.io/en/latest/
- Getting Started with OpenACC, Jul. 2015. https://developer.nvidia.com/blog/getting-started-openacc/
- OpenACC Example - NVIDIA Technical Blog https://developer.nvidia.com/blog/openacc-example-part-1/
- NVIDIA CUDA Archived Documentation https://docs.nvidia.com/cuda/archive/
- COMP Superscala (COMPSs) https://www.bsc.es/research-and-development/software-and-apps/software-list/comp-superscalar/documentation

이 보고서는 2024년도 한국과학기술정보연구원(KISTI)의
기본사업으로 수행된 연구입니다.
과제번호: (KISTI) K24-L02-C06-S01
과제명: 초고성능컴퓨팅 공동활용을 위한 통합 환경 개발 및 구축

비매품/무료
95560
9 788929 416966
ISBN 978-89-294-1696-6 (PDF)