Paralelno programiranje s OpenMP bibliotekom

Živković, Ivan

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Ivan Živković

PARALELNO PROGRAMIRANJE S OPENMP BIBLIOTEKOM

Diplomski rad

Zagreb, veljača 2024.

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Ivan Živković

PARALELNO PROGRAMIRANJE S OPENMP BIBLIOTEKOM

Diplomski rad

Voditelj rada:
prof. dr. sc. Mladen Jurak

Zagreb, veljača 2024.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Zahvaljujem se obitelji na podršci i razumijevanju tijekom mog školovanja.
Zahvaljujem se i prijateljima na druženjima, suradnji i podršci.
Posebno zahvaljujem mentoru prof. dr. sc. Mladenu Juraku na savjetima, pomoći i strpljenju tijekom izrade mog diplomskog rada.

Sadržaj

Uvod

Tijekom dvadesetog stoljeća, većina računala dostupna za osobnu upotrebu imala su samo jedno-jezgrene procesore. Paralelno programiranje je bilo ograničeno na računanje visokih performansi u raznim znanstvenim i tehničkim područjima. Međutim, razvojem višejezgrenih procesora i njihovom većom dostupnošću, paralelno programiranje je postalo važan dio razvoja aplikacija za upotrebu na osobnim računalima. U zadnjih 15-ak godina, osim procesora, za paralelno programiranje su se počele koristiti i grafičke kartice sa tisućama jezgri, osobito u područjima strojnog učenja, kriptografije i raznih simulacija iz područja fizike, zahvaljujući mogućnosti značajnog poboljšanja performansi u rješavanju nekih specifičnih problema.

U paralelnom programiranju postoje tri glavna načina organizacije glavne memorije; dijeljena memorija (eng. *shared memory*), raspodijeljena memorija (eng. *distributed memory*) i raspodijeljena dijeljena memorija (eng. *distributed shared memory*). Kod dijeljene memorije, svi podatci se nalaze u jednom adresnom prostoru i dostupni su svim procesorima, dok kod raspodijeljene memorije svaki procesor ima svoj adresni prostor i s drugim procesorima mora komunicirati porukama. U ovom radu dati ćemo pregled paralelnog programiranja s dijeljenom memorijom, koje je ujedno i najčešći oblik paralelnog programiranja za osobna računala, budući da velika većina osobnih računala ima samo jedan višejezgreni procesor sa dijeljenom glavnom memorijom. Također ćemo uvesti OpenMP standard, prikazati njegove primjene u paralelnom programiranju s dijeljenom memorijom i dotaknuti se nekih njegovih prednosti i mana. Koristit ćemo programski jezik C++ (implementacije OpenMP standarda postoje za C, C++ i Fortran), koji je vrlo raširen u programiranju za sustave visokih performansi, ugrađene sustave, ali i osobna računala.

Poglavlje 1

Programska paradigma i programsko okruženje

1.1 Paralelno programiranje s dijeljenom memorijom

Klasifikacija (paralelnih) računala

Različita računala možemo klasificirati po tome koliko se instrukcija izvršava nad koliko podataka u isto vrijeme. Najčešće se koristi klasifikacija koju je predstavio Flynn [5], koju ćemo ovdje i navesti.

Definicija 1.1. ***SISD računalo*** *(od eng. Single-Instruction stream, Single-Data stream) je računalo kod kojeg se jedan niz instrukcija izvršava nad jednim nizom podataka.*

Računala ove klase nemaju mogućnost paralelizacije.

Definicija 1.2. ***SIMD računalo*** *(od eng. Single-Instruction stream, Multiple-Data stream) je računalo kod kojeg se jedan niz instrukcija izvršava nad više nizova podataka.*

Važan primjer ovakve procesorske jedinice su grafičke kartice (GPU - eng. *Graphics Processing Unit*), koje imaju vrlo velik broj jezgri koje sve istovremeno izvršavaju istu instrukciju, samo na svojem dijelu memorije. Moguće je da neće sve jezgre biti potrebne u danom trenutku, bilo zbog kontrole toka programa, bilo zbog količine podataka na kojima se instrukcija izvršava, te tada neke jezgre mogu u tom trenutku čekati da one jezgre koji su potrebne izvrše instrukciju. Svi moderni procesori također imaju vektorske naredbe koje omogućavaju simultano izvršavanje instrukcija nad vektorima duljine 128, 256 ili 512 bitova (SSE4.2, AVX, AVX2, AVX-512 proširenja za x86 procesorsku arhitekturu, Neon/MPE i Helium/MVE proširenja za ARM arhitekturu, V proširenje za RISC-V arhitekturu). Na primjer, vektor duljine 128 bitova može sadržavati 2 64-bitna (*double*) ili 4

32-bitna (*float*) broja s pomičnim zarezom, odnosno 2 64-bitna , 4 32-bitna, 8 16-bitnih ili 16 8-bitnih cijelih brojeva.

Definicija 1.3. ***MISD računalo*** *(od eng. Multiple-Instruction stream, Single-Data stream) je računalo kod kojeg se više nizova instrukcija izvršava nad jednim nizom podataka.*

Računalo ove klase ima više procesora koji svaki ima svoju upravljačku (kontrolnu) jedinicu te istovremeno izvršavaju različite instrukcije, ali nad istim podatkom (kojeg ne mijenjaju). Ovakva računala nisu u širokoj upotrebi.

Definicija 1.4. ***MIMD računalo*** *(od eng. Multiple-Instruction stream, Multiple-Data stream) je računalo kod kojeg se više nizova instrukcija izvršava nad više nizova podataka.*

Važan primjer računala ove klase su moderna računala s višejezgrenim procesorima. U ovu klasu možemo svrstati i raspodijeljene (distribuirane) sustave računala povezanih mrežom.

Programiranje s dretvama

Višezadaćni rad (eng. *multitasking*) je mogućnost operacijskog sustava da izvodi više od jednog zadatka (programa) u isto vrijeme i time postiže konkurentno izvršavanje. Kod SISD računala, višezadaćni rad se postiže jako brzim naizmjeničnim izvršavanjem različitih programa, čime se postiže privid istovremenog izvršavanja. Kod sustava s više procesora, moguće je postići i pravo istovremeno izvršavanje programa (na hardverskoj, ne samo logičkoj razini). Ovaj poseban slučaj konkurentnosti nazivamo paralelno izvršavanje.

Svaki put kada pokrenemo računalni program, operacijski sustav izradi proces koji odgovara tom programu. Svakom procesu se dodijeli određen dio glavne memorije (adresni prostor) kojem ostali procesi ne mogu pristupiti te mu se omogući izvršavanje na procesoru. Međutim, procesi nisu ograničeni na izvršavanje samo jednog, glavnog slijeda naredbi, već im je dana mogućnost pokretanja dodatnih dretvi (tokova izvršavanja). Budući da su te dretve još uvijek dio istog procesa, one dijele resurse koji su dodijeljeni tom procesu. Podjelom problema na zadatke koji se mogu izvršavati neovisno jedan od drugoga možemo postići konkurentno izvršavanje zadataka, što može olakšati kodiranje određenih mehanizama (npr. upravljanje događajima / *event handling*, mrežno programiranje itd.), a kod višeprocesorskih sustava može dovesti i do znatnog poboljšanja performansi.

Paralelan pristup dijeljenim podacima

Vrlo važan dio paralelnog programiranja je upravljanje pristupom (dijeljenoj) memoriji. Glavna memorija ima značajno vrijeme pristupa (latenciju) koje je puno sporije od radnog takta procesora, što znači da su operacije dohvata ili promjene vrijednosti memorijske

lokacije relativno skupe. Kod modernih procesora taj problem se značajno olakšava hardverski, koristeći L1, L2 i L3 *cache* memoriju. Procesor sprema kopije dijelova memorije u svoju L1 i L2 cache memoriju za brzi dohvat podataka, budući da L1 i L2 cache imaju puno manju latenciju nego glavna memorija. Iako ovaj sustav znatno ubrzava manipulaciju podatcima, on također unosi nove probleme. Naime, ako se vrijednost neke memorijske lokacije promijeni u L2 cache-u jednog procesora (i glavnoj memoriji), ona se mora promijeniti i u L2 cache-u svih ostalih procesora koji imaju taj dio memorije u svojem L2 cache-u. O rješavanju ovakvih problema se brine hardver, no mehanizmi kojim se to ostvaruje mogu usporiti izvršavanje programa pa ih je ponekad potrebno uzeti u obzir kod optimizacije.

Sinkronizacija dretvi

Kako bi se osiguralo da se paralelnim radom dretvi dođe to točnog rezultata, potrebno je zaštiti pristup zajedničkim resursima korištenjem alata za sinkronizaciju dretvi.

Primjer 1.5. *Uzmimo jednostavan primjer s dvije dretve koje izvršavaju slijedeći kod. Neka su početne vrijednosti* $brojac_1 = 100$ *i* $brojac_2 = 0$*. Dok god je* $brojac_1$ *veći od nule, dretve povećavaju* $brojac_2$ *za jedan i smanjuju* $brojac_1$ *za* 1*. Kada bi ovaj kod izvršavala samo jedna dretva, na kraju bi vrijednost* $brojac_1$ *bila nula, a vrijednost* $brojac_2$ *bi bila jednaka početnoj vrijednosti* $brojac_1$ (100)*.*

while $brojac_1 > 0$ ***do***
    $brojac_2 \leftarrow brojac_2 + 1$
    $brojac_1 \leftarrow brojac_1 - 1$
end while

U slučaju dvije dretve, bez ikakvog mehanizma za sinkronizaciju dretvi može se npr. dogoditi da prva dretva pročita vrijednost $brojac_1 = 1 > 0$*, poveća* $brojac_2$ *za jedan, no onda druga dretva pročita vrijednost* $brojac_1 = 1 > 0$*, te također poveća* $brojac_2$ *za jedan i tek onda jedna od dretvi smanji* $brojac_1$ *na vrijednost* 0*, a onda sljedeća na* -1*. Vidimo da su u ovom slučaju konačne vrijednosti* $brojac_2$ *i* $brojac_1$ *različite od onih koje smo željeli, odnosno ovise o redoslijedu kojim operacijski sustav daje procesorsko vrijeme različitim dretvama. Ovakav problem kod paralelnog korištenja zajedničkih resursa nazivamo stanje natjecanja (eng. race condition).*

Postoje dva osnovna mehanizma za sinkronizaciju dretvi, *međusobno isključivanje* (eng. *mutual exclusion*) i semafori. Uz njih još ćemo navesti i barijeru.

Definicija 1.6. ***Kritični odsječak*** *(eng. critical section) je odsječak koda koji pristupa zajedničkim resursima. Takav dio koda je potrebno zaštititi mehanizmom za sinkronizaciju dretvi.*

Kod međusobnog isključivanja, kritične odsječke možemo zaštititi tako da osiguramo da samo jedna dretva može imati pristup kritičnom odsječku u isto vrijeme. Za to koristimo *mutex* (od *"mutual exclusion"*), resurs koji po definiciji može biti u posjedu samo jedne dretve u isto vrijeme, čije će posjedovanje dati dretvi mogućnost ulaska u kritični odsječak. Pri izlasku iz kritičnog odsječka, dretva otpušta vlasništvo nad mutexom tako da ga slijedeća dretva može preuzeti i sama ući u kritični odsječak.

Primjer 1.7. *Možemo jednostavno modificirati kod iz prošlog primjera tako da radi korektno korištenjem međusobnog isključivanja.*

while $brojac_1 > 0$ ***do***
    ZatražiMutex*(mutex1)*
    if $brojac_1 > 0$ ***then***
        $brojac_2 \leftarrow brojac_2 + 1$
        $brojac_1 \leftarrow brojac_1 - 1$
    end if
    OtpustiMutex*(mutex1)*
end while

Ponekad je potrebno da dretva čeka na neki resurs kojeg neka druga dretva prvo treba proizvesti ili dohvatiti. Npr. imamo dvije skupine dretvi, jedne dohvaćaju poruke i stavljaju ih u red, dok druge uzimaju poruke iz reda i obrađuju ih. Tada za sinkronizaciju možemo koristiti semafore. Postoje dvije najvažnije vrste semafora, binarni i opći (brojački/eng. *counting*) semafori.

Definicija 1.8. ***Opći semafor*** *je struktura podataka koja se sastoji od dvije vrijednosti, broja kojeg nazivamo vrijednost semafora i reda blokiranih dretvi nad semaforom.*

Vrijednost semafora uglavnom odgovara količini dostupnih resursa (kod binarnog semafora može biti samo 0 ili 1). Nad semaforom se mogu ostvariti dvije operacije, postavljanje semafora (signal) ili čekanje na semafor (wait). Postavljanjem semafora uvećavamo vrijednost semafora v za 1 i, ako je *red* neprazan, odblokiramo prvu dretvu te je izbacimo iz reda. Čekanjem na semafor ako je $v > 0$ smanjujemo v za 1 i dretva nastavlja s radom, dok se za $v = 0$ dretva blokira i stavlja u *red*.

Primjer 1.9. *Implementacija sustava za obradu poruka s više ulaznih dretvi i više radnih dretvi (početna vrijednost semafora* $osem1.v = 0$*).*

function UlaznaDretva
    while not *krajRada* ***do***
        lokalnaPoruka ← DohvatiPoruku*()*
        ZatražiMutex*(mutex1)*
        PripremiPorukuZaObradu*(lokalnaPoruka)*

```
        OtpustiMutex(mutex1)
        PostaviOpćiSemafor(osem1)
    end while
end function

function RadnaDretva
    while not krajRada do
        ČekajOpćiSemafor(osem1)
        ZatražiMutex(mutex1)
        lokalnaPoruka ← PreuzmiPorukuZaObradu()
        OtpustiMutex(mutex1)
        ObradiPoruku(lokalnaPoruka)
    end while
end function
```

Definicija 1.10. ***Barijera*** *je struktura podataka koja se sastoji od dvije vrijednosti, prolaznosti barijere p (true ili false) i broja dretvi koje su došle do barijere c (*0 *do ukupnog broja dretvi N).*

Kod inicijalizacije barijere početne vrijednosti se postavljaju na $p = false$ i $c = 0$ (također se vodi računa i o ukupnom broju dretvi N). Svaki put kada dretva dođe do barijere, c se povećava za 1 i, ako je $c \neq N$, dretva započinje s čekanjem. Kada zadnja dretva dođe do barijere i postavi $c = N$, prolaznost p se postavlja na *true* i sve dretve se otpuštaju te mogu nastaviti s radom.

Primjer 1.11. *Uzmimo primjer sa više dretvi koje generiraju članove dijeljenog polja A (svaka svoj član polja), te onda svaka dretva koristi cjelokupno generirano polje za nekakav daljnji izračun.*

```
function FunkcijaDretve(A)
    tid ← DohvatiThreadId()
    A[tid] ← GenerirajČlanPolja(tid)
    ČekajNaBarijeri(barrier1)
    DaljnjiIzračun(A,tid)
end function
```

Memorijski model

Definicija 1.12. ***Memorijski model*** *sustava s dijeljenom memorijom je skup jamstava koje programer može očekivati od memorijskog sustava kod višedretvenog pristupa dijeljenim memorijskim lokacijama. [1]*

Postoji više različitih memorijskih modela koji se koriste, od kojih svaki ima prednosti i mane.

Definicija 1.13. *Višeprocesorski sustav nazivamo* ***sekvencijalno konzistentnim*** *ako je rezultat svakog izvršavanja programa jednak kao da su instrukcije svih procesora izvršene u nekom sekvencijalnom redoslijedu, i instrukcije svakog pojedinačnog procesora se u tom redoslijedu pojavljuju u jednakom redoslijedu kao u onom navedenom u programu. [8]*

Ovakav memorijski model je vrlo jednostavan, no s njim postoje i problemi. Glavni problem je da model pretpostavlja da su operacije nad memorijom (čitanje i pisanje) instantne i događaju se u istom redoslijedu u kojem se pojavljuju u programu. Takav pristup ograničava mogućnosti za optimizaciju, pogotovo stoga što traži strog uređaj među memorijskim operacijama koje nisu međusobno kauzalno povezane. Kako bi se prevodiocu otvorile značajne mogućnosti za optimizaciju, koriste se relaksirani memorijski modeli koji relaksiraju zahtjeve iz definicije sekvencijalne konzistentnosti. Kasnije ćemo predstaviti relaksirani memorijski model kojeg koristi OpenMP.

1.2 Programski jezik C++

U ovom radu koristit ćemo programski jezik C++ i implementaciju OpenMP standarda za C++. Trenutno najnovija revizija C++ standarda je C++20, dok slijedeća revizija standarda (C++23) još nije dovršena. Standardna biblioteka C++ već sadrži podršku za višedretveno programiranje, uvedeno u standardu C++11 i nadograđeno u standardima C++14, C++17 i C++20 (zaglavlja <*thread*>, <*atomic*>, <*mutex*>, <*condition_variable*>, <*semaphore*>, <*latch*>, <*barrier*>, <*future*>), koje ćemo koristiti u nekim primjerima i napraviti usporedbe između koda napisanog koristeći OpenMP i koda napisanog koristeći standardnu biblioteku. Osim standardne biblioteke, u C++-u se za višedretveno programiranje mogu koristiti i pthreads (POSIX threads) i Windows (Win32, MFC) biblioteke, no njih se nećemo doticati zbog njihove ograničene prenosivosti. Također, slična OpenMP je Intelova biblioteka oneAPI Threading Building Blocks (oneTBB).

Poglavlje 2

Standard OpenMP

2.1 Ciljevi OpenMP standarda

OpenMP standard je nastao u 1990-ima s ciljem pružanja načina programiranja za višeprocesorske sustave s dijeljenom memorijom (SMP sustav, od eng. *Symmetric multiprocessing*) bez obzira na hardversku arhitekturu.[3] Definirao ga je *OpenMP Architecture Review Board* (OpenMP ARB, skupina proizvođača SMP sustava) i objavio verziju za Fortran 1997, a kasnije i za C i C++. Trenutna verzija standarda je 5.2, objavljena 2021. Svi prevodioci u raširenoj upotrebi podržavaju neku verziju OpenMP, najvažnije GCC 13 (OpenMP 4.5 u potpunosti, velik dio OpenMP 5.0 i dio 5.1), Intel openAPI 2022.3 (OpenMP 4.5 u potpunosti, dio 5.x), Clang 16.0.4 (samo C/C++, OpenMP 4.5 u potpunosti, velik dio 5.0 i dio 5.1), MSVC 14.30 (OpenMP 2.5 u potpunosti, dijelovi 4.x) itd.

2.2 Kratki pregled standarda

OpenMP Common Core

Kako je duljina OpenMP specifikacije rasla sa svakom novom verzijom zbog evolucije hardvera i algoritama koji se koriste u paralelnom računanju, specifikacija OpenMP 5.2 ima više od 650 stranica. S obzirom na broj elemenata u cijeloj specifikaciji, Mattson, He i Koniges u [9] uvode OpenMP Common Core u svrhu olakšanja učenja programiranja s OpenMP bibliotekom. OpenMP Common Core se sastoji od 21 odabranih elemenata OpenMP biblioteke koje koristi većina OpenMP programa, a njihovo učenje i razumijevanje bi trebalo dati dobru podlogu za daljnje programiranje s OpenMP za specifične namjene (tada je potrebno samo dodatno naučiti dio OpenMP bitan za specifičnu namjenu).

U ovom kratkom pregledu ćemo uvesti elemente OpenMP Common Core te spomenuti dijelove standarda izvan Common Core gdje bude prikladno.

Pri pisanju OpenMP programa, potrebno je uključiti zaglavlje omp.h.

```
#include <omp.h>
```

Programer piše serijski kod, te dijelove koje je potrebno paralelizirati stavlja u blokove koda kojima prethode direktive (*#pragma omp ...*). Kod kompilacije programa potrebno je uključiti odgovarajuću zastavicu (kod GCC to je -fopenmp), u suprotnom se direktive ignoriraju i kod će se prevesti u sekvencijalan program (barem što se tiče OpenMP paralelizacije). OpenMP standard od svojih implementacija traži da se ponašaju u skladu s trenutnim vrijednostima tzv. *Internal Control Variables* (ICVs), odnosno unutarnjih kontrolnih varijabli. Svaka od varijabli ima svoj doseg (ovisi o ICV) i vrijednost. Vrijednosti ICV su određene implementacijom, no postoje i određene funkcije kojima programer može direktno pristupiti i upravljati nekim od varijabli. Nama će važne biti samo *nthreads-var*, koja upravlja brojem dretvi koje zatražuju paralelni odsječci koji započinju unutar trenutne direktive, *team-size-var* koja predstavlja broj dretvi u timu koji izvršava trenutni paralelni odsječak, te *thread-num-var*, koja predstavlja redni broj dretve unutar tima dretvi koje izvršavaju trenutni paralelni odsječak.

void omp_set_num_threads(**int** *num_threads*) - funkcija kojom postavljamo *nthreads-var* na *num_threads*.

int omp_get_num_threads(**void**) - funkcija kojom dohvaćamo *team-size-var*.

int omp_get_thread_num(**void**) - funkcija kojom dohvaćamo redni broj trenutne dretve unutar tima koji izvršava paralelni odsječak (*thread-num-var*).

Stvaranje paralelnog odsječka

#pragma omp parallel - direktiva koja se koristi za početak paralelnog odsječka. U C++-u se odnosi na dio koda koji se nalazi u sljedećem kodnom bloku. Konstruira tim dretvi od kojih svaka izvršava sadržaj kodnog bloka. Na kraju odsječka se nalazi implicitna barijera koja se ne može ukloniti *nowait* odredbom.

```
#pragma omp parallel
    {
        // kodni blok kojeg izvrsava svaka dretva u timu
    } // implicitna barijera
```

Raspodjela poslova

#pragma omp for - direktiva koja paralelizira sljedeću **for**-petlju tako da podijeli iteracije petlje između dretvi koje se nalaze u timu koji izvršava trenutni paralelni odsječak. Način raspodjele se može promijeniti odredbom *schedule*. Na kraju kodnog bloka se nalazi implicitna barijera (osim ako se doda *nowait* odredba). U slučaju da se petlja koju treba

paralelizirati nalazi između dva odsječka sekvencijalnog koda, može se koristiti skraćeni oblik *#pragma omp parallel for* koji stvara paralelni odsječak koji obuhvaća samo (također stvoreni) *for* konstrukt.

```
#pragma omp parallel
    {
    #pragma omp for
        for (int i = 0; i < N; i++) {

        } // end for
        // implicitna barijera, osim ako je umetnuta nowait
        // odredba
    } // implicitna barijera
#pragma omp parallel for
    for (int j = 0; j < N; j++) {

    } // end parallel for
    // implicitna barijera
```

---

schedule(static, [chunk_size]), schedule(dynamic, [chunk_size]), schedule(auto) - odredbe koje se mogu nadodati na *#pragma omp for* direktivu. *chunk_size* je opcionalni argument cjelobrojnog tipa.

schedule(static, [chunk_size]) - iteracije petlje se podijele na nizove uzastopnih iteracija veličine *chunk_size*. Ako ukupan broj iteracija nije djeljiv sa *chunk_size*, zadnji niz će sadržavati manji broj iteracija (ostatak pri dijeljenju). Nizovi iteracija se dijele dretvama po njihovom rednom broju, a ako je dretvi manje (N) nego nizova iteracija, $(N+1)$-ti niz će opet biti dodijeljen dretvi s rednim brojem 0, $(N+2)$-ti dretvi s rednim brojem 1 itd. Ako *chunk_size* nije naveden, iteracije petlje se podijele na nizove otprilike jednakih veličina na način da je svakoj dretvi dodijeljen najviše jedan niz iteracija.

schedule(dynamic, [chunk_size]) - iteracije petlje se dijele na nizove na isti način kao i kod *schedule*(*static*), ali se nizovi dodijeljuju dretvama tako da svaka dretva izvrši jedan niz iteracija, te kada završi zatraži novi niz iteracija od preostalih dok ne preostane 0 nezauzetih nizova iteracija. Ako *chunk_size* nije naveden, iteracije petlje se podijele na nizove duljine 1.

schedule(auto) - ovisi o implementaciji, odabir načina raspodjele iteracija se ostavlja na izbor prevodiocu.

#pragma omp single - direktiva koja označava kodni blok kojeg će izvršiti samo jedna dretva iz tima koji izvršava trenutni paralelni odsječak. Na kraju kodnog bloka je implicitna barijera koja se može ukloniti *nowait* odredbom.

Primjer 2.1. *Uzmimo kao primjer program u kojem prvo punimo polje duljine 17 rezultatima kratkih izračuna s malim varijacijama u vremenu izvršavanja, a onda na dobivenim elementima polja obavljamo duge izračune čije vrijeme izvršavanja može znatno ovisiti*

o ulaznim podatcima. Za kratke izračune ćemo koristiti schedule(static) sa chunk_size = 3*, dakle dijelimo iteracije petlje na nizove uzastopnih iteracija veličine* 3 *(dretva* 0 *će izvršavati iteracije s* $i = 0, 1, 2$*, dretva* 1 *s* $i = 3, 4, 5$*, ..., dretva* 3 *s* $i = 9, 10, 11$*, te onda ponovno dretva* 0 *s* $i = 12, 13, 14$ *i dretva* 1 *s* $i = 15, 16$*). Za duge izračune ćemo u ovom slučaju koristiti schedule(dynamic) zbog pretpostavke o varijabilnosti njihovog vremena izvršavanja. Naime, dodjela nizova iteracija dretvama dinamički (tijekom izvršavanja programa) unosi značajan dodatni rad te se zbog toga schedule(dynamic) može isplatiti samo kada bi pri statičkoj dodjeli nizova iteracija nekim dretvama bile dodjeljene iteracije sa značajno duljim vremenom izvršavanja od prosječnog.*

```
#define N 17
    ...
    int polje[N];
    omp_set_num_threads(4);
    #pragma omp parallel
        {
        #pragma omp for schedule(static,3)
            for (int i = 0; i < N; i++) {
                polje[i] = kratki_izracun(i);
            } // end for
            // implicitna barijera
        } // implicitna barijera
    #pragma omp parallel for schedule(dynamic)
        for (int j = 0; j < N; j++) {
            polje[j] = dugi_izracun(polje[j]);
        } // end parallel for
        // implicitna barijera
```

Upravljanje podatkovnim okruženjem

Svaka varijabla koja se koristi unutar nekog OpenMP konstrukta može biti dijeljena ili privatna. Ukoliko je varijabla dijeljena, svaka dretva će pri pokušaju pristupa toj varijabli pristupati istoj memorijskoj lokaciji. Ukoliko je varijabla privatna, svaka dretva će imati svoju verziju varijable te će se verzije iz različitih dretvi nalaziti na različitim memorijskim lokacijama. Privatne varijable mogu biti *private* (privatna verzija varijable je neinicijalizirana) ili *firstprivate* (privatna verzija varijable je inicijalizirana na trenutnu vrijednost). Hoće li varijabla biti dijeljena ili privatna može biti predodređeno, eksplicitno određeno ili implicitno određeno.

- Varijable nereferencijskog tipa automatskog vijeka trajanja deklarirane u kodnom bloku unutar konstrukta su uvijek privatne.

- Dinamički alocirani objekti su uvijek dijeljeni(no ne i pokazivači na njih).
- Varijabla iteratora u *for* konstruktu je privatna.
- Varijable statičkog vijeka trajanja su dijeljene.

Za sve ostale varijable *data-sharing attribute* se navodi eksplicitno uz pomoć odredbi *private, shared* i *firstprivate*, a ako nije eksplicitno naveden, određuje se implicitno.

private(list) - odredba koja se može nadodati na direktive. *list* je lista varijabli. Svaka dretva će unutar direktive imati privatnu neinicijaliziranu varijablu s istim imenom.

shared(list) - odredba koja se može nadodati na direktive. *list* je lista varijabli. Varijable iz *list* će biti dijeljene i sve dretve će imati pristup tim varijablama.

firstprivate(list) - odredba koja se može nadodati na direktive. *list* je lista varijabli. Svaka dretva će unutar direktive imati privatnu varijablu s istim imenom inicijaliziranu na trenutnu vrijednost dijeljene varijable pri dolasku izvršavanja koda do direktive.

default(data-sharing-attribute) - odredba koja se može nadodati na direktive. *data-sharing-attribute* može biti *firstprivate, none, private, shared* i određuje hoće li varijable koje se pojavljuju u konstruktu implicitno biti *firstprivate, private, shared*. Ako je *data-sharing-attribute* **none**, onda je za svaku varijablu koja se pojavljuje unutar kodnog bloka koji odgovara direktivi, a nema predodređen *data-sharing attribute*, potrebno eksplicitno navesti taj atribut koristeći *private, shared* itd. odredbe.

Ukoliko *default* odredba nije prisutna, implicitni *data-sharing attribute* se određuje po sljedećim pravilima: [2, 99-100]

- Kod *parallel* konstrukta, varijabla će biti dijeljena.
- Kod ostalih konstrukata osim onih koji generiraju eksplicitne zadatke, varijabla će predstavljati istu varijablu kao i u kodnom bloku koji okružuje konstrukt.
- Kod konstrukata koji generiraju eksplicitne zadatke, varijabla za koju je u kodnom bloku koji okružuje konstrukt određeno da bude dijeljena među svim implicitnim zadatcima u trenutnom timu dretvi će biti dijeljena.
- Kod konstrukata koji generiraju eksplicitne zadatke, ako varijabla ne ispunjava nijedan od prethodnih uvjeta, njen *data-sharing attribute* će biti *firstprivate*.

```
...
int x = 0, y = 3, z = 0;
#pragma omp parallel default(private)
    {
        x = 1; // x je privatna varijabla zbog default(private)
               // isto kao i private(x)
    } // end parallel
```

```
std::cout  << x << std::endl; // x je 0
#pragma omp parallel default(none) shared(x)
    {
        // x je dijeljena varijabla unutar ovog bloka,
        // sto bi inace bilo zadano ponasanje
        // y i z se ne smiju pojaviti jer njihov
        // data-sharing-attribute nije eksplicitno naveden
    } // end parallel
#pragma omp parallel private(x) firstprivate(y)
    {
        // x je privatna varijabla i nije inicijalizirana
        // y je privatna varijabla inicijalizirana na 3
        std::cout << y << std::endl;
        // svaka dretva ispisuje vrijednost 3
        y += 1;
    } // end parallel
std::cout << y << std::endl; // y je 3
```

Redukcija je agregacija skupa vrijednosti u jednu vrijednost korištenjem asocijativnog operatora (npr. suma ili produkt niza vrijednosti, traženje maksimuma ili minimuma, brojanje ponavljanja neke vrijednosti u nizu...). Često se koristi za obradu podataka u paralelnom i distribuiranom računanju, te je stoga efikasna implementacija paralelne redukcije vrlo važna. U OpenMP-u operacija redukcije je već ugrađena i možemo je koristiti uz pomoć *reduction* odredbe.

reduction(reduction-identifier: list) - odredba koja se može nadodati na direktive kao *#pragma omp parallel*, *#pragma omp for* itd. *reduction-identifier* je jedan od simbola +,∗, &, |, ˆ, &&, ||, *max*, *min* koji odgovaraju asocijativnim operacijama s istim simbolom u C++-u. *list* je lista varijabli koje su **dijeljene** (i inicijalizirane) u paralelnom odsječku kojem pripada direktiva. Izvršava redukciju nad svakom od varijabli iz liste *list* tako da svakoj dretvi dodijeli privatnu varijablu s istim imenom inicijaliziranu na neutralni element za operaciju danu s *reduction-identifier*. Na kraju kodnog bloka koji odgovara direktivi, vrijednosti privatnih varijabli kod svake dretve se kombiniraju sa vrijednostima njihovih dijeljenih verzija koristeći odgovarajući operator (konačni rezultat će biti spremljen u dijeljene varijable iz *list*).

```
#define N 1000
    ...
    int suma = 0;
    #pragma omp parallel for reduction(+ : suma)
        for(int i = 0; i < N; i++){
            suma = 1;
        } // end parallel for
    std::cout << suma; // suma ce biti jednaka broju dretvi koje su
                       // izvrsile bar jednu iteraciju petlje
                       // uvecanom za 0 (pocetna vrijednost varijable
```

```
                        // suma)
```

Paralelizam korištenjem zadataka

Kod mnogih algoritama za uspješnu (ili jednostavnu) paralelizaciju potrebno je korištenje eksplicitnih zadataka, bilo da je algoritam rekurzivan, ili broj iteracija petlje koju je potrebno paralelizirati nije unaprijed poznat, ili se u svakoj iteraciji petlje izvršava neki račun nad odgovarajućim elementom vezane liste, ili vrijeme izvršavanja iteracija jako varira, ili algoritam ima vrlo kompleksan kontrolni tok itd. U svrhu kreiranja eksplicitnih zadataka u verziji 3.0 OpenMP standarda uveden je *task* konstrukt, te su uz njega uvedene i odredbe i sinkronizacijski konstrukti za rad s eksplicitnim zadatcima (npr. *taskwait* konstrukt).

#pragma omp task - direktiva koja kreira eksplicitni zadatak. Kodni blok će ili odmah izvršiti dretva koja kreira zadatak ili ga kasnije izvršiti neka dretva iz tima koji izvršava trenutni paralelni odsječak. Ukoliko je dodana *if(expression)* odredba, te *expression* ima vrijednost *false*, dretva koja stvara zadatak mora odmah početi s njegovim izvršavanjem (te ga mora izvršiti do kraja da bi nastavila s radom). Dretve koje dođu do barijere moraju čekati i dok se izvrše svi zadatci koje su pokrenule dretve iz tima unutar trenutnog kodnog bloka prije barijere.

Sinkronizacijski konstrukti i odredbe

#pragma omp critical [(name)] - direktiva koja označava kritični odsječak. *name* je opcionalni argument koji označava mutex koji se koristi za taj odsječak. Odsječci s istim argumentom *name* će koristiti isti mutex (*name* služi samo kao identifikator, nije potrebno eksplicitno konstruirati mutex). Ako *name* nije naveden, odsječak će koristiti bezimeni mutex kojeg koriste i svi ostali odsječci bez navedenog *name* argumenta.

```
...
int x = 0;
#pragma omp parallel
    {
    #pragma omp critical primjer1
        {
            x += 1;
        } // end critical
    } //end parallel
std::cout << x << std::endl; // x je jednak broju dretvi
```

#pragma omp barrier - direktiva koja inicijalizira i umeće eksplicitnu barijeru na mjesto pozivanja.

U sljedećem odsječku koda svaka dretva izračunava svoj element polja te se onda pomoću redukcije računa njihov prosjek. Kako bi se redukcija po polju ispravno izvršila potrebno je postaviti barijeru ispred for-petlje da bi svi elementi polja bili spremni prije njihovog ubrajanja u sumu polja.

```
#define N 100
    ...
    int polje[N];
    omp_set_num_threads(N);
    double avg = 0.0;
    #pragma omp parallel
        {
            int tid = omp_get_thread_num();
            polje[tid] = izracun(tid);
            #pragma omp barrier
            #pragma omp for reduction (+ : avg)
                for(int i; i < N; i++){
                    avg += polje[i];
                } // end for
            avg = avg / N;
        } // end parallel
```

#pragma omp master - direktiva koja označava kodni blok kojeg će izvršiti samo glavna dretva u timu (ona koja je pokrenula trenutni paralelni odsječak). Na kraju kodnog bloka nema implicitne barijere.

U sljedećem odsječku koda glavna dretva u dijeljenu varijablu *pg* sprema rezultat nekog izračuna s prvim elementom polja, dok se *single* konstrukt koristi kako bi se napravila neka transformacija na ostatku polja. Na kraju *single* konstrukta je implicitna barijera, tako da će *pg* biti izračunat i transformacija ostatka polja biti gotova prije ulaska u drugi *for* konstrukt.

```
#define N 100
    ...
    int polje[N];
    int pg;
    #pragma omp parallel
        {
        #pragma omp for
            for(int i = 0; i < N; i++){
                polje[i] = prvi_izracun(i);
            } // end for
        #pragma omp master
            {
                pg = nekiIzracun(polje[0]);
            }
        #pragma omp single
            {
```

```
            transformacija(polje, 1, N);
        } // end single
          // implicitna barijera
    #pragma omp for
        for(int i = 0; i < N; i++){
            polje[i] = drugi_izracun(polje[i]);
        } // end for
    } // end parallel
```

---

nowait - odredba koja se može nadodati na *#pragma omp for* i još neke direktive. Uklanja implicitnu barijeru s kraja kodnog bloka.

#pragma omp taskwait - kada dretva dođe do ove direktive, mora čekati dok se svi zadatci koje je ta dretva do tada kreirala izvrše (samo direktna djeca te dretve, ne čeka na djecu djece itd.).

U sljedećem odsječku koda pokazujemo slučaj u kojemu nije potrebna sinkronizacija uz pomoć *taskwait* direktive i slučaj u kojemu je potrebna. U prvom slučaju u svakoj iteraciji for petlje izračunava se odgovarajući element polja te u slučaju da je element veći od 100 stvara se eksplicitni zadatak koji dodatno obrađuje element polja. Budući da se taj element nakon stvaranja zadatka nigdje drugdje ne koristi unutar petlje, te *for* konstrukt završava s implicitnom barijerom, nema potrebe za *taskwait*.

```
#define N 1000
    ...
    int polje[N], polje2[N];
    #pragma omp parallel
        {
        #pragma omp for
            for(int i = 0; i < N; i++){
                polje[i] = izracun(i);
                if(polje[i] > 100){
                    #pragma omp task shared(polje)
                    {
                        polje[i] = dodatna_obrada(polje[i]);
                    } // end task
                }
            } // end for, implicitna barijera
        #pragma omp for
            for(int i = 0; i < N; i++)
                polje[i] = izracun2(polje[i]);
            #pragma omp task shared(polje, polje2)
                { // task 1
                    polje2[i] = dugiIzracun(polje[i]);
                #pragma omp task shared(polje2)
                    { // task 2
                        josNekakavDugiIzracun(i, polje2[i]);
                    } // end task
```

```
            } // end task
            int j = opetDugiIzracun(i, polje[i]);
        #pragma omp taskwait
            // ceka na zavrsetak task 1
            josNekiDugiIzracun(i, j, polje2[i]);
        } // end for
          // implicitna barijera, ceka na zavrsetak svih task 2
    } // end parallel
```

U drugom slučaju u svakoj iteraciji prvo se računa nova vrijednost odgovarajućeg elementa polja. Nakon toga stvara se eksplicitni zadatak koji koristi tu vrijednost odgovarajućeg elementa polja da bi izračunao element drugog polja te onda pokreće još jedan ugniježđeni zadatak koji koristi dobivenu vrijednost za nekakav dugi izračun. Budući da se u pozivu funkcije *josNekiDugiIzracun* koristi vrijednost elementa *polje2*, moramo ispred njega umetnuti *taskwait* direktivu. Task 2 ne mijenja elemente *polje2* pa ne moramo čekati na kraj njegovog završavanja, te se na njegov kraj čeka tek na implicitnoj barijeri na kraju *for* konstrukta.

Memorijski model

OpenMP memorijski model spada pod relaksirane memorijske modele s dijeljenom memorijom. Svaka dretva ima pristup dijeljenoj memoriji te svoj privatan dio memorije kojem ostale dretve nemaju pristup. Također, dopušteno joj je da održava privremeni prikaz dijeljenih varijabli koji ne mora u svakom trenutku biti konzistentan sa vrijednostima tih varijabli u glavnoj memoriji. Kako bi se omogućilo ispravno izvršavanje višedretvenih programa, koristi se *(strong) flush* operacija koja prisiljava dretvu da uskladi svoj prikaz dijeljenih varijabli s onim u glavnoj memoriji. Želimo li osigurati da dretva koja čita vrijednost dijeljene varijable koju je mijenjala druga dretva pročita točnu (promijenjenu) vrijednost, moramo osigurati da se sljedeći koraci dogode u točnom redoslijedu[2, 29-30]:

1. Druga dretva mijenja vrijednost varijable
2. Druga dretva *flush* operacijom usklađuje svoj prikaz varijable s globalnom memorijom
3. Prva dretva *flush* operacijom usklađuje svoj prikaz varijable s globalnom memorijom
4. Prva dretva čita vrijednost varijable

Flush operacija može biti eksplicitna ili implicitna, no u većini slučajeva za korektnu i performantnu sinkronizaciju dovoljno je korištenje implicitnih *flush* operacija, npr. na ulasku i izlasku iz paralelnog odsječka ili eksplicitnog zadatka, ulasku u kritični odsječak ili izlasku iz bilo kojeg implicitnog ili eksplicitnog sinkronizacijskog konstrukta.

Poglavlje 3

Primjeri implementacije algoritama

3.1 Primjena na regularne probleme

Kod regularnih problema zadatci koje je potrebno dodijeliti različitim procesorima su međusobno slični i njihova međuovisnost je predvidljiva[10, 41]. Dodjela zadataka je stoga jednostavna i ne utječe značajno na vrijeme izvršavanja programa.

Metoda konjugiranih gradijenata

Kao primjer regularnog problema implementirali smo metodu konjugiranih gradijenata (KG).

Primjer 3.1. *Neka je zadan linearni sustav*

$$Ax = b \tag{3.1}$$

$A \in M_n(\mathbb{R})$, $b \in M_{n1}(\mathbb{R})$, *A regularna matrica i želimo naći* $A^{-1}b = x \in M_{n1}(\mathbb{R})$ *koji je rješenje sustava.*

Za manji n možemo izračunati A^{-1} i pomnožiti s b, no za vrlo velike n taj pristup može biti vrlo neefikasan. Iz tog razloga za rješavanje ovakvih problema usavršeni su mnogi algoritmi, među kojima je i metoda konjugiranih gradijenata (KG). KG je iterativna metoda primjenjiva na simetrične matrice. Koristi se jer je efikasnija od metoda upotrebljivih na općenitim regularnim matricama. Počinjemo s proizvoljnim x_0 i u svakom koraku dobivamo bolju aproksimaciju konačnog rješenja. Može se pokazati [7, 411] da uz egzaktnu aritmetiku metoda konjugiranih gradijenata konvergira do rješenja x u najviše n iteracija, međutim zbog grešaka zaokruživanja na računalima u općenitom slučaju možemo dobiti samo dovoljno dobru aproksimaciju rješenja, tj. takvu da je norma greške manja od neke vrijednosti $m > 0$. Predstavljamo algoritam konjugiranih gradijenata kako je prikazan u [7,

411] uz male izmjene (povremena korekcija r_{i+1} kako ne bi prerano zaustavili izvršavanje algoritma zbog grešaka zaokruživanja).

function KONJUGIRANIGRADIJENTI(A,b,m)
  $x_0 \leftarrow 0$ ▷ proizvoljan
  $r_0 \leftarrow b - Ax_0$ ▷ rezidual
  $p_0 \leftarrow r_0$
  $i \leftarrow 0$
  while $|r_i| > m$ **do**
    $\alpha_i \leftarrow |r_i|^2/(p_i \cdot Ap_i)$
    $x_{i+1} \leftarrow x_i + \alpha_i p_i$
    $r_{i+1} \leftarrow r_i - \alpha_i A p_i$
    if $|r_{i+1}| \leq m$ **or** $i \equiv 0 \mod 25$ **then**
      $r_{i+1} \leftarrow b - Ax_{i+1}$ ▷ povremena korekcija, uključujući kada bi inače petlja završila s izvršavanjem
    end if
    $\beta_i = |r_{i+1}|^2/|r_i|^2$
    $p_{i+1} = r_{i+1} + \beta_i p_i$
    $i \leftarrow i + 1$
  end while
  return x_i
end function

U metodi KG daleko najviše vremena je potrebno za izvršavanje matričnih operacija (posebno umnoška matrice A s vektorom) i stoga se paralelizacija ovog algoritma svodi na paralelizaciju matričnih operacija. Za testne podatke koristimo matricu dobivenu diskretizacijom Poissonove diferencijalne jednadžbe

$$\nabla^2 u = g \tag{3.2}$$

za $g(x, y) = \sin y + \cos x$ na domeni $\Omega = (0, 1) \times (0, 1)$ s Dirichletovim rubnim uvjetom $u(x, y) = 0$ ako $(x, y) \in \delta\Omega$. Dijelimo kvadrat $[0, 1] \times [0, 1]$ na $(N + 2) \times (N + 2)$ mrežu s razmakom $d = 1/(N + 1)$. Tražimo približno rješenje $U(i, j) \approx u(di, dj)$ na točkama mreže (tj. za $i, j = 0...N + 1$). Za vrijednosti na rubu ($i = 0$ ili $N + 1$ ili $j = 0$ ili $N + 1$) iz Dirichletovih rubnih uvjeta dobivamo $U(i, j) = 0$. Iz metode konačnih razlika za unutarnje točke dobivamo da mora vrijediti

$$4U(i, j) - U(i - 1, j) - U(i + 1, j) - U(i, j - 1) - U(i, j + 1) = b(i, j), \tag{3.3}$$

gdje je $b(i, j) = -d^2 f(di, dj)$. Ovime smo dobili linearnu jednadžbu $PU' = b'$ s $n = N^2$ nepoznanica, gdje su U' i b' vrijednosti $U(i, j)$ i $b(i, j)$ za $i, j = 1...N$ preuređene u stupčaste vektore, tj. $U'(k) = U(i, j)$ i $b'(k) = b(i, j)$ za $k = i + (j - 1)N$.

Paralelizacija matričnih operacija u KG uz pomoć OpenMP biblioteke

Pišemo sekvencijalan kod za matrični račun i paraleliziramo ga dodavanjem OpenMP direktiva. Budući da je Poissonova matrica vrlo rijetka, za spremanje matrice koristimo CRS format (engl. *compressed row storage*). Matrica $A \in M_{mn}(\mathbb{R})$ u CRS formatu se sastoji od tri polja:

- **Polje vrijednosti** V sadržava sve vrijednosti elemenata matrice različitih od nule, poredane prvo po retcima pa po stupcima unutar redaka, tj. $index(A_{ij}, V) < index(A_{kl}, V) \equiv (i < k) \vee ((i = k) \wedge (j < l))$.
- **Polje indeksa stupaca** C je jednake duljine kao i polje vrijednosti. Za A_{ij} koji se u polju vrijednosti nalazi na indeksu k vrijedi $C(k) = j$.
- **Polje indeksa redaka** R je duljine $m + 1$. Za i-ti redak $R(i)$ je indeks prvog elementa tog retka različitog od nule u V, a $R(i + 1)$ je indeks zadnjeg elementa tog retka različitog od nule uvećan za 1. Ako i samo ako je $R(i) = R(i + 1)$, redak nema elemenata različitih od nule.

```
class Matrix
{
public:
    size_t m_non_zero;
    size_t m_rows;
    Vector m_values;
    std::vector<size_t> m_col_index;
    std::vector<size_t> m_row_index;

    Matrix(size_t non_zero, size_t rows) : m_non_zero(non_zero),
       m_rows{rows}, m_values(non_zero), m_col_index(non_zero),
       m_row_index(rows+1) {}

};


void add_vector(Vector & result, Vector const & vec_1,
        Vector const & vec_2, Scalar alpha_1, Scalar alpha_2)
{
    size_t n = vec_1.size();

    #pragma omp parallel for
        for(int i = 0; i < n; i++)
        {
            result[i] = alpha_1 * vec_1[i] + alpha_2 * vec_2[i];
        }
    // end parallel for
```

```
    return;
}

Scalar scalar_product(Vector const & vec_1, Vector const & vec_2)
{
    size_t n = vec_1.size();
    Scalar result{};

    #pragma omp parallel for reduction(+ : result)
        for(int i = 0; i < n; i++)
        {
            result += vec_1[i] * vec_2[i];
        }
    // end parallel for

    return result;
}

void mul_matrix_vector(Vector & result, Matrix const & matrix,
        Vector const & vec)
{
    size_t n = vec.size();
    size_t m = matrix.m_rows;

    #pragma omp parallel for
        for(int i = 0; i < m; i++)
        {
            result[i] = 0;
            for(int j = matrix.m_row_index[i];
                    j < matrix.m_row_index[i+1]; j++)
            {
                result[i] += matrix.m_values[j] * vec[matrix.
                    m_col_index[j]];
            }
        }
    // end parallel for

    return;
}

void conj_gradient(Vector & result, Matrix const & matrix_A,
        Vector const & vec_b, int & iterations, Scalar & err_norm_sq,
        Scalar const max_error_norm, int N)
{
    Vector r_i(N), p_i(N), Ap_i(N), Ax_i(N);
    Scalar alpha_i, beta_i, r_norm_sq_old;
```

```
    // r_0 = b - Ax_0
    mul_matrix_vector(Ax_i, matrix_A, result);
    add_vector(r_i, vec_b, Ax_i, 1.0, -1.0);
    err_norm_sq = scalar_product(r_i, r_i);

    iterations = 1;

    if(std::sqrt(err_norm_sq) > max_error_norm){
        p_i = r_i;
        for(; std::sqrt(err_norm_sq) > max_error_norm; iterations++){
                    // j def= i + 1
            mul_matrix_vector(Ap_i, matrix_A, p_i);
                    // Ap_i = A * p_i
            alpha_i = err_norm_sq / scalar_product(p_i, Ap_i);
                    // alpha_i = <r_i, r_i> / <p_i, p_i>_A
            add_vector(result, result, p_i, 1.0, alpha_i);
                    // x_j = x_i + alpha_i * p_i
            add_vector(r_i, r_i, Ap_i, 1.0, -alpha_i);
                    // r_j = r_i - alpha_i * Ap_i
            r_norm_sq_old = err_norm_sq;
                    // r_norm_sq_old = <r_i, r_i>
            err_norm_sq = scalar_product(r_i, r_i);
                    // err_norm_sq = <r_j, r_j>

            if((iterations % 25 == 0) ||
                (std::sqrt(err_norm_sq) <= max_error_norm)){
                    // svako malo izracunamo r_i direktno kako bi
                    // izbjegli akumulaciju gresaka zaokruzivanja,
                    // takodjer osiguravamo da ne bi prerano zavrsili
                    // s izracunom
                mul_matrix_vector(Ax_i, matrix_A, result);
                add_vector(r_i, vec_b, Ax_i, 1.0, -1.0);
                        // r_j = b - Ax_i
                err_norm_sq = scalar_product(r_i, r_i);
                        // err_norm_sq = <r_j, r_j>
            }

            beta_i = err_norm_sq/r_norm_sq_old;
                    // beta_i = <r_i, r_i> / <r_j, r_j>
            add_vector(p_i, r_i, p_i, 1.0, beta_i);
                    // p_j = r_j + beta_i * p_i
        }

    }
}
```

Kompilacijom implementacije metode KG uz ovu implementaciju matričnih operacija te -O2 zastavicom za optimizaciju dobivena su sljedeća vremena izvršavanja na paralelnom računalu phi03 Matematičkog odsjeka Prirodoslovno-matematičkog fakulteta u Zagrebu. Računalo ima procesor Intel®Xeon Phi™CPU 7210 @ 1.30 GHz sa 64 jezgre i jednom dretvom po jezgri. Procesor ima 2MiB L1d, 2MiB L1i i 32 MiB L2 *cache*-memorije. Računalo također ima 94 GiB radne memorije.

Tablica 3.1: Vrijeme izvršavanja u ms, matrica dimenzija $N^2 \times N^2$

broj dretvi	N = 500	N=1000
1	13230	104923
2	6665	52398
4	3356	26320
8	1707	13306
16	899	6800
32	510	3580
64	415	2194

Tablica 3.2: Ubrzanje u odnosu na izvršavanje s jednom dretvom

broj dretvi	N = 500	N=1000
1	1.00x	1.00x
2	1.98x	2.00x
4	3.94x	3.99x
8	7.75x	7.89x
16	14.72x	15.43x
32	25.94x	29.31x
64	31.88x	47.82x

Vidimo da ubrzanje skalira gotovo linearno s povećanjem broja procesora, odnosno postižemo gotovo optimalno smanjenje vremena izvršavanja.

3.2 Primjena na neregularne probleme

Kod neregularnih problema zadatci koje je potrebno dodijeliti različitim procesorima su međusobno različiti na način koji stvara nepredvidive ovisnosti između zadataka[10, 41]. Dodjela zadataka stoga može značajno utjecati na vrijeme izvršavanja programa.

Quicksort

Kao primjer neregularnog problema implementirali smo algoritam za sortiranje polja *quicksort*. Koristili smo Hoareovu schemu particije s odabirom središnjeg elementa polja kao pivota:[4, 183] [4, 199]

```
function QUICKSORT(A,p,r)
    if p < r then
        pivotIndex ← HOAREPARTITION(A, p, r)
        QUICKSORT(A, p, pivotIndex)
        QUICKSORT(A, pivotIndex + 1, r)
    end if
end function
function HOAREPARTITION(A,p,r)
    pivot ← A[p + (r − p)/2]
    i ← p − 1
    j ← r + 1
    while TRUE do
        do
            i ← i + 1
        while A[i] < pivot
        do
            j ← j − 1
        while A[j] > pivot
        if i < j then
            zamijeni vrijednosti u A[i] i A[j]
        else
            return j
        end if
    end while
end function
```

Quicksort je rekurzivan algoritam tipa ”podijeli pa vladaj” koji prima tri argumenta, polje koje sortiramo A, početni indeks potpolja p te završni indeks potpolja r. Za sortiranje cijelog polja A duljine N pozivamo QUICKSORT$(A, 0, N-1)$ (kod indeksiranja polja koje počinje od nule). Funkcija particije odabire neki element potpolja (u ovom slučaju središnji element) kao pivot (istaknuti element) te dijeli potpolje na dva dijela tako da se svi elementi manji od pivota nalaze ispred njega u potpolju, a svi elementi veći od pivota iza njega u potpolju. Povratna vrijednost funkcije particije je konačan indeks pivota. Quicksort pokreće funkciju particije, prima indeks pivota, te rekurzivno poziva samoga sebe na odgovarajućim potpoljima (na potpolju sa svim elementima manjim od pivota te na potpolju

sa svim elementima većim od pivota).

Paralelizacija quicksorta uz pomoć OpenMP biblioteke

Pišemo sekvencijalan kod za quicksort i paraleliziramo ga dodavanjem OpenMP *task* direktiva. Svaki eksplicitni zadatak izvršava svoju funkciju particije sekvencijalno te onda stvara dva nova eksplicitna zadatka, po jedan za svako potpolje. Za jako malena potpolja paralelizacija nadodaje previše dodatnih troškova, tako da se potpolja veličine manje od 1000 elemenata u potpunosti sortiraju sekvencijalno. Budući da rekurzivno stvaramo veliki broj zadataka, dodatni rad pri stvaranju zadataka te problemi s lokalnosti podataka mogu značajno utjecati na vrijeme izvršavanja.

```
template <typename T>
size_t partition(std::vector<T> & vec, size_t lo, size_t hi){
    T pivot = vec[lo + (hi-lo)/2];
    --lo; ++hi;

    while(true){
        do{
            ++lo;
        } while(vec[lo] < pivot);
        do{
            --hi;
        } while(vec[hi] > pivot);

        if(lo >= hi){
            return hi;
        }

        std::swap(vec[lo], vec[hi]);
    }
}

template <typename T>
void sequential_quicksort(std::vector<T> & vec, size_t lo, size_t hi)
   {
    if(lo >= 0 && hi >= 0 && lo < hi){
        size_t p = partition(vec, lo, hi);
        sequential_quicksort(vec, lo, p);
        sequential_quicksort(vec, p+1, hi);
    }
}

template <typename T>
void quicksort(std::vector<T> & vec, size_t lo, size_t hi, int limit)
   {
```

```
    if(lo >= 0 && hi >= 0 && lo < hi){
        if(hi - lo < limit){
            sequential_quicksort(vec, lo, hi);
        } else{
            size_t p = partition(vec, lo, hi);
            #pragma omp task shared(vec) firstprivate(lo, p, limit)
                quicksort(vec, lo, p, limit);
            #pragma omp task shared(vec) firstprivate(p, hi, limit)
                quicksort(vec, p+1, hi, limit);
        }
    }
}

int main(int argc, char * argv[]){
    ...

    #pragma omp parallel
        {
            #pragma omp single
                quicksort(b, 0, N-1, limit);
                        // b je vektor koji sortiramo
                        // N je duljina vektora
                        // limit je 1000
        }
    // end parallel

    ...
}
```

Za testne podatke koristimo dva skupa podataka. Prvi skup je nasumično promiješana lista riječi dobivena obradom liste riječi engleskog jezika iz Leipzig Corpora kolekcije (News kategorija, 1M, 2023)[6]. Konačna lista riječi se sastoji od 263555 riječi (tip podataka korišten u C++-u *std* :: *string*). Drugi skup podataka je nasumično generirano polje realnih brojeva (*double*, 64-bit) s 5 milijuna elemenata. Kompilacijom ove implementacije quicksorta te -O2 zastavicom za optimizaciju dobivena su sljedeća vremena izvršavanja na paralelnom računalu phi03 u kojima vidimo da ubrzanje značajno prestaje rasti nakon što koristimo više od 8 dretvi, te nakon više od 16 dretvi više nema nikakvog ubrzanja:

Tablica 3.3: Vrijeme izvršavanja u ms, matrica dimenzija $N^2 \times N^2$

broj dretvi	wordlist	double 5M
1	508	2186
2	267	1125
4	150	609
8	67	357
16	66	269
32	-	258
64	-	378

Tablica 3.4: Ubrzanje u odnosu na izvršavanje s jednom dretvom

broj dretvi	wordlist	double 5M
1	1.00x	1.00x
2	1.90x	1.94x
4	3.39x	3.59x
8	7.58x	6.12x
16	7.70x	8.13x
32	-	8.47x
64	-	5.78x

Bibliografija

[1] Victor Alessandrini, *Shared memory application programming: Concepts and strategies in Multicore application programming*, Morgan Kaufmann, 2016.

[2] OpenMP Architecture Review Board, M. Klemm i B. R. de Supinski, *OpenMP Application Programming Interface Specification Version 5.2.*, 2021.

[3] Barbara Chapman, Gabriele Jost i Ruud Van Der Pas, *Using OpenMP: portable shared memory parallel programming*, MIT press, 2007.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest i Clifford Stein, *Introduction to algorithms, Fourth Edition*, MIT press, 2022.

[5] Michael J. Flynn, *Very high-speed computing systems*, Proceedings of the IEEE **54** (1966), br. 12, 1901–1909.

[6] D. Goldhahn, T. Eckart i U. Quasthoff, *Building Large Monolingual Dictionaries at the Leipzig Corpora Collection: From 100 to 200 Languages.*, Proceedings of the 8th International Language Resources and Evaluation (LREC'12) (2012).

[7] Magnus R. Hestenes, Eduard Stiefel et al., *Methods of conjugate gradients for solving linear systems*, Journal of research of the National Bureau of Standards **49** (1952), br. 6, 409–436.

[8] Leslie Lamport, *How to make a multiprocessor computer that correctly executes multiprocess programs*, IEEE Transactions on Computers c-28 **9** (1979), 690–691.

[9] Timothy G. Mattson, Yun Helen He i Alice E. Koniges, *The OpenMP common core: making OpenMP simple again*, MIT Press, 2019.

[10] Michael McCool, James Reinders i Arch Robison, *Structured parallel programming: patterns for efficient computation*, Elsevier, 2012.

Sažetak

U ovom radu dajemo pregled paralelnog programiranja s dijeljenom memorijom, kao i primjene OpenMP biblioteke na takvo programiranje.

U prvom poglavlju obrađujemo programiranje s dretvama, paralelan pristup dijeljenim podatcima, sinkronizaciju dretvi te memorijske modele sustava s dijeljenom memorijom. Također se dotičemo programskog okruženja koje ćemo kasnije koristiti (programski jezik C++).

U drugom poglavlju uvodimo OpenMP standard, dotičemo se njegovih ciljeva te dajemo kratki pregled standarda. Objašnjavamo kako OpenMP odgovara na potrebe paralelnog programiranja s dijeljenom memorijom.

U trećem poglavlju dajemo primjere paralelizacije algoritama koristeći OpenMP. Kao primjer regularnog problema implementiramo metodu konjugiranih gradijenata za numeričko rješavanje linearnih sustava, a kao primjer neregularnog problema implementiramo algoritam za sortiranje quicksort. Testiramo skalabilnost obje implementacije izvršavanjem na paralelnom računalu s različitim brojevima dretvi i mjerenjem njihovih vremena izvršavanja.

Summary

In this paper we provide an overview of shared memory parallel programming and the application of the OpenMP library to such programming.

In the first chapter we go through multithreading, shared memory access, thread synchronization and shared-memory system memory models. We also touch upon the programming environment used later (the C++ programming language).

In the second chapter we go through the basics of the OpenMP standard, touch upon its goals and give a short overview of the included functionality. We cover the use of OpenMP on shared memory parallel programming.

In the third chapter we give examples of algorithm parallelization using OpenMP. As an example of a regular problem we implement the conjugate gradient method for numerical solving of systems of linear equations, and as an example of an irregular problem we implement the quicksort sorting algorithm. We test the scalability of both implementations by running them on a parallel computer with different thread counts and measuring their execution time.

Životopis

Rođen sam 21.10.1998. godine u Rijeci. Pohađao sam Osnovnu školu Kozala u Rijeci od 2005. do 2013. godine. Za vrijeme osnovnoškolskog obrazovanja sudjelovao sam na županijskim, regionalnim i državnim natjecanjima iz matematike, kemije, geografije, biologije i povijesti. Iz matematike sam osvojio 4 prva mjesta na županijskim natjecanjima i 1 prvo mjesto na regionalnom natjecanju. Iz kemije sam 2 puta bio drugi na županijskom natjecanju i osvojio četvrto mjesto na državnom natjecanju. Također sam osvojio 1 prvo mjesto iz geografije i 2 prva mjesta iz biologije na županijskim natjecanjima.

Nakon završenog osnovnoškolskog obrazovanja upisao sam se u Gimnaziju Andrije Mohorovičića Rijeka, prirodoslovno-matematički smjer, kojeg sam pohađao od 2013. do 2017. godine. Za vrijeme srednjoškolskog obrazovanja sudjelovao sam na županijskim i državnim natjecanjima iz matematike (A varijanta), logike, kemije, fizike i informatike (kategorija Primjena algoritama). Iz matematike sam osvojio 4 prva mjesta na županijskim natjecanjima te II. i III. nagradu na državnim natjecanjima. Iz logike sam osvojio drugo mjesto na državnom natjecanju. Iz kemije sam osvojio 2 prva mjesta na županijskim natjecanjima i 1 peto mjesto na državnom natjecanju. Također sam osvojio 2 prva i 1 drugo mjesto iz fizike te 2 prva mjesta iz informatike na županijskim natjecanjima. Na državnoj maturi sam osim obaveznih predmeta položio i fiziku(5) i kemiju(4).

2017. godine sam upisao preddiplomski sveučilišni studij Matematika na Matematičkom odsjeku Prirodoslovno-matematičkog fakulteta u Zagrebu, kojeg sam uspješno završio 2020. godine i time stekao titulu sveučilišnog prvostupnika matematike. Iste godine upisao sam diplomski studij Računarstvo i matematika na istom fakultetu.