Paralelni Algoritmi

Veljko Petrović
Decembar, 2022

Digresija—uopštavanje

- Ovo je namenjeno da važi samo za HPC i naučne proračune, ali, istinski, važi za skoro sve.
- Naročito zanimljivo jeste koliko dobro važi za matematiku.
- Biti spolja i gledajući unutra, ponekad se čini da je matematika forma čarobnjaštva gde ljudi inicirane u njene unutarnje misterije jednostavno znaju šta da urade kroz neobjašnjive gnostičke metode.
- Ovo, naravno, nije tačno: proces je gotovo uvek pokazivanje homomorfizma između nečega što znamo i nečega što proučavamo i portovanje znanja iz prvog u drugi.
- Zanimljiv sajt koji pokušava da ilustruje univerzalne alate za ovako što je **http://www.tricki.org/tricki/map**

Objedinjenje

- Broj problema je manje-više neograničen.
- Rešenja, sa druge strane, često imaju zajedničke osobine.
- Ove zajedničke osobine mogu da se iskoriste da formiraju grupe rešenja.
- Ovo je odlična olakšica, pošto moramo da razumemo samo ograničen broj stvari da bi rešili veliki broj problema.

- Dobra knjiga na ovu temu je Concrete Mathematics

Klase numeričkih metoda

- HPC je gotovo uvek namenjen raznim numeričkim metodama.
- Numeričke metode su, jednostavno rečeno, mehanizmi za rešavanje matematičkih problema kroz mehaničke metode.
- Možda se naleteli na termin kroz ‘numeričku analizu’ što je rešavanje analitičkih problema kroz mehaničke metode, ali štošta-drugo je podložno istom procesu. Čak i stvari za koje ne bi pomislili da imaju tu ranjivost.
- Numeričke metode koje se često primenjuju u HPC sistemima se mogu podeliti u sedam klasa poznatih kao ‘sedam patuljaka’

Sedam patuljaka

- Gusta linearna algebra
- Retka linearna algebra
- Spektralni metodi
- Metodi N tela
- Struktuirane mreže
- Nestruktuirane mreže
- Monte Karlo metode

Proširenje

- Ovo su, kada je podela napravljena, zaista bile glavne stvari zbog kojih se tračilo HPC vreme, no vreme je donelo i par novih oblasti koje su bitne.

Četiri... džina? Gnoma?

- Prolazak kroz grafove.
- Konačne mašine stanja.
- Kombinatorička logika.
- Statističke tehnike mašinskog učenja.

Klase problema i klase rešenja

- Dok klase problema donekle nameću prirodu rešenja, klase rešenja određuju prirodu rešenja
- Klase problema su tipovi izazova sa kojima se susrećete
- Klase rešenja su više kao alatke koje imate da te izazove nadmašite.

Klasa	Primer
podataka	

Primeri generičkih klasa paralelnih algoritama

Klasa	Primer
Fork/join	Paralelni for
Zavadi pa vladaj	FFT, paralelno sortiranje
Halo zamena	Sistemi konačnih elemenata/razlika
Permutacije	Kanonov algoritam
Sramotno paralelna	Monte Karlo
Menadžer/radnik	Aktivno meš rafiniranje
Zadaci protoka	Pretraga po širini

Terminološka beleška

- Zavadi pa vladaj je nezgodan termin, gledano jezički
- Prevod je engleskog termina ‘divide & conquer’ što je pak prevod Latinske izreke **divide et impera** što je pak njihov prevod izreke pripisane Filipu II Makedonskom *διαίρει καὶ βασίλευε.*
- Bukvalan prevod je ‘podeli i vladaj.’ Prevod koji najviše hvata suštinu je ‘zavadi pa vladaj’ budući da se odnosilo na strategiju eksploatacije podela i nesloge za očuvanje vladavine.
- Engleski prevod je netačan, ali je *mnogo* bolji kao opis stvarne tehnike u algoritmima.

Fork/join

Fork/join

- Ovo je najjednostavniji generički algoritam za paralelizaciju.
- Vi ste ga koristili, konzervativno, barem hiljadu puta.
- Lepa stvar jeste u tome što radi za veliki broj problema.
- Fork znači da jedinstvenu nit izvršavanja cepamo na više niti, a join znači da posle faze paralelnog izvršavanja:
  - Re-sinhronizujemo ono što se izvršava.
  - Akumuliramo rezultate.
  - Vraćamo se na jednostruko izvršavanje.

- Fork/join nije samo OpenMP stvar, nešto vrlo slično je moguće na svakoj arhitekturi, a naročito prirodno na bilo čemu sa deljenom memorijom.

Zavadi pa vladaj

- Ideja zavadi-pa-vladaj algoritama jeste da se nekakav problem:
  - Podeli na manje delove
  - Rekurzivno se nastavi deljenje
  - Deljenje na manje delove ide sve dok se ne dostigne ‘atomski’ nivo operacije koji:
    - Ne može dalje da se deli
    - Komputaciono je trivijalan.
- Zavadi-pa-vladaj imaju smisla i u serijskoj implementaciji, ali zaista zablistaju u paralelnom slučaju, naročito u situaciji deljene memorije.

- Sistemi distribuirane memorije stvaraju problem u tome što kašnjenje u komunikaciji može da “pojede” prednosti koje zavadi-pa-vladaj donosi.

Quicksort

- Najklasičniji primer zavadi-pa-vladaj algoritma je, naravno, quicksort algoritam za sortiranje.
- Quicksort je najbrži algoritam za sortiranje u opštem slučaju.
  - Moguće je dokazati da nema bržeg ključ-baziranog algoritma za sortiranje.
  - To i dalje ostavlja sisteme za sortiranje koji nisu bazirani na ključu.
  - Takođe, Quicksort je danas manje primenjen nego nekada zbog pragmatike upotrebe sortiranja u stvarnom kodu i frekvencije patoloških ulaza.

- Uprkos tome, i dalje se koristi (npr. .NET koristi QS podrazumevano, i zanimljiv je kao test paralelizacije)

Quicksort

- Trebalo bi da ovo znate ali...
- Za neki niz odabrati nasumično element koji će biti stožer (pivot), te onda napraviti dva pod-niza, elementi veći od stožera i elementi manji od stožera.
- Ovaj proces izvršiti rekurzivno na rezultujućim pod-nizovima, sve dok se ne dobiju nizovi sa samo jednim elementom.
- Pročitati sortirani niz sa leva na desno

Quicksort

Quicksort

```rust
fn qsort<T: Eq + PartialEq + Clone +
        PartialOrd>(arr: &mut Vec<T>) {
    let len = arr.len();
    quicksort(arr, 0, len - 1, &|a, b| a <
        b);
}
```

Quicksort

```rust
fn quicksort<T: Eq + PartialEq + Clone +
        PartialOrd, F: Fn(&T, &T) -> bool>
        (
    arr: &mut Vec<T>,
    lo: usize,
    hi: usize,
    comparator: &F,
) {
    if lo <= hi {
        let p = part(arr, lo, hi,
        comparator);
        if p == 0 {
            return;
        }
        quicksort(arr, lo, p - 1,
```

Quicksort

```
fn part<T: Clone, F: Fn(&T, &T) -> bool>(
    arr: &mut Vec<T>,
    lo: usize,
    hi: usize,
    comparator: &F,
) -> usize {
    let pivot = match arr.get(hi) {
        Some(v) => v.clone(),
        _ => {
            panic!("Indeks {:?} nije u nizu.
        Kako?", hi)
        }
    };
    let mut i = lo;
```

Quicksort

```
    for j in lo..hi {
        match &arr.get(j) {
            Some(v) => {
                if comparator(v, &pivot) {
                    arr.swap(i, j);
                    i += 1;
                }
            }
            _ => {
                panic!("Indeks {:?} nije u
        nizu u get_pivot petlji.", j)
            }
        }
    }
    arr.swap(i, hi);
```

Quicksort

```rust
fn main() {
    let mut niz = vec![3, 1, 4, 1, 5, 9, 2,
        6];
    qsort::<i32>(&mut niz);
    println!("{:?}", niz);
}
```

Paralelizacija Quicksort algoritma

- Momenat kada počne rekurzija, mi možemo da algoritam paralelizujemo.
- Ako pogledate videćete da do samog kraja, svaka rekurzivna komponenta se izvršava u potpunoj paraleli.
- Nema izazova, pokrenemo rekurzivne komponente u paraleli i gotovi smo.
- Problem je u distribuiranim arhitekturama gde to što je neophodno da se podaci prebacuju sa mesta na mesto dramatično smanjuje efikasnost.
- Rešenje? Quicksort modifikacija bazirana na uzorkovanju.

Quicksort sa uzorkovanjem

- Za niz od N elemenata i P procesa se niz podeli na P jednakih segmenata veličine $\frac{N}{P}$. Svaki proces dobije jedan taj segment.
- Svaki proces lokalno QuickSort-uje svoj segment.
- Rezultujuće sortirane nizove mi uzorkujemo tj. uzimamo vrednosti iz njih na način baziran na globalnoj veličini N i broju procesa P tako što ozimamo uzorke na svakoj Q-toj lokaciji počevši od 0 gde je $Q = \frac{N}{P^2}$
- To znači da su indeksi koje uzorkujemo oblika: $0, \frac{N}{P^2}, \frac{2 \cdot N}{P^2}, \ldots \frac{(P-1) \cdot N}{P^2}$

Quicksort sa uzorkovanjem

- Sada kada imamo odabrane uzorke po svim procesima, oni se skupljaju u korenski proces i sortiraju sa sekvencijalnim QuickSort-om.
- Iz sortiranog skupa uzoraka se bira P-1 stožerskih vrednosti koristeći isti sistem uzorkovanja koji se koristio da se skup uzoraka napravi.
- Sve stožerske vrednosti se pošalju svakom procesu.
- Svaki proces podeli svoj deo globalnog niza u P segmenata koristeći P-1 sožerskih vrednosti.
- Nad rezultujućim podacima se primeni MPI All-to-all operacija čiji je rezultat da svaki od P procesa dobije sve primere jednog od P segmenata.
- Pristigle komponente segmenata se lokalno spoje i serijski QuickSort-uju.

- Sortirani nizovi se spoje u redosledu P-vrednosti. Algoritam je gotov.

Quicksort sa uzorkovanjem

- Primetite da često quick sort-ujemo serijski, lokalno.
- Ovo je šansa da još ubrzamo ovaj algoritam kroz hibridni pristup gde MPI koristimo da implementiramo ceo algoritam, a OpenMP da bi maksimalno ubrzali lokalne QuickSort-ove koji se prirodno paralelizuju na arhitekturama sa deljenom memorijom.

Menadžer-radnik

- Ovo je opšti obrazac paralelnog programiranja gde ima jedna nit odn. proces sa posebnim zaduženjima započinjanja procesa i kontrole, i određeni broj uslužnih niti odn. procesa koji rade sav posao.
- Ovo je prirodno u interaktivnim aplikacijama gde, očigledno, zaista postoji posebna nit: GUI nit.
- Menadžer-radnik rešenja su naročito dobro prilagođena problemima gde ne znamo unapred šta će se u svakom koraku algoritma desiti, no to dinamički evoluira tokom rada.

Deljena magistrala poruka

- Softverski šablon kojim se implementira menadžer-radnik jeste da postoji deljena magistrala podataka (message bus) koji niti/procesi dele na koji, tipično, piše korenska nit/proces, a koji osluškuju uslužne niti/procesi.
- Ako ste radili moderne tehnike distribuiranih serverskih sistema možda ste naleteli na jednu implementaciju ovoga: Kafka.
- U lokalu se nešto slično koristi: svaki GUI sistem je napravljen na sličan način, naročito u Windows-u gde se zaista sve radi preko poruka.

Sramotno paralelni algoritmi

- Zašto sramotno?
- Pa, ideja je, da je toliko lako paralelizovati ove algoritme da vas je, kao HPC inženjera, praktično sramota.
- Renderovanje je klasičan primer.
- Još, možda, impresivniji primer je skoro bilo koji Monte Karlo algoritam.
- Monte Karlo algoritmi su takvi da, efektivno, pogađaju nanovo i nanovo i nanovo i nanovo proizvodeći rezultat koji, istina, nije tačan, ali je svakom iteracijom pogađanja sve tačniji, tj. koriste nasumično uzorkovanje da se asimptotski približavaju tačnom odgovoru.
- Ono što čini većinu Monte Karlo metoda sramotno paralelnim jeste to što iteracija 3039 nema ništa zajedničko sa iteracijom 2929. Mogu se izvršiti u bilo kom redosledu i ne komuniciraju.
- **Fantastično.**

Primer Monte Karlo algoritma—računanje broja π

- Napravimo jedinični kvadrat 1x1.
- U tom kvadratu upišemo jedinični krug.
- Nasumično generišemo vrednosti unutar jediničnog kvadrata.
- Brojimo dve vrednosti: koliko smo tačaka generisali i koliko od tih tačaka je u krugu.
- Odnos između broja tačaka u krugu i broja tačaka ukupno će asimptotski prilaziti π/4.

Kako paralelizovati ovaj algoritam?

- Lako!
- Sve što treba jeste da radimo ovu operaciju paralelno koliko god hoćemo i da na kraju toga skupimo sve brojeve kroz redukciju.
- Gotovo.
- Sramota vas je, zar ne?

Halo komunikacija

- Često imamo veoma paralelan slučaj gde svi procesni elementi rade istu stvar nad različitim podacima skoro bez komunikacije.
- Skoro?
- Pa ako svaki procesni element ima svoj prostorno kompaktan domen gde radi svoju stvar jedini problem su granice tih prostorno kompaktnih domena.
- Budući da particija podataka odgovara particiji prostora, komunikacija je relevantna samo na tim slojevima između.
- Taj sloj se zove ‘Halo’ odn. ‘Oreol’ i ima osobinu dubine, tj. može biti dubok 1, 2, 3, itd. tačaka.

Primeri halo komunikacije

- Traženje ivica nad velikom slikom (dubina od 0 do 7)
- Rešavanje parcijalnih diferencijalnih jednačina za advekciju.
- Množenje retkih matrica.

Advekcija

- Advekcija je opšti slučaj da se nekakvo skalarno polje $f(x,t)$ širi ka smeru uvećane vrednosti skalarne vrednosti x, brzinom v kroz vreme.
- *Šta?*
- Mislite, na primer, provođenje toplote.
- Matematički, ovo je problem da za neku graničnu vrednost, rešimo parcijalnu diferencijalnu jednačinu: $\frac{\partial f}{\partial t} = -v\frac{\partial f}{\partial x}$

Advekcija

- Numerički, možemo da rešimo pređašnju jednačinu kroz metod konačnih razlika.
- Prvo, diskretizujemo i vreme i prostor, vreme u korake simulacije dt, a prostor u uniformni meš.
- Onda se situacija svodi na rešavanje jednačine

Advekcija

$$\frac{f_i^{n+1} - f_i^n}{dt} = -v\frac{f_{i+1}^n - f_i^n}{dx}$$

Konačni oblik jednačine

$$f_i^{n+1} = f_i^n - v\frac{dt}{dx}\frac{f_{i+1}^n - f_i^n}{dx}$$

Diskretizacija i parametri

- U pređašnjem dx i dt su samo veličine koraka diskretizacije, a v je parametar simulacije, tako da možemo da uzmemo da te vrednosti imamo.
- Takođe, ako imamo početne vrednosti (što je neophodno za rešavanje diferencijalnih jednačina bilo koje vrednosti, budući da bez početnih vrednosti diferencijalne jednačine u stvari definišu porodice funkcija, a ne specifične funkcije) jasno je da za momenat n+1 sve što nam treba proističe iz momenta n, tj. poznato je.

Komunikacija između procesa

- U slučaju advekcije komunikacija je trivijalna: samo nam treba pređašnje stanje elementa odmah ‘desno’ od nas i ništa više. To znači da je transmisija podataka relativno jednostavna.
- Halo je ovde dubok samo 1 i nesimetričan je.

Permutacija

- Sistem permutacije jeste specijalizovana forma paralelizma na nivou podataka koji je karakterističan i za halo komunikaciju
- Razlika jeste u tome što jednostavna halo komunikacija nije dovoljna, no je potrebno komunicirati kompleksnije na takav način da su prave informacije na pravom mestu u pravom trenutku.
- Ovo može biti poprilično izazovno zato što zahteva da se, efektivno, radi na konstantno pomerajućem modelu memorije.
- Primer je Kanonov algoritam za distribuirano množenje gustih matrica.

Kanonov algoritam

- Treba da pomnožimo dve matrice i smestimo rezultat u treću.
- Prvo, podelimo matrice na ravnomerne blokove koje odgovaraju jedne drugima, tj. i C i A i B imaju isti broj blokova.
- Onda te blokove podelimo između niti/procesa. ## Kanonov algoritam

Kanonov algoritam

Problem

- Ovde je teškoća u tome što u procesu odgovornom za C11, recimo, imamo samo A11 i B11. Ništa drugo. Možemo da pomnožimo pod-matrice A11 i B11, nema problema, ali šta onda?
- Naravno, mogli bi da držimo celu matricu svuda, ali onda sve što imamo jeste model deljene memorije što je lepo ali...

Rešenje

- Rešenje je komunikacija korak-po-korak.
- Ako opet pogledamo na onu jednačinu...

Rešenje

Proračun	Korak
$A_{10}B_{01}$	3
$A_{11}B_{11}$	0
$A_{12}B_{21}$	1
$A_{13}B_{31}$	2

Rešenje

- Podelili smo ono što treba da se uradi na korake (imamo i još jedan korak na kraju toga: redukciju, ali to nije problem) i valja primetiti da možemo da tačno vidimo na onoj slici podele kad nam treba koji komad podataka.

Sekvenca zavisnosti od podataka

Generalizacija

- Prirodno, ovo isto bi mogli da napravimo za bilo koju ćeliju C
- Obrazac je apsolutno isti.
- Ostaje da vidimo šta tačno razlikuje šta nam treba u koraku N+1 u odnosu na korak N
- Odgovor: Treba nam šta je u A u ćeliji odmah desno, a u B u ćeliji odmah dole.
- Uz wraparound kao da je matrica projektovana na površinu torusa, naravno.
- I to nam treba za svaki korak u svakom procesu.
- Drugim rečima, uvek nam treba ista količina podataka, samo koji su to podaci se menja.
- Način na koji se to menja je predvidiv.

Kanonov algoritam

- Kanonov algoritam je da, dakle, podelimo matricu kao što smo pričali, a onda za svaki korak (koji zavisi od broja blokova na koje delimo matrice) izvršimo mroženje, a onda šiftujemo matricu A ulevo, a matricu B nagore.
- Onda samo ponovimo stvar.
- Drugim rečima, permutacijom ukupnog skupa vrednosti mi činimo algoritam vrlo jednostavnim budući da uvek radi isto, samo sa drugim podacima.
- Dodatan bonus: C matrica se ne pomera, što znači da će na kraju svaki proces imati isti blok kao na početku, samo sa skupljenim vrednostima u sebi koje samo treba gather-ovati.

Kanonov algoritam

Kanonov algoritam

Model toka zadataka

- Task dataflow (model toka zadatka) je, na neki način, maksimalno generički model paralelizacije.
- Postave se podaci i veze između podataka, i onda se rezultujući graf particioniše između procesa uz očekivanu sinhronizovanu komunikaciju tamo gde linija particije između procesa seče ivice grafa.
- Ovaj pristup se često koristi kada je potrebno raditi mašinsku paralelizaciju
  - Breakthrough wanted: Imati softverske alate koji paralelizuju umesto nas bi bilo zgodno.
- Takođe se još više koristi za grafove usled prirodnog homomorfizma između njega i domena problema.