



5—Tehnologije Paralelnog Programiranja

ŠTA JE NA RASPOLAGANJU?



OpenMP

MODEL DELJENE MEMORIJE

Šta je OpenMP?

- ▶ Ovo bi sve trebali da već znate.
- ▶ No, dovoljno je bitno da pređemo ponovo.
- ▶ OpenMP je 'Open MultiProcessing'
- ▶ Centralna ideja je deljena memorija i više niti izvršavanja
- ▶ Nekada davno, svo paralelno programiranje je bilo ovako.
- ▶ Ovo je i dalje stil programiranja koji je najčešći u korisničkim aplikacijama zato što se izvršavaju na jednoj mašini, bez obzira koliko procesora ima.
- ▶ Sam jezik je C/C++ mada OpenMP postoji i za Fortran.

Arhitektura OpenMP-a

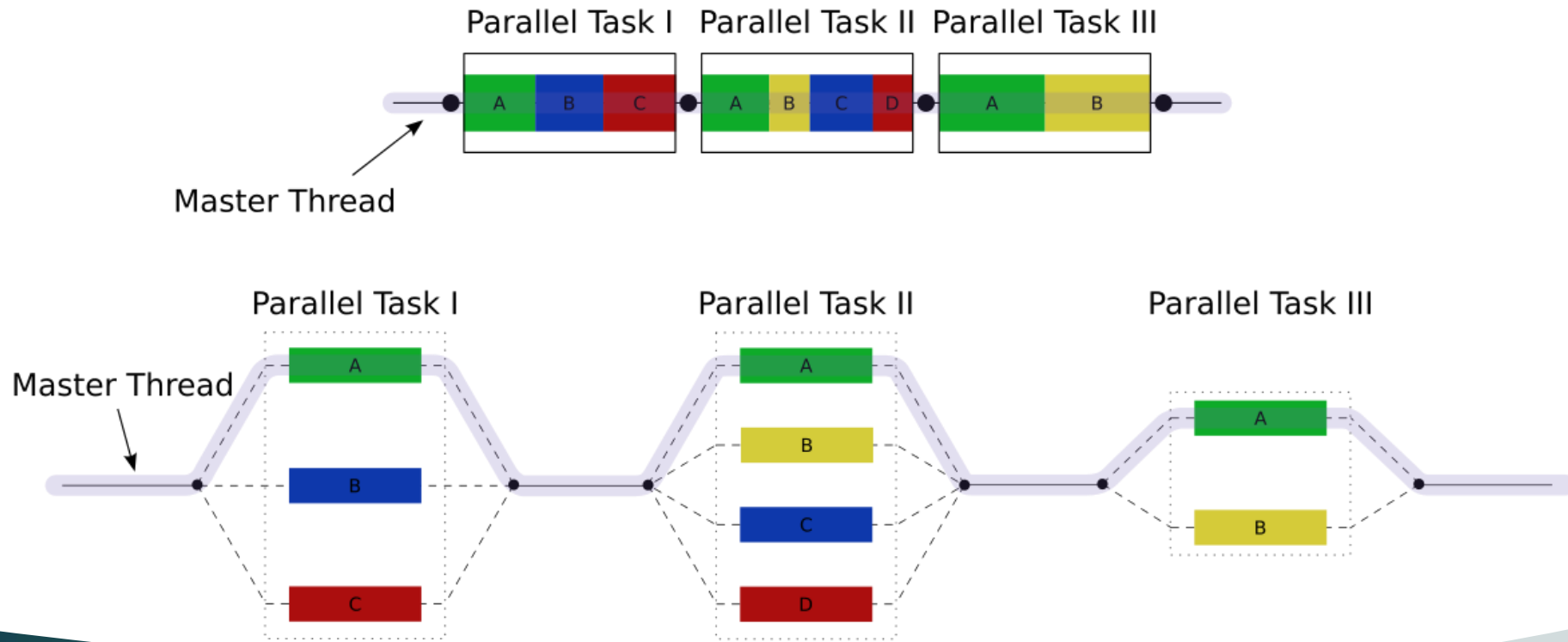
- ▶ Glavna jedinica podele izvršavanja u OpenMP je *nit* (thread)
- ▶ Niti se mogu podeliti u:
 - ▶ Glavnu
 - ▶ Niti radilice (worker threads)
- ▶ Svaka nit je nezavisna traka izvršavanja koja prolazi kroz program
- ▶ Ono što odlikuje glavnu nit jeste što počinje prva i što njeno završavanje završava program. Glavna nit takođe pokreće, zaustavlja i kontroliše niti-radilice.
- ▶ Način na koji se ovakvo ponašanje postiže jeste kroz OpenMP direktive.

Koliko niti?

- ▶ Koliko mi hoćemo.
- ▶ U praksi, broj niti jako zavisi od naše svrhe. Ako imamo više niti da bi u stvari efektno *čekali* na više stvari istovremeno zato što smo u sistemu koji je I/O-ograničen onda je broj niti neograničen, odnosno, samo ograničen sposobnošću sistema da to istrpi.
- ▶ Ovako rade sistemi, npr. web servera ili sistemi koji rade sa velikim brojem operacija nad fajlovima.
- ▶ Ovako, takođe, rade sistemi koji hoće da pruže glatko interaktivno iskustvo dok nešto rade u pozadini: onda obično postoji jedna GUI nit i više niti-radilica.
- ▶ Neki visoko interaktivni sistemi dodeljuju niti zadacima. Tipičan primer su video igre gde obično imate podelu tipa: nit za iscrtavanje, nit za kontrolu, nit za zvuk i nit za veštačku inteligenciju, u zavisnosti od prirode engine-a koji se koristi.

Koliko niti?

- ▶ Ovde, naš cilj je maksimum *brzine* stoga je pravilo mnogo jednostavnije.
- ▶ Broj OpenMP niti treba da bude manji ili jednak ukupnom broju sistemskih niti na raspolaganju.
- ▶ Sistemske niti su ravne broju 'procesora' u kontekstu SLURM-a, tj. broju stvari koje procesor(i) u sistemu mogu da rade istovremeno.
- ▶ Što ne više? Zato što se paralelizam preko hardverske granice ostvaruje koristeći preemptivno izvršavanje što znači da umesto da dobijamo na ukupnoj ostvarenoj brzini mi je gubimo na context-switching overhead.



Fork-join model izvršavanja

Fork-Join model izvršavanja

- ▶ Program počinje izvršavajući samo jednu stvar, sekvencijalno.
- ▶ U nekim trenucima, stvara se veći broj dodatnih niti izvršavanja (račvanje).
- ▶ Te niti izvršavanja se izvode nezavisno sve dok se ne priključe ponovo glavnoj niti kroz formu implicitne sinhronizacije barijerom.

SPMD

- ▶ Najčešća forma koji fork/join paralelizam ima u okviru OpenMP-a jeste Single Program Multiple Data, praktični rođak SIMD arhitekture koji smo pominjali pre par časova.
- ▶ To znači da svaka od niti izvršava isti kod koga samo razlikuju za nit specifične privatne promenljive (više o ovome kasnije).
- ▶ Ovo je onda način da podelimo posao na iste komade.

Alternative SPMD

- ▶ OpenMP je jednako lako konfigurisati da paralelno izvršava različite komade koda.
- ▶ Ponekad algoritam ovo zahteva, mada to nije nužno dobra ideja.
- ▶ Heterogen paralelni kod se ne uklapa tako glatko u fork/join arhitekturu zbog toga što je vreme izvršavanja nepredvidivo što znači da join može da ima nepredvidivu količinu čekanja da se izvršavanje sinhronizuje.
- ▶ Najbolje je ovo prilagoditi kontekstu algoritma.

Ugnježden paralelizam

- ▶ Moguće je računati nit izvršavanja *unutar* već računane niti izvršavanja.
- ▶ Tj. fork unutar fork-a.
- ▶ Kako se ovo izvrši zavisi od toga kako je OpenMP implementacija koja se koristi implementirana.
- ▶ Neke ignorišu dublje slojeve paralelizacije i tretiraju ih kao sekvencijalan kod, a neki izvršavaju kako je napisano dok god ima neiskorišćenih sistemskih niti.
- ▶ Ovakva forma ugnježdavanja je korisna ili u specijalizovanim algoritmima ili kada imamo računar sa *jako puno* paralelnih niti.
- ▶ Intel planira da napravi procesor sa 28 jezgara i 56 sistemskih niti: možda nekakav hipotetički čvor sa četiri takva procesora.

Promenljive niti

- ▶ OpenMP je baziran na modelu deljene memorije.
- ▶ Podrazumevano je da sve promenljive budu *deljene* tj. da svaka nit može da im pristupi.
- ▶ Određene promenljive, sa druge strane, mogu da budu podeljene tako da svaka nit ima svoj primerak.
- ▶ Ovo bi moglo da se uradi i ručno, tako što imamo nekakvu strukturu podatka gde se promenljive indeksiraju kroz broj niti, ali OpenMP to omogućava automatski.
- ▶ Neke promenljive nisu ni privatne ni globalne, doduše, no mešaju ta dva pristupa. Ovo je od koristi kada se rade operacije redukcije, o čemu više kasnije.

Sistemske promenljive i OpenMP

- ▶ OpenMP gleda vrednosti sistemskih promenljivih ne bi li odredio svoje ponašanje.
- ▶ Sistemske promenljive mogu biti nameštene:
 - ▶ U profilu korisnika.
 - ▶ U skripti koja pokreće aplikaciju.
 - ▶ Na komandnoj liniji.

Sistemske promenljive koje poznaje OpenMP

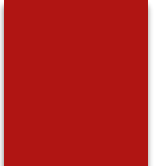
Promenljiva	Tip vrednosti	Značenje
OMP_NUM_THREADS	Broj	Broj istovremenih niti
OMP_DYNAMIC	Bulova	Dinamički menja broj istovremenih niti. Može povećati efikasnost kroz adaptaciju, ali ima cenu u performansama.
OMP_SCHEDULE	Reč.Broj	Tip rasporeda izvršavanja praćen dimenzijom particije izvršavanja. Više o tome kasnije.
OMP_NESTED	Bulova	Da li imamo ugnježdeni paralelizam ili ne. Može da nam pomogne da upravljamo distribucijom niti u komplikovanim situacijama, ali ima fiksnu cenu u overhead-u.

Sistemske promenljive koje poznaje OpenMP

Promenljiva	Tip vrednosti	Značenje
OMP_CANCELLATION	Bulova	Da li 'cancel' direktiva radi ili ne.
OMP_MAX_ACTIVE_LEVELS	Broj	Koliko ugnježenih regiona je dozvoljeno. Podrazumevano je da je neograničeno.
OMP_MAX_TASK_PRIORITY	Broj	Najveći sistemski prioritet koji se dodeljuje zadacima.
OMP_STACKSIZE	Broj praćen sa B, K, M, ili G	Veličina sistemskog steka za jednu nit izvršavanja

Bibliotečke rutine

- ▶ OpenMP se sastoji od: sistemskog okruženja izvršavanja, biblioteka, i direktiva.
- ▶ Biblioteke su klasične C biblioteke i nude funkcionalnost aplikacije kroz funkcije.
- ▶ Fajl zaglavlja za C biblioteku je `omp.h`



```
int n = omp_get_num_threads(); //primetite stil imena
//prefiks je omp, a reči su razdvojene sa _ umesto kroz camelCase
//ova funkcija vraća tekući broj niti koji, prirodno, nikada neće
//biti veći od najvećeg broja niti koji smo podesili.
int k = omp_get_thread_num(); //terminologija može biti
//malo konfuzna ali ono što ovo radi jeste vraća broj tekuće
//niti u opsegu od 0 do n - 1. Glavna nit uvek dobija vrednost
//0, što nam omogućava da je detektujemo.
```

Bibliotečke rutine

Čemu direktive?

- ▶ OpenMP ima nezavidan zadatak
- ▶ C je fundamentalno napravljen sa idejom jednostrukog izvršavanja
- ▶ To je ugrađeno u sam jezik
- ▶ Većina sistema za paralelnog izvršavanje radi tako što nas tera da eksplicitno pravimo strukture podataka koje predstavljaju jedinice paralelnog izvršavanja, te ih pokrećemo sami.
- ▶ Ovo čini takav kod izuzetno nezgodnim za praćenje.
- ▶ Razmišljajte o, recimo, Pthread tehnologiji

Pthreads

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

/* this function is run by the second thread */
void *inc_x(void *x_void_ptr)
{
    /* increment x to 100 */
    int *x_ptr = (int *)x_void_ptr;
    while(++(*x_ptr) < 100);

    printf("x increment finished\n");

    /* the function must return something - NULL will do */
    return NULL;
}
```

Morali smo da paralelni kod enkapsuliramo u svojoj funkciji koja, zbog univerzalnosti i šalje i vraća void* pokazivače.

Ovo je užasavajuće sa tačke gledišta dobre prakse kodiranja. Plus, kod sada nije tamo gde se izvršava nego gurnut u funkciju koja ništa u stvari *ne vraća* no menja okruženje iz koga je pozvana.

Pthreads

```
int main() {
    int x = 0, y = 0;
    /* this variable is our reference to the second thread */
    pthread_t inc_x_thread;
    /* create a second thread which executes inc_x(&x) */
    if(pthread_create(&inc_x_thread, NULL, inc_x, &x)) {
        fprintf(stderr, "Error creating thread\n");
        return 1;
    }
    /* increment y to 100 in the first thread */
    while(++y < 100);
    printf("y increment finished\n");
    /* wait for the second thread to finish */
    if(pthread_join(inc_x_thread, NULL)) {
        fprintf(stderr, "Error joining thread\n");
        return 2;
    }
    /* show the results - x is now 100 thanks to the second thread */
    printf("x: %d, y: %d\n", x, y);
    return 0;
}
```

Primerite da za sve koristimo niti koje identifikuje `pthread_t` tip. Ovo je zato što novu paradigmu programiranja moramo da "proteramo" kroz kod koji za to nije namenjen.

Šta je alternativa?

- ▶ Pa i nema je baš.
- ▶ Problem je u tome što C *nije* napravljen da bude proširiv.
- ▶ Ako konstrukti ugrađeni u jezik ne zadovoljavaju naš obrazac programiranja nema mnogo toga što možemo da uradimo.
- ▶ Možemo da koristimo pre-procesor da modifikujemo kod pre nego stigne kod kompajlera.
- ▶ Ovakav pristup vrlo uspešno koristi, npr. Qt koji dodaje konstrukte u C++ koji prvobitno nisu bili tu.
- ▶ Problem sa tim jeste što kod koji se zaista izvršava (i kod koji posle moramo da debugujemo) nema baš puno veze sa onim što smo napisali. Ovo je *generator* glavobolja.

FOSS rešenje

- ▶ GCC je open source.
- ▶ `git clone git@github.com:gcc-mirror/gcc.git` i već se bavimo razvojem kompajlera.
- ▶ Zašto jednostavno ne proširiti C funkcionalnošću koja nam treba?
- ▶ Zato što to onda nije više standardni C.
- ▶ Suptilne nekompatibilnosti između standarda i implementacije su odgovorne za *neverovatne* komplikacije. Plus, naš kod je sada zauvek vezan za tu, modifikovanu verziju GCC-a.
- ▶ Može li bolje?

#pragma

- ▶ #pragma je jednostavna ideja
- ▶ To je način da se direktno obratimo kompajleru i damo nekakve instrukcije.
- ▶ #pragma direktive su eksplicitno tu da budu nestandardne: svaki kompajler je potpuno slobodan da doda bilo koji broj svojih pragmi koje rade šta god taj kompajler hoće.
- ▶ ...sve dok *ignorišu sve pragme koje ne znaju šta rade*.
- ▶ Ovo je tajni sastojak koji omogućava proširivost. Naš kod je i dalje legalan čak i ako je pun pragmi: kompajler koji ih ne podržava će samo da ih ignoriše i sve će i dalje da radi.

gcc i openMP podrška

- ▶ Da li moramo onda da modifikujemo GCC?
- ▶ Naravno da ne, neko je već bio fin i to uradio umesto nas.
- ▶ Sve što je neophodno jeste da kompajliramo naš kod sa opcijom -fopenmp i dobijemo svu OpenMP podršku koja nam treba, a bibliotečke funkcije pruža libgomp

GCC podrška po kompajlerima

Od GCC verzije	Podržan OpenMP standard	Na jezicima
4.2.0	2.5	C/C++/Fortran
4.4.0	3.0	C/C++/Fortran
4.7.0	3.1	C/C++/Fortran
4.9.0	4.0	C/C++
4.9.1	4.0	C/C++/Fortran
6.1	4.5	C/C++

Najosnovniji OpenMP program

```
#include <stdio.h>
#include <omp.h>
int main() {
    #pragma omp parallel
    {
        printf("Hello World\n");
    }
    return 0;
}
```

Napomena: Pragme nisu, u stvari, magične.

- ▶ Lako je zaboraviti da, koliko god da su zgodne za programiranje, pragme i konstrukti paralelnog programiranja koje donose, sve se to i dalje izvršava na istom procesoru kao i sav naš drugi kod.
- ▶ Pre ili kasnije to postanu pozivi nad funkcijama, komande kontrole toka itd.

Napomena: Pragme nisu, u stvari, magične.

```
#pragma omp parallel  
{  
    body;  
}
```



```
void subfunction (void *data)  
{  
    use data;  
    body;  
}  
  
setup data;  
GOMP_parallel_start (subfunction, &data, num_threads);  
subfunction (&data);  
GOMP_parallel_end ();
```

Napomena: Pragme nisu, u stvari, magične.

- ▶ Ipak, ako *mi* sve dobro programiramo i ako je FSF sve dobro programirao trebalo bi da ne moramo puno da mislimo o tome šta to OpenMP radi iza kulisa.
- ▶ Ne puno, ali *pomalo*.
- ▶ Ne valja da zaboravimo da su naši alati *alati*.

Osnovna sintaksa

- ▶ Svaka pragma za OpenMP počinje sa `#pragma omp`
- ▶ Zatim ide ključna reč pragme koja definiše šta ta pragma radi praćena parametrima u zagradi (ako ih ima)
- ▶ Moguće je pragme slagati u jednoj direktivi radi uštede prostora.

#pragma parallel

- ▶ Maksimalno račva izvršavanje pred ulazak u predstojeći izraz/blok i izvršava ga u onoliko niti koliko je specificirano.
- ▶ Nit se implicitno račva na početku, a sinhronizuje na kraju.

#pragma private

- ▶ Ova pragma definiše niz promenljivih (definisanih u nizu u zagradama) kao privatne za nit koja ih koristi.
- ▶ Navodi se posle 'parallel' direktive i odnosi se na niti tako stvorene.

Paralelna for petlja

- ▶ Najčešća forma paralelizacije jeste podela iteracija for petlje među nitima. Ako je svaki ciklus petlje nezavisan, onda stepen paralelizma zavisi samo od ograničenja našeg hardvera.
- ▶ Prirodno, OpenMP ima metode koje olakšavaju ovako nešto.

Sekvencijalno zbrajanje nizova

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    const int N = 20;
    int nthreads, threadid, i;
    double a[N], b[N], result[N];
    for(i=0; i<N; i++) { a[i] = 1.0*i; b[i] = 2.0*i; }
    for(i=0; i<N; i++) {
        result[i] = a[i] + b[i];
    }
    printf("TEST result[19] = %g\n", result[19]);
    return 0;
}
```

Paralelno zbrajanje nizova

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main() {
    const int N = 20;
    int nthreads, threadid, i;
    double a[N], b[N], result[N];
    for(i=0; i<N; i++) { a[i] = 1.0*i; b[i] = 2.0*i; }
    #pragma omp parallel
    { //fork
        #pragma omp for
        for(i=0; i<N; i++) {
            result[i] = a[i] + b[i];
        }
    } //join
    printf("TEST result[19] = %g\n", result[19]);
    return 0;
}
```

Neke opservacije

- ▶ Primetite da je paralelno izvršavanje i for petlja različita stvar
- ▶ Paralelno izvršavanje je baš to.
- ▶ For petlja je konstrukt koji *deli rad* između niti.
- ▶ Takođe, valja primetiti da je brojačka promenljiva automatski privatna, što je zgodno.
- ▶ Uprkos tome što su paralelizacija i for-deljenje odvojene operacije moguće je (i poželjno je!) da ih *pišemo* zajedno

Paralelno zbrajanje nizova, kraće

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main() {
    const int N = 20;
    int nthreads, threadid, i;
    double a[N], b[N], result[N];
    for(i=0; i<N; i++) { a[i] = 1.0*i; b[i] = 2.0*i; }
    #pragma omp parallel for
    for(i=0; i<N; i++) {
        result[i] = a[i] + b[i];
    }
    printf("TEST result[19] = %g\n", result[19]);
    return 0;
}
```

Koliko ima niti? I koja radi na čemu?

- ▶ Ovde se valja podsetiti OMP_SCHEDULE promenljive.
- ▶ Ona definiše podrazumevano ponašanje for direktive.
- ▶ SCHEDULE mehanizam ima za cilj, jednostavno, da sve indekse for petlje od 0 do N-1 podeli na neki broj regiona koji se ne preklapaju.
- ▶ Moguće je za svaku for petlju kontrolisati kako će to da uradi kroz schedule direktivu koja u zagradama ima prvo ključnu reč a zatim veličinu regiona.

Ključne reči schedule direktive

Ključna reč	Značenje
static	Iteracije se dele u komade veličine koja je specificirana, ako je specificirana, u suprotnom se veličina računa tako što se ukupan broj iteracija podeli brojem niti. Zatim se tako definisani regioni dele među nitima koristeći "round robin" pristup.
dynamic	Kao prethodno, ali osim na početku, niti se dele u first-come first-served pristupu gde niti traže još posla kada završe rad.
guided	Kao prethodno, ali veličina regiona je fleksibilna i proporcionalna je broju nedodeljenih iteracija podeljenih sa brojem dostupnih niti uz minimum ravan podešenoj veličini regiona.
auto	Kompajler/runtime bira šta se izvršava i kada.
runtime	Omogućava da se schedule podesi iz koda koristeći void <code>omp_set_schedule(omp_sched_t kind, int chunk_size);</code>

Monotono i nemonotono izvršavanje

- ▶ specifikaciju schedule direktive može pratiti ključna reč monotonic i ključna reč nonmonotonic.
- ▶ Monotonic znači da svaka nit izvršava dodeljene iteracije u redosledu strogo povećavajuće vrednosti brojača
- ▶ Nonmonotonic znači da svaka nit izvršava dodeljene iteracije u proizvoljnom, nedeterminističkom redosledu.

Alternativa SPMD modelu: sekcije

- ▶ Umesto da niti izvršavaju fundamentalno isti kod (nad različitim podacima) kroz deljenje posla u nitima alternativa su sekcije.
- ▶ Ideja je jednostavna: u okviru paralelnog regiona se napravi više blokova anotiranih sa 'section' pragmom.
- ▶ Među nitima se onda ravnomerno raspoređuju one koje rade različite sekcije istovremeno i paralelni region se join-uje kada su sve sekcije obavljene.
- ▶ Očigledno, redosled izvršavanja apsolutno nije garantovan.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main() {
    #pragma omp parallel
    {
        #pragma omp sections
        {
            //prvi paralelni blok
            #pragma omp section
            {
                //drug paralelni blok
            }
            #pragma omp section
            {
                //treci paralelni blok
            }
        }
    }
    return 0;
}
```

Sekcije

Sinhronizacija

- ▶ OpenMP ima vrlo labavu komunikaciju između niti
- ▶ To je potencijalno *odlično* budući da omogućava da kod koji je zamalo isti kao sekvencijalni ima sve prednosti paralelizacije, i zaista, ako samo koristimo paralelizaciju operacija u petljama nema šta puno da se brinemo.
- ▶ Dosta programa radi ovako. Numpy na primer: cela tajna je multiprocesorsko jezgro za rad sa vektorima pisano u C-u koje se onda koristi u fundamentalno sekvencijalnim algoritmima.
- ▶ Ovo je maksimum koristi za minimum glavobolje.
- ▶ Ali šta ako hoćemo još brzine i fleksibilnije algoritme?

Sinhronizacija

- ▶ Onda moramo kontrolisati "labavu" komunikaciju između niti.
- ▶ Treba da bude onoliko otvorena koliko može, naravno, ali nikako ne preko toga.
- ▶ U suprotnom imamo grozomorne probleme gde sav kod udara po istoj memoriji i ona završi u nedefinisanoj stanju.
- ▶ Nedefinisana stanja valja izbeći.

```
ERROR: Attempted to parse HTML with regular expression; system returned Cthulhu.
```

Uvećavanje promenljive kao primer haosa

Mi napišemo

- ▶ Thread A: i++
- ▶ Thread B: i++

Ali dobijemo

- ▶ A očita i sa vrednošću 7
- ▶ B očita i sa vrednošću 7
- ▶ B poveća 7 na 8 i smesti ga u i
- ▶ A poveća 7 na 8 i smesti ga u i

Ispovest: Zbog toga kako procesori rade, ovo ne može da se desi baš ovako, ali je zgodan primer.

Implicitna sinhronizacija

- ▶ Svaki paralelni blok je implicitno sinhronizovan, budući da sve niti moraju da sačekaju da se sve ostale niti završe pre nego se izvršavanje nastavi.
- ▶ Ovo omogućava da delimo izvršavanje na sekcije koje zavise jedna od druge.
- ▶ Centralni problem paralelizacije jeste, na kraju krajeva, to što u sekvencijalnom izvršavanju relacija sekvence izvršavanja i relacija zavisnosti su ista stvar.
- ▶ Paralelizacija je *razdvajanje* te dve relacije gde je to moguće.
- ▶ Negde, nešto *mora* da se završi da bi se kasniji deo koda izvršio (nije moguće koristiti neki podataka pre nego što je izračunat osim jako blizu površini horizonta Košija rotirajućih crnih rupa sa Kerovom metrikom. Možda.)

Eksplicitna sinhronizacija

- ▶ Moguće je naterati niti da se sinhronizuju kroz direktive i to:
 - ▶ Direktiva kritičnog regiona
 - ▶ Direktiva glavne niti
 - ▶ Direktiva barijere
 - ▶ Direktiva jednostrukog izvršavanja

Direktiva kritičnog regiona

- ▶ Direktiva kritičnog regiona (`#pragma omp critical`) definiše blok koda kao kritičan.
- ▶ Kritičan kod je takav da se u njemu u jednom trenutku može naći samo jedna jedina nit.
- ▶ Ovo nas štiti baš od onog problema sa inkrementacijom promenljive, budući da sve što treba da uradimo jeste da se postaramo da je `l++` u kritičnom regionu i znamo da će *cela* operacija biti završena odjednom.

Direktiva glavne niti

- ▶ Direktiva glavne niti (`#pragma omp master`) definiše blok koda koji je takav da ga može izvršiti samo i isključivo glavna nit.
- ▶ Svaka nit-radilica koja naleti na ovaj blok će ga ignorisati kao da nije tu.
- ▶ To znači da ovakav kod za razliku od većine sinhronizacija ne usporava izvršavanje.

Direktiva barijere

- ▶ Direktiva barijere (`#pragma omp barrier`) služi da natera niti da se sinhronizuju. Gde god da se stavi u kodu definiše graničnu tačku.
- ▶ Kada bilo koja nit stigne to granične tačke pauzira dok sve druge niti nisu, takođe, stigle do granične tačke. Tek onda se izvršavanje nastavlja.

Direktiva jednostrukog izvršavanja

- ▶ Direktiva jednostrukog izvršavanja (`#pragma omp single`) definiše blok koda koji se izvršava u samo jednoj niti.
- ▶ Radi kao direktiva glavne niti osim što:
 - ▶ Važi za prvu nit koja stigne do nje.
 - ▶ Zahteva sinhronizaciju na kraju bloka, tj. druge niti će čekati dok se ne završi izvršavanje bloka pod direktivnom jednostrukog izvršavanja.
- ▶ Ova druga razlika se može isključiti kroz upotrebu 'nowait' dodatka iza single direktive (`#pragma omp single nowait`)

Redukcija

- ▶ Neke operacije su lake za paralelizaciju pošto uzimaju n ulaza a proizvode n ili više izlaza koji su nezavisni.
- ▶ Onda ih je lako iscepati na delove.
- ▶ Recimo da hoćemo da izračunamo sinus svake vrednosti u nekom ogromnom nizu: vrednost elementa 400494 ne zavisi od vrednosti elementa 403222 i nije bitno da li ih računa jedna nit ili više.
- ▶ Šta kada imamo zavisnost?
- ▶ Pa, ponekad ništa. Neki problemi jednostavno *ne mogu* da se paralelizuju ili zahtevaju lukavstvo (kako paralelizovati računanje Fibonačijevih brojeva?)
- ▶ Ali ponekad je zavisnost malo pravilnija i moguće je koristiti za to specijalizovane konstrukte.

#pragma reduction

- ▶ Postoji 'reduction' pragma sa sintaksom
`#pragma reduction(op : var)`
- ▶ Tu je op operator koji može biti: +, *, -, /, &, ^, ili |
- ▶ var je promenljiva za rezultat
- ▶ Ima samo smisla u paralelnom kontekstu
- ▶ Može se odnositi na blok u kome se pojavljuje komad koda koji izgleda ovako:
`var = var op izraz`

Trivijalan primer

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main() {
    const int N=1024;
    int i;
    double a[N], suma=0;
    for(i = 0; i < N; i++) a[i] = 1.0 * i;
    #pragma omp parallel for default(shared) private(i) schedule(static, 4) \
    reduction(+ : suma)
    for(i = 0; i < N; i++) {
        result = result + a[i];
    }
    printf("Suma niza je %f\n", result);
    return 0;
}
```

Fibonačijevi brojevi?

- ▶ Fibonačijevi brojevi su divan primer situacije gde tradicionalne tehnike paralelizacije nisu osobito korisne.
- ▶ Konvencionalni algoritam za njih je *fundamentalno* serijski.
- ▶ Šta znači fundamentalno ovde?
 - ▶ N-ti korak zavisi od $n-1$ i $n-2$ koraka.
 - ▶ $N-1$ korak zavisi od $n-2$ i $n-3$ koraka
 - ▶ $N-2$ korak zavisi od $n-3$ i $n-4$ koraka
 - ▶ itd.
- ▶ Kako onda?
 - ▶ Varanje!

Fina umetnost varanja

- ▶ Reformulišemo problem tako da odgovara našim potrebama.
- ▶ Postoji rekurzivna implementacija računanja fibonačijevog broja, ali takođe postoji *direktna* formula (izvinjavam se za užasavajuću Office tipografiju.)

$$F_n = \sum_{k=0}^{\lfloor \frac{n-1}{2} \rfloor} \binom{n-k-1}{k}$$

Fina umetnost varanja

- ▶ Da li ovo pomaže? Ne skroz. Računanje binomijalnih koeficijenata je bazirano na računanju faktoriijela što opet stvara umeren problem. Može se paralelizovati (to je proizvod niza umesto sume), ali zahteva ugnježdavanje paralelizama što može da bude nepotpuno podržano na našoj arhitekturi.
- ▶ Možemo da budemo direktniji, možda, naročito ako hoćemo da računamo fibonačijeve brojeve zaredom.

Fina umetnost varanja

$$F(n+2) = F(n+1) + F(n)$$

$$F(n+3) = F(n+1) + F(n+2) = F(n+1) \cdot 2 + F(n)$$

$$F(n+4) = F(n+2) + F(n+3) = F(n+1) \cdot 3 + F(n) \cdot 2$$

$$F(n+5) = F(n+3) + F(n+4) = F(n+1) \cdot 5 + F(n) \cdot 3$$

$$F(n+6) = F(n+4) + F(n+5) = F(n+1) \cdot 8 + F(n) \cdot 5$$

Stoga, uopštavajući:

$$F(n+k) = F(n+1) \cdot F(k) + F(n) \cdot F(k-1)$$

Fina umetnost varanja

- ▶ Sada je naš zadatak jednostavan: izračunamo m Fibonačijevih brojeva i čuvamo ih u memoriji
- ▶ Zatim sve brojeve od $m+1$ do $2m$ računamo u m -tostruko paraleli uzimajući $n = m$ a $k = 1..m$.
- ▶ Onda samo pomerimo da je $n = 2m+1$ i ponovimo proces
- ▶ Drugim rečima, naš algoritam je serijski korak prekomputacije, a zatim serijsko ponavljanje m -tostruko paralelizovanog računanja bloka vrednosti
- ▶ Stepen paralelizacije je, onda, praktično neograničen.
- ▶ Pobeda!
- ▶ Više o ovakvim egzibicijama za čas-dva.



OpenMPI

TEHNOLOGIJA PROSLEĐIVANJA PORUKA

Ako imamo OpenMP...

- ▶ ...čemu ovo?
- ▶ OpenMPI je optimizovan za situacije gde je model deljene memorije jednostavno nije primenljiv.
- ▶ MPI je skraćenica od Message Passing Interface i predstavlja metod kojim se rešava programiranje masivno paralelnih arhitektura.
- ▶ U jednom trenutku udarimo u limit na broj procesora koji možemo nagurati u jedno kućište: ako ništa drugo, svi ti procesori se bore oko iste memorije, a transfer stope memorije su prilično oštro ograničene.
- ▶ Postoje situacije gde ta ograničenja ne važe (i to će biti vrlo detaljno istraženo kada budemo pričali o OpenACC-u) ali u opštem slučaju podeljena memorija i vrlo pažljiva komunikacija su naša jedina opcija.

Istorija i poreklo

- ▶ MPI nije biblioteka nego *specifikacija*.
- ▶ Kolekcija standarda koja specificira način programiranja i protokol mrežne komunikacije.
- ▶ Entoni Hoar je principe 'message passing' modela postavio 70-tih godina prošlog veka, a 1992 je tim predstavnika akademije i industrije je postavio MPI specifikaciju vođen stručnjacima kao što je Viljem D. Grop.
- ▶ MPI ima verzije (MPI-1, MPI-2, MPI-3 itd.) koje mogu biti implementirane različitim skupovima alata i biblioteka.
- ▶ MPICH je referentna implementacija.
- ▶ OpenMPI je opšta implementacija.
- ▶ Mnogo specijalizovanih implementacija postoji.

Struktura

- ▶ Kao i OpenMP, OpenMPI nije *jezik*.
- ▶ Odavno je primećeno da posebni jezici retko doživljavaju široku prihvaćenost.
- ▶ OpenMPI je:
 - ▶ Protokol (OSI nivo 5)
 - ▶ Biblioteka koja proširuje postojeće jezike sa konstruktima za paralelizam
 - ▶ Alati

Osnovna implementacija

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv){
    MPI_Init(&argc, &argv);
    printf("Hello World\n");
    MPI_Finalize();
    return 0;
}
```

mpi.h je obavezno zaglavlje

Ova komanda uvek počinje izvršavanje

Ovo mora biti zadnja mpi komanda

Osnovna implementacija

```
veljko@HPC:~/c/mpi1$ mpicc hello.c -o hello
veljko@HPC:~/c/mpi1$ mpirun -np 4 ./hello
Hello World
Hello World
Hello World
Hello World
```

```
9165 pts/0      00:00:00 mpirun
9170 pts/0      00:00:00 hello
9171 pts/0      00:00:00 hello
9172 pts/0      00:00:00 hello
9173 pts/0      00:00:00 hello
```

Komunikacija

- ▶ Kao i većina ovakvih programa, ovo ne radi ništa korisno.
- ▶ Da bi OpenMPI bio koristan, programi koji se paralelno izvršavaju moraju da *komuniciraju*.
 - ▶ Ovo i nije baš 100% tačno. Moguće je držati programe potpuno nezavisno u tkzv. 'share nothing' modelu koji rešava prilično širok dijapazon problema poznat kao 'throughput.'
 - ▶ Rendering je dobar primer.
 - ▶ Ali za ovo bi koristili SLURM i ništa drugo. OpenMPI je čist višak.
- ▶ Ne mogu komunicirati na način na koji to radi OpenMP, preko deljene memorije, zato što iako u našem primeru malopre sve četiri instance su na istom računaru (i istoj ps tabeli) OpenMPI je namenjen da radi na potencijalno udaljenim računarima.

Komunikator

- ▶ Komunikator u OpenMPI-u je kolekcija *procesa* tj. nezavisnih pokrenutih instanci našeg programa i može se porediti sa radio frekvencijom ili kanalom na IRC/Discord serveru.
- ▶ Proces može učestvovati u proizvoljnom broju komunikatora: oni su tu da bi se komunikacija lakše organizovala.
- ▶ Minimalan broj komunikatora je jedan: svaki MPI program apsolutno mora imati barem jedan komunikator koji ne moramo da stvaramo: `MPI_COMM_WORLD`.
- ▶ `MPI_COMM_WORLD` je globalni kanal komunikacije: svi procesi su deo njega.

API komunikatora

- ▶ MPI_Comm je tip identifikatora komunikatora
- ▶ Postoje komande koje služe za manipulaciju komunikatorom koje sve primaju kao parametar identifikator komunikatora

Veličina komunikatora

- ▶ Veličina komunikatora je broj procesa pridruženih komunikatoru.

```
MPI_Init(&argc, &argv);  
int size;  
MPI_Comm_size(MPI_COMM_WORLD, &size);  
printf("%d\n", size);  
MPI_Finalize();  
return 0;
```

MPI komande
tipično vraćaju
vrednost preko
parametara, ne
povratne
vrednosti.

Red procesa

- ▶ Red (rank) procesa je njegov identifikator *unutar komunikatora*.
- ▶ U pitanju je ceo broj nasumično dodeljen u okviru komunikatora u opsegu 0..size-1

```
int main(int argc, char** argv){  
    MPI_Init(&argc, &argv);  
    int rank;  
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);  
    printf("%d\n", rank);  
    MPI_Finalize();  
    return 0;  
}
```

Nedeterminizam izvršavanja

```
veljko@HPC:~/c/mpi1$ mpirun -np 4 ./hello
0
2
3
1
veljko@HPC:~/c/mpi1$ mpirun -np 4 ./hello
2
3
0
1
veljko@HPC:~/c/mpi1$ mpirun -np 4 ./hello
0
2
3
1
veljko@HPC:~/c/mpi1$ mpirun -np 4 ./hello
3
2
0
1
veljko@HPC:~/c/mpi1$ mpirun -np 4 ./hello
```

Slanje poruka na specifičnu adresu

```
MPI_Send(void *message, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

- message—pokazivač na poruku generičkog tipa
- count—koliko elemenata ima u poruci
- datatype—Kog je tipa poruka
- dest—red procesa kome se šalje
- tag—celobrojna vrednost rezervisana za proizvoljnu upotrebu
- comm—komunikator koji se koristi

Primanje poruka

```
MPI_Recv(void *message, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
```

- message—pokazivač na poruku generičkog tipa gde smeštamo vrednost koju dobijemo
- count—koliko elemenata ima u poruci
- datatype—Kog je tipa poruka
- source—red procesa od koga se prima
- tag—celobrojna vrednost rezervisana za proizvoljnu upotrebu
- comm—komunikator koji se koristi
- status—podaci o poruci uključujući ko je stvarno šalje i šta je poslat tag

Osnovni MPI tipovi

MPI tip	C tip
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	unsigned char

Korisnički tipovi podataka

- ▶ Moguće je dodati i naš tip podataka baziran na struct-u
- ▶ Nažalost sintaksa je malo kompleksna
- ▶ Tip se prvo definiše a zatim upiše

Definisanje tipa

```
MPI_Type_create_struct(  
    int number_items,  
    const int *blocklength,  
    const MPI_Aint *array_of_offsets,  
    const MPI_Datatype *array_of_types,  
    MPI_Datatype *new_datatype)
```

Upisivanje tipa

```
MPI_Type_commit(MPI_Datatype *new_datatype)
```

Primer

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char **argv){
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    int message[2];
    int dest, src, tag = 0;
    MPI_Status status;
    if(rank != 0){
        message[0]=rank;
        message[1]=size;
        dest = 0;
        MPI_Send(message, 2, MPI_INT, dest, tag, MPI_COMM_WORLD);
    }else{
        for(src=1;src<size;src++){
            MPI_Recv(message, 2, MPI_INT, src, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
            printf("Poruka sa procesa %d od %d\n", message[0], message[1]);
        }
    }
    MPI_Finalize();
    return 0;
}
```

Primer

```
veljko@HPC:~/c/mpi2$ mpirun -np 4 ./msg
```

```
Poruka sa procesa 1 od 4
```

```
Poruka sa procesa 2 od 4
```

```
Poruka sa procesa 3 od 4
```

Primer

- ▶ Ovo je klasična implementacija radnik-menadžer obrasca u MPI.
- ▶ Nema ničeg *posebnog* oko procesa sa redom 0, ali ga je lako naći.
- ▶ Primetite da je izlaz ovog procesa sada deterministički.

Barijerna sinhronizacija

- ▶ Neke komande nisu namenjene posebnom procesu: neke služe za adresiranje više procesa
- ▶ To su kolektivne komande ('collectives')
- ▶ Primer toga je `MPI_Barrier(MPI_Comm communicator)`
- ▶ Ovo je zaustavljanje izvršavanja dok svi procesi u komunikatoru nisu stigli na isto mesto.
- ▶ Ovo je 1:1 ekvivalentno sa `#pragma omp barrier`

Kolektivna komunikacija

- ▶ Kolektivna komunikacija opisuje situaciju gde imamo više procesa koji komuniciraju sa više procesa.
- ▶ Može se isprogramirati šta god da je to što želim, ali ovaj tip komunikacije pada uglavnom u četiri obrasca:
 - ▶ emitovanje (broadcast)
 - ▶ rasipanje (scatter)
 - ▶ skupljanje (gather)
 - ▶ sinhronizovano skupljanje (allgather)

Broadcast

- ▶ Broadcast služi da podatak sa jednog procesa premesti na sve ostale ciljane procese.
- ▶ Može da koristi da osveži nekakvu tabelu međurezultata, ali je najkorisnije prilikom početka rada procesa.
- ▶ Divan primer broadcast-a se može videti ako razmišljamo o onoj fibonači implementaciji i njenom prvom koraku

Broadcast

```
int MPI_Bcast(void* shared_data, int number, MPI_Datatype  
datatype, int source_process, MPI_Comm communicator)
```

Svi procesi pozivaju istu funkciju. Razlika je u tome što onaj proces čiji je red jednak 'source_process' parametru čita iz shared_data, a ostali samo smeštaju ono što dobiju tu.

Scatter

- ▶ Scatter, kao i broadcast, jeste slučaj slanja sa jednog na više procesa
- ▶ Razlika je u tome što proces koji šalje takođe podeli ono što šalje u onoliko (tipično ne-preklapajućih) podskupa koliko ima procesa kojima se šalje.
- ▶ Ako se razmisli, u onom fibonači primeru posle broadcast-a treba scatter koji podeli skup rednih brojeva fibonačijevog broja koji se računa između procesa

Scatter

```
MPI_Scatter(void *send_data, int send_number, MPI_Datatype  
datatype, void *put_data, int put_number, int source_rank,  
MPI_Comm communicator)
```

send_data/number je bafer za slanje (prazan i nebitan za sve osim za proces koji ima source_rank red)

put_data/number je bafer za primanje koji *svi* koriste.

Gather

- ▶ Gather je obrnut proces od scatter (očigledno).
- ▶ Više procesa šalje jednom procesu delove nekog većeg skupa podataka koji se zatim spaja.
- ▶ Posle broadcast i scatter faze, naš fibonači algoritam bi imao gather da iz procesa izvadi podatke o sledećih m brojeva i skupi ih u niti koja upravlja podacima.

Gather

```
MPI_Gather(void *send_data, int send_number, MPI_Datatype  
datatype, void *put_data, int put_number, int  
destination_rank, MPI_Comm communicator)
```

send_data/number je bafer za slanje (koji svi koriste)

put_data/number je bafer za primanje (koji je prazan i nebitan za sve osim procesa sa redom koji je ravan destination_rank)

Allgather

- ▶ Allgather je isto što i gather samo što ga odmah prati broadcast onoga što se gather-uje.
- ▶ Fibonači program bi, u stvari, imao allgather operaciju umesto gather operacije za sve blokove osim poslednjeg.

Allgather

```
MPI_Allgather(void *send_data, int send_number, MPI_Datatype  
datatype, void *put_data, int put_number, MPI_Comm  
communicator)
```

send_data/number je bafer za slanje (koji svi koriste)

put_data/number je bafer za primanje (koji svi koriste)

Naravno, mogli bi i da radimo gather praćen sa broadcast-om, ali nema razloga.

Redukcije

- ▶ Redukcije su olakšica koja omogućava kompleksnu komunikaciju neophodnu da se paralelizuje, npr. sumiranje niza i slične operacije.
- ▶ Apsolutno je ista namena kao ekvivalentne OpenMP konstrukcije.
- ▶ Veoma je slična sintaksi gather komande:
- ▶ `int MPI_reduce(const void *send_data, void *put_data, int send_number, MPI_Datatype, MPI_OP operation, int destination_rank, MPI_Comm comm)`
- ▶ Jedina razlika jeste operacija koja služi za kombinovanje koja može biti korisnički definisana ili jedna od osnovnih operacija

Redukcioni operatori

Operacija	Operator
Maksimum	MPI_MAX
Minimum	MPI_MIN
Suma	MPI_SUM
Proizvod	MPI_PROD
Logičko I	MPI_LAND
Bit I	MPI_BAND
Logičko ILI	MPI_LOR
Bit ILI	MPI_BOR
Logičko XOR	MPI_LXOR
Bit XOR	MPI_BXOR
Maksimalna vrednost i lokacija	MPI_MAXLOC
Minimalna vrednost i lokacija	MPI_MINLOC

Sinhronizovana redukcija

- ▶ Kao što je Reduce bazirano na Gather, Allgather proizvodi Allreduce
- ▶ `int MPI_Allreduce(const void *send_data, void *put_data, int send_number, MPI_Datatype datatype, MPI_OP operation, MPI_Comm comm)`

Svi/svi komunikaciona šema

- ▶ To je šema komunikacije koja:
 - ▶ Aranžira sve procese tako da formiraju matricu gde su
 - ▶ Redovi procesi
 - ▶ Kolone particije podataka kao što bi ih scatter komanda napravila
 - ▶ Formira transfer podataka iz izlaznog u ulazni bafer tako da efektno *transponira* podatke

Svi/svi komunikaciona šema

Particije podataka				
P r o c e s i	1	2	3	4
	5	6	7	8
	9	10	11	12
	13	14	15	16



Particije podataka				
P r o c e s i	1	5	9	13
	2	6	10	14
	3	7	11	15
	4	8	12	16

Svi/svi komunikaciona šema

```
int MPI_Alltoall(void *send_data, int send_number,  
MPI_Datatype send_datatype, void *put_data, int put_number,  
MPI_Datatype put_datatype, MPI_Comm communicator)
```


Neblokirajuća usmerena komunikacija

- ▶ Do sada, svo slanje podataka je blokirajuće
- ▶ Ako nema Recv za svako Send program *stane*.
- ▶ Takođe, imamo nužno sinhrono ponašanje, to može da uspori program: ako ne moramo da sinhronizujemo, ne treba.
- ▶ Radi isto kao ranije, ali vraća MPI_Request objekat

```
int MPI_Isend(void *message, int count, MPI_Datatype datatype,  
int dest, int tag, MPI_Comm comm, MPI_Request *send_request)
```

```
int MPI_Irecv(void *message, int count, MPI_Datatype datatype,  
int source, int tag, MPI_Comm comm, MPI_Request  
*receive_request)
```

MPI_Request

- ▶ Ovo je promenljiva koja pokazuje na potencijalno neispunjenu operaciju slanja/primanja
- ▶ Kada je imamo, možemo je koristiti da sačekamo da se operacija završi, sinhronizujući naš poziv kada zaželimo:

```
int MPI_Wait(MPI_Request *req, MPI_Status *status)
```

Ovo će vratiti status kada se operacija bude završila.

- ▶ Moguće je i asinhrono proveriti da li se operacija završila kroz:

```
int MPI_Test(MPI_Request *req, int *flag, MPI_Status *status)
```
- ▶ Ovo se odmah završi i postavi flag na true ako je operacija gotova i status na vrednost statusa ako je flag true.



Koncept akceleratora i akceleratorске arhitekture

FINALNI GRADIVNI ELEMENT HETEROGENIH SUPER-RAČUNARSKIH
ARHITEKTURA

Šta je akcelerator?

- ▶ Procesor računara je napravljen da, manje-više, bude beskonačno podesiv i prigodan bilo kom poslu.
- ▶ Iako su određene primene *brže* sve je u principu moguće na CPU arhitekturi.
- ▶ Ovo se plaća: univerzalnost procesora znači da on nije maksimalno prilagođen nijednom poslu.
- ▶ Akcelerator je komponenta koja je deo računara *pored* procesora, a koja omogućava računaru da radi neke stvari brže kroz hardver optimizovan za specifičan scenario korišćenja.

Tipovi akceleratora

- ▶ Akcelerator se najbolje razlikuje po svojoj poziciji u sistemu. Akcelerator može biti:
 - ▶ Na čipu
 - ▶ Koprocesori
 - ▶ Na magistrali
- ▶ Akceleratori na čipu su deo samog procesora i predstavljaju specijalizovanu komponentu u njima.
- ▶ Koprocesori su montirani odmah uz procesor, ali u posebnom pakovanju.
- ▶ Akceleratori na magistrali su odvojene komponente koje su nekom magistralom povezane za nekakav centralni procesor ili procesore.

Tipovi akceleratora

- ▶ Akceleratori koji su na čipu komuniciraju sa procesorom *izuzetno* brzo iz očiglednih razloga.
- ▶ Problem instaliranja akceleratora na čip jeste u tome što:
 - ▶ To povećava toplotno/električne zahteve prema čipu koji se koristi što nije beskonačan resurs.
 - ▶ Povećava kompleksnost čipa što se direktno negativno odražava na cenu.
 - ▶ Nužno deli istu vezu sa memorijom koju normalno koristi procesor što je u skoro svakom sistemu ozbiljno usko grlo.
 - ▶ Akceleratori se i dalje integrišu na čipove, ali samo u posebnim prilikama.

Tipovi akceleratora

- ▶ Akcelerator na magistrali ima određene izuzetno značajne prednosti:
 - ▶ Problem upravljanja temperaturom i strujom je odvojen od glavnog procesora te ga je lakše rešiti.
 - ▶ Akcelerator je u svom, jeftinijem, čipu.
 - ▶ Akcelerator može da ima svoje memorijske resurse.
- ▶ Glavan mana akceleratora u ovoj formi jeste što stvara novo, ključno, usko grlo, a to je magistrala koja ga povezuje sa procesorom koji, nužno, mora funkcionisati kao kontroler i obavljati poslove učitavanja podataka iz glavne memorije i smeštanja rezultata u glavnu memoriju.
- ▶ Ovo ograničenje takođe smanjuje sposobnost upravljanja akceleratorским resursima budući da su oni izolovani u svom malom svetu.
- ▶ Ova ograničenja se rešavaju programski.

Istorija akceleratora—koprocesori

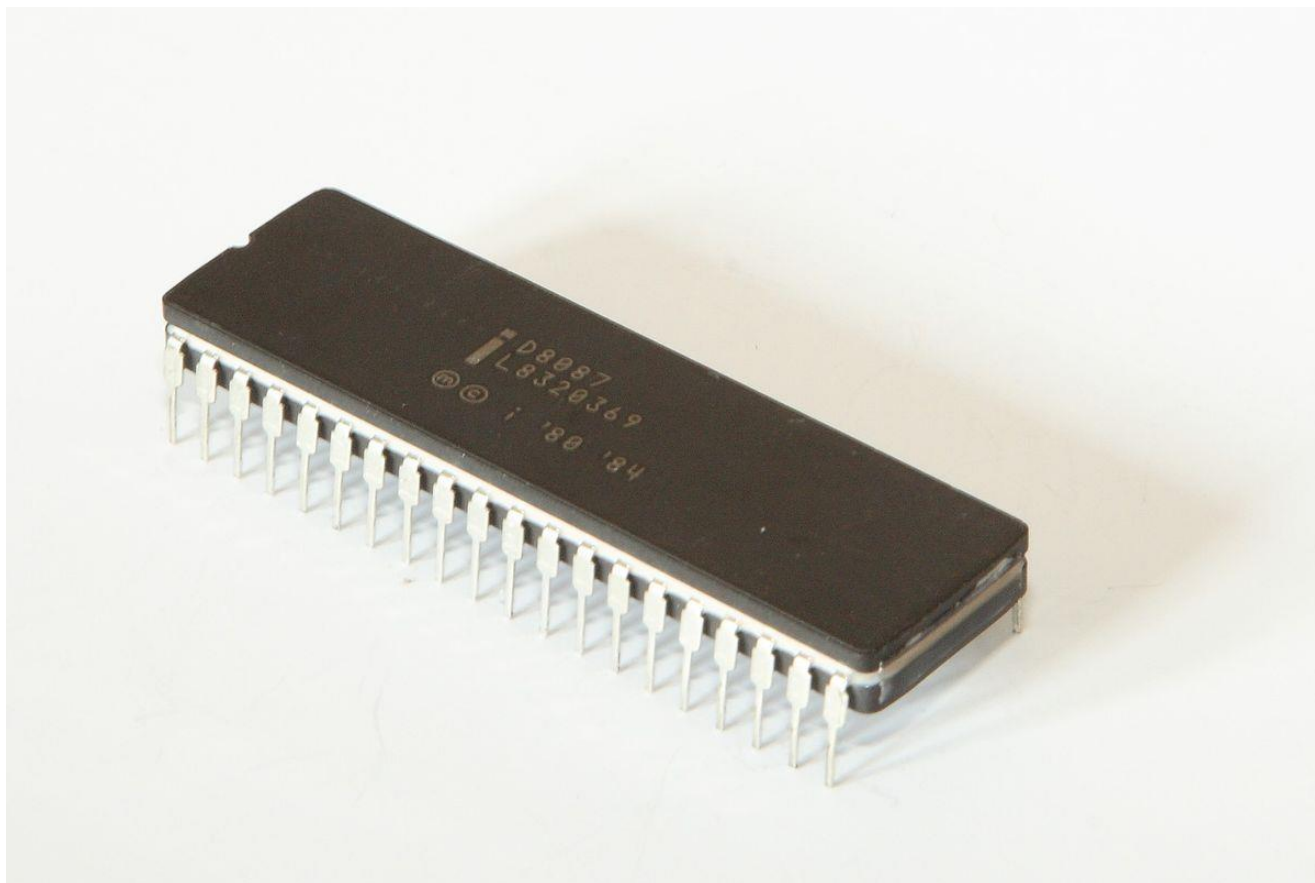
- ▶ Koprocesori su, fundamentalno, dodatni čipovi koji proširuju set instrukcija računara kroz hardverski-implementirane brze verzije operacija iz nekog specifičnog domena.
- ▶ Koprocesori mogu imati svoju memoriju (obično jako malo) ili deliti glavnu ili i jedno i drugo.
- ▶ U ličnim računarima i igračkim konzolama, koprocesori su dugo bili glavni način na koji su se relativno jeftino proizvodili impresivni efekti u performansama.
- ▶ Cena ovakve arhitekture na nivou individualne mašine jeste u ne-uniformnosti programiranja i tome što je fleksibilnost mašine dramatično ograničena.
- ▶ Ovo je fantastično ilustrovano u istoriji računarskih igara u periodu ranih 90-tih.

Koprocesori

- ▶ Koprocesori mogu da imaju izuzetno širok dijapazon primena, recimo:
 - ▶ I/O kroz pametan DMA transfer
 - ▶ video/audio kodiranje/dekodiranje i demultipleksiranje
 - ▶ kriptografija
 - ▶ DSP
 - ▶ grafika

Sudbina koprocesora

- ▶ Koprocesori su i dalje sa nama, ali su uglavnom migrirali, specifično:
 - ▶ Dosta I/O funkcionalnosti je (budući da je lako) završilo u 'omnibus' čipovima na matičnim pločama koji više nego adekvatno obavljaju sve ove poslove.
 - ▶ Određene funkcije su ugrađene direktno u procesor uključujući FPU, MMX, i slično.
 - ▶ Ostatak funkcionalnosti je završio u posebnim akceleratorским dodacima, ponajviše u grafičkim karticama.



Istorijski primer
koprocссора—
Intel 8087

Operacije sa pokretnim zarezmom

- ▶ Operacije sa pokretnim zarezmom nisu lake
- ▶ Sabirati dva cela broja je lako na računaru
- ▶ Svodi se na XOR uz upotrebu bita prenosa, lako je kolo koje to radi napraviti (uistinu, ne moramo, TTL IC 7483 je baš to)
- ▶ Sabiranje dva broja sa pokretnim zarezmom?

Relativna kompleksnost FLOPS-ova

- ▶ Kao rezultat ovoga za lične računare, naročito, operacije sa pokretnim zarezom su bile gotovo nemoguće spore.
- ▶ To je značilo da čitave kategorije proračuna ne mogu da se efektno rade.
- ▶ Ali, budući da su algoritmi za operacije na brojevima pokretnog zareza poznati i definisani IEEE 754 standardom, moguće je implementirati ih u hardveru i ubrzati operacije pokretnim zarezom dramatično: to je Intel i uradio čipom Intel 8087

Intel 8087

- ▶ Trik je bio da se u mašinskom kodu ubaci ESCAPE komanda koja bi predavala kontrolu koprocesoru.
- ▶ Procesor bi (ne znajući da koprocesor uopšte postoji) izvršio očitavanje memorije koju instrukcija pominje (ako je ima), a onda bi te podatke presretnuo 8087 čip.
- ▶ Teoretski govoreći, procesor i koprocesor su mogli da *istovremeno* izvršavaju instrukcije.
- ▶ Teoretski.
- ▶ U praksi bi se pobili oko magistrale i srušili mašinu do tačke da je bio potreban hardverski restart. Zbog toga većina kompajlera iz tog perioda ubacuje WAIT komande odmah posle FPU komandi.
- ▶ Ovo je divna ilustracija problema on-chip i koprocesorski akceleratora.

Evolucija Intel 8087

- ▶ 8087 je unapređivan onako kako su unapređivani i procesori: 80187, 80287, 80387.
- ▶ 80387 koji je uparen sa Intel 80386 procesorom (koji je na vrlo realan način direktan predak modernih procesora, AMD i Intel) je prvi Intel FPU koji je u potpunosti kompatibilan sa IEEE 754 standardom (revizija iz 1985).
- ▶ 80486 serija procesora (i sve od tada do danas) integriše FPU na samom čipu.

Matematički koprocesori kao akceleratori na magistrali

- ▶ U ranim danima ličnih računara, postojala je ogromna glad za dobrim performansama u proračunima sa pokretnim zarezom.
- ▶ Dosta ljudi je htelo računara da *računa* nešto, a većina proračuna koju ljudi žele uključuje decimalne zareze.
- ▶ Ovo je stvorilo značajno tržište za posebne uređaje za ubrzane proračune.
- ▶ Primer toga je Weitek familija *izuzetno* (za to vreme) brzih FP akceleratora (Intel 8087 ostvaruje ~50 kFLOPS-a, Weitek WTL 3167 je mogao da ostvari 5.6 MFLOPS-a samo par godina kasnije).
- ▶ Weitek uređaji nisu bili integrisani u procesor, no su bili dodatni uređaji.

Rani dani akceleratora na magistrali

- ▶ Lako je staviti akcelerator na magistralu: kako on *radi*?
- ▶ Rani akceleratori ove vrste su sami implementirali svoj I/O mapiran na memoriju.
- ▶ To znači da su se Weitek uređaji pretvarali da su dodatan modul memorije i procesor je sa njima komunicirao tako što je izvršavao potpuno obične komande za premeštanje memorijskog sadržaja.
- ▶ PC arhitektura i danas koristi memorijsko mapiranje ove vrste, ali retko tako 'divlje' kao što su rani Weitek uređaji koristili.
- ▶ Kasniji modeli su i emulirali 8087 seriju (kroz presretač na utičnici za koprocetor) i komunicirali na standardniji način kao što je, na primer, EISA VESA local bus tehnologija.

Moderni magistralni akceleratori pre GPU revolucije

- ▶ Pre nego što je GPU tehnologija zavládala kao dominantna u svetu akceleratora, postojale su posebne kartice koje su radile istu stvar *specifično* za HPC svrhe.
- ▶ Recimo, ClearSpeed Advance X620
 - ▶ Dva procesora
 - ▶ PCI-X interfejs sa DMA transferom
 - ▶ 50 GFLOPS
 - ▶ 1GB ECC memorije
 - ▶ 43W TDP

ClearSpeed Advance X620



Arhitektura X620

- ▶ Svaki X620 je imao dva CSX600 procesora
- ▶ Svaki od tih procesora je imao jednu izvršnu jedinicu i određeni broj (zavisi od revizije) procesnih jedinica koje su grupisane u dve sekcije:
 - ▶ Mono (obrađuje skalarne podatke kao blago glup procesor)
 - ▶ Poly (Niz od 96 procesnih elemenata koji obrađuje *nizove* podataka uz tkzv. enable registre kao mehanizam da se određeni elementi koriste ili ne.
- ▶ Poly sekcija je operisala u klasičnom SIMD režimu.

Programiranje sa X620

- ▶ Sam po sebi X620 ne radi ništa.
- ▶ Mora se ručno aktivirati.
- ▶ Da bi se to olakšalo ClearSpeed je napisao ekstenziju programskog jezika C (Cn) koja je dodala 'poly' oznaku za memoriju koja je garantovala da će ta vrednost biti pristupa (i sinhronizovana) između memorije procesora i memorije kartice.
- ▶ ClearSpeed je još morao i da distribuira posebne biblioteke koje su implementirale čestu naučnu funkcionalnost, tj. BLAS, FFT, itd. na način koji je mogao da koristi ClearSpeed proizvode.

Mane akceleratora ove vrste

- ▶ Strahovito skupi
- ▶ Možete programirati samo u jeziku proizvođača!
- ▶ Ograničeni ste na biblioteke proizvođača ili one koje sami napišete da koriste hardver kako treba.
- ▶ Konstantna briga oko lokacije podataka u memoriji
- ▶ ClearSpeed hardver je danas načisto zaboravljen
- ▶ Zašto? GPU revolucija.

GPU revolucija

- ▶ Super-računari su bitni i skupi i sve je to jako lepo ali i najskuplji super-računar na svetu, strahovito impresivni IBM Summit je koštao \$325 000 000.
- ▶ To je dosta novca, ali drugi način da se to kaže jeste 'onoliko novca koliko je zaradio Call of Duty Black Ops 4 za oko dva dana.'
- ▶ Igračka industrija je *ogromna* i kao rezultat:
 - ▶ Može da baci enormne sume novca na istraživanje
 - ▶ Ekonomski efekti skaliranja znače da će cena čuda tehnologije koje proizvede to istraživanje biti dramatično manja.
 - ▶ Budući da jako puno ljudi koristi i hoće da koristi taj hardver softverska podrška će biti mnogo univerzalnija.

Šta je GPU napravljen da radi?

- ▶ Prvi GPU-ovi (tada poznati pod imenom 'video kartice') su bili jako drugačije sprave nego danas i nisu imale nikakvu HPC primenu.
- ▶ Glavne funkcije GPU-ova te vrste su bile da digitalni signal (sadržaj framebuffer-a) pretvori u signal koji se može prikazati na ekranu, u to doba gotovo uvek analogni signal.
- ▶ Dodatne funkcije GPU-ova ove vrste u ne PC arhitekturama su bile funkcije 2D ubrazanja: brz blt transfer, hardverski sprajtovi, hardverska paralaksa...
- ▶ Jako jako korisno ako pravite video igru, ali nije interesantno za nas
- ▶ GPU revolucija ima svoj koren u operacijama neophodnim za 3D grafiku.

Šta 3D grafika hoće?

- ▶ Način na koji se tradicionalna 3D grafika radi jeste da se:
 - ▶ Napravi 3D reprezentacija onoga što treba da se vidi kao serija temena, i ivica među njih definišući nekakvu geometriju.
 - ▶ Ta 3D reprezentacija se anotira sa podacima koji opisuju detalje toga kako ta geometrija reaguje na svetlost:
 - ▶ Normale
 - ▶ Materijali
 - ▶ Mapiranje teksture
 - ▶ Zatim se sistemu dodaju podaci o izvorima svetlosti, pozicije kamere i sličnim globalnim detaljima.
 - ▶ Onda počne proračun koji počne od ovoga a završi sa nizom vrednosti piksela, spremnim za (2D) prikaz.

Proračun 3D grafike

- ▶ Svako teme se procesira tako da je transformisano kako je to odgovarajuće. Ovo se svodi na množenje matrica i vektora.
- ▶ Temena se grupišu u primitive u skladu sa specifikacijom.
- ▶ Geometrija scene se rasterizuje koristeći raycast metod, rezultat ovoga su *fragmenti*. Fragmenti su preteče piksela.
- ▶ Fragmenti scene se osvetljavaju i teksturiraju.
- ▶ Fragmenti se uzorkuju u piksele.
- ▶ 2D pikseli se prikazuju.

Osobine proračuna 3D grafike

- ▶ 3D grafika stalno vrši kompleksne operacije pokretnog zarezavanja nad velikim nizovima podataka.
- ▶ Drugim rečima, grafička kartica je odličan SPMD akcelerator.
- ▶ Ali, kako je opisana ovakva grafička kartica je beskorisna za nas.

Istorija ranih 3D akceleratorskih kartica

- ▶ Sve počinje od legendarnih SGI radnih stanica: IRIX i OpenGL i najranije forme 3D ubrzanja baziranog na Quad primitivu.
- ▶ Poseban hardver za arkadne mašine.
- ▶ Na PC to stiže kroz posebnu seriju Voodoo kartica kompanije 3Dfx. Ovo su *isključivo* akceleratori 3D proračuna i zahtevaju posebnu 2D karticu.
- ▶ Kasnije Voodoo Banshee, S3 ViRGE, ATI Rage i NVidia TNT kartice integrišu 2D i 3D funkcionalnost.
- ▶ Ove kartice su, u početku obavljale samo određene deliće 3D operacija, tipično korekciju perspektive, mapiranje, i filtriranje. Karakterističan 'mutan' izgled ove ere 3D ubrzanja proizvodi niska rezolucija tekstura i linearan režim filtriranja

Istorija ranih 3D akceleratorskih kartica

- ▶ Napredak je nastao sa NVidia GeForce 256 (poznato ime?) i ATI Radeon (poznato ime?) karticama (1999 i 2000) koje su uvele hardverske implementacije svih transformacija i osvetljenja.
- ▶ No, i sa tim dodacima ove kartice su za nas beskorisne.
- ▶ Što?
- ▶ Zato što *samo* rade operacije nad grafikom.
- ▶ Imaju *potencijal* da brzo računaju, ali ne i način da ih nateramo da to rade.
- ▶ Negde oko GeForce 6 generacije se polako uvodi koncept 'programabilnih šejdera'

Shader?

- ▶ Danas ta reč znači nešto drugo (više o tome kasnije) ali u eri o kojoj govorimo šejder (shader) je bila operacija koja je punila delove ekrana nekakvim proračunatim vrednostima boja, tj. *senčila* je deo ekrana.
- ▶ Rani shader-i su bili fiksni ali su mogli da se parametrizuju
- ▶ Kasnije, određena količina programiranja je bila, jedva, moguća.
- ▶ To je bilo vrlo skućeno programiranje, doduše, često bez kontrole toka i ograničenim brojem funkcija i bez sposobnosti da se upravlja resursima kartice.

Razvoj programabilnog shader-a

- ▶ Vođeni željom programera video igara da mogu da implementiraju različite grafičke algoritme sa većom fleksibilnošću, moć shader-a je napredovala, uvodeći sve veću kontrolu.
- ▶ To je eventualno dovelo do pada fiksnog toka izvršavanja (fixed function pipeline) gde su opšti stadijumi renderovanja bili hardverski uslovljeni i njegove zamene sa dinamičnim tokom izvršavanja.
- ▶ Ovo je vrlo brzo posle toga prošireno sa unificiranim shader-ima gde se oni sada tretiraju mnogo više kao programi koji se izvršavaju nego parametrizacije fiksnih koraka.
- ▶ Ovaj razvoj dostiže svoj vrhunac kroz moderne Vulkan/DirectX 12 implementacije

HPC primena

- ▶ U ovom stadijumu, primena GPU-ova za HPC je *mnogo* lakša.
- ▶ GPU možemo da opskrbimo teksturom i modelima koji su u stvari naši podaci i onda 'render' je u stvari naš proračun.
- ▶ Srećom, ni ovo nije potrajalo, i moderni GPU-ovi imaju poseban režim izvršavanja namenjen proračunima.

Arhitektura modernog GPU-a

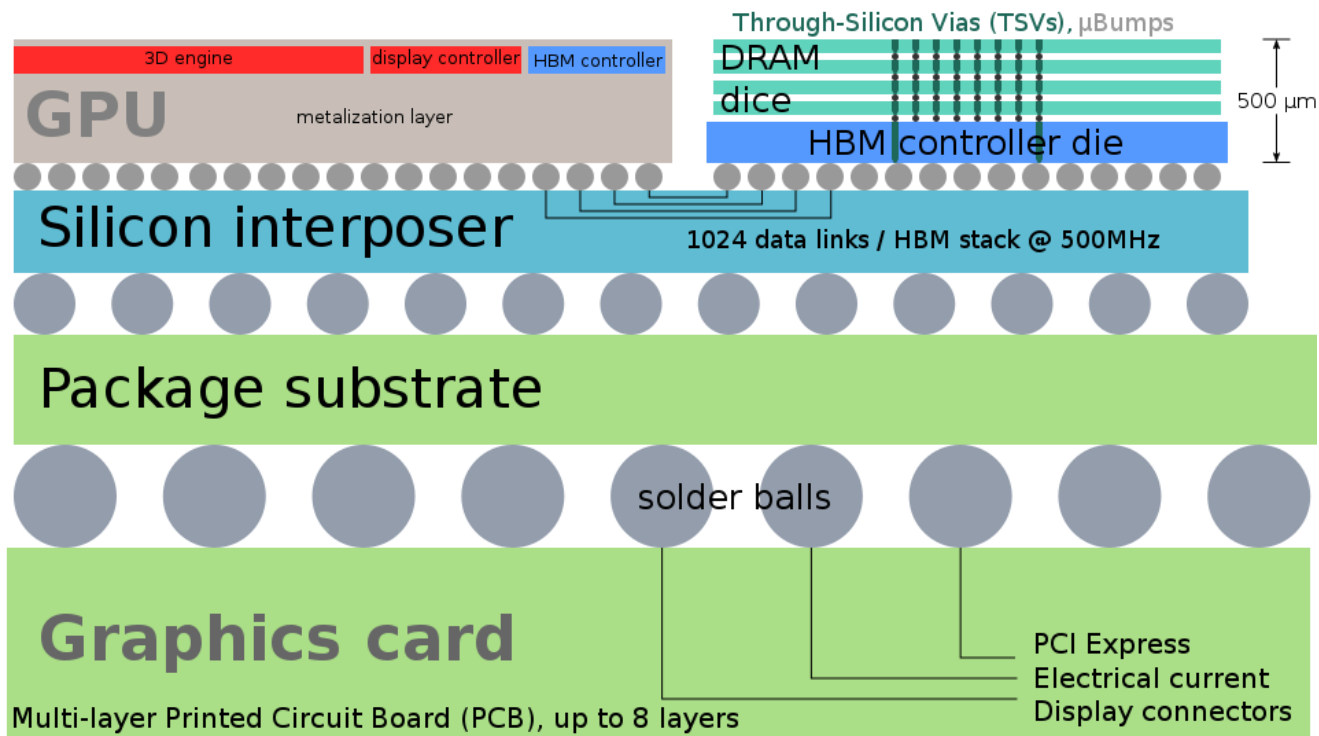
- ▶ Brojke se odnose na Paskal seriju Nvidia kartica, ali same informacije bi trebale da važe u trenutku pisanja.
- ▶ Većina procesa je u samom čipu kartice
- ▶ Centralni čip (GPU) se sastoji od:
 - ▶ Grafičkih klastera (GPC-ova) (Paskal: 6 komada)
 - ▶ L2 keša (Paskal: 4MB)
 - ▶ Kontrolera za memoriju (Paskal: 8 512-bitnih)
 - ▶ PCI Express kontrolera
 - ▶ GigaThread podsistem

GPC

- ▶ Svaki GPC je potpun mini-GPU i radi sve što i GPU.
- ▶ GPC se deli u klastere teksturiranja koji se sastoje od glavne gradivne jedinice GPU-a: multiprocesore toka (streaming multiprocessors, SM)
- ▶ Svaki SM je procesorski element koji se sastoji od CUDA (Compute Unified Device Architecture) jezgara (Paskal: 64 jednostruke preciznosti i 32 dvostruke preciznosti) podeljenih u dva bloka od kojih svaki ima instrukcioni bafer, raspored niti, i 128KB registarske memorije. SM takođe ima 16 jedinica za rad sa memorijom i 16 jedinica za posebne funkcije aproksimacije.

Memorija

- ▶ Ovde postoji oštra razlika između potrošačkih modela i modela napravljenih baš za HPC
- ▶ Potrošački modeli i dalje koriste istu DDR memoriju kao i glavna mašina, samo na visokoj brzini i povezanu kroz *izuzetno* široku magistralu.
- ▶ Uprkos tome, pristup memoriji je značajno usko grlo.
- ▶ HPC modeli koriste mnogo efikasniju HBM2 memoriju
- ▶ HBM2 koristi tehniku gde se GPU i kontroler memorije nalaze u istom fizičkom paketu na deljenom supstratu (silikonskom sistemu presretanja) koji omogućava vrlo brzu komunikaciju
- ▶ Sama memorija je u pločama *na samom kontroleru*, i povezana je ultramalim ultrabrzim vezama *kroz* silicijum samog kontrolera i drugih pločica memorije.



HBM2

Pristup memoriji

- ▶ Moderni GPU-ovi podržavaju izuzetno kompleksan sistem pristupa memoriji koji koristi memorijske stranice da se postara da GPU kod može uniformno da pristupa svojoj memoriji u računaru.
- ▶ Ovo je veliko olakšanje, ali naravno, pristup glavnoj memoriji ima ogromnu cenu.
- ▶ Tipično, problem se rešava tako što se memorija kopira u memorijski prostor kartice, no ovo stvara potencijalne probleme sa sinhronizacijom.
- ▶ Ovo je predmet intenzivnog istraživanja.

Povezivanje

- ▶ Standardno povezivanje je preko PCI Express linka.
- ▶ PCI Express je brz (ali uvek može brže) plus problem nastaje:
 - ▶ Šta kada hoću 4 kartice u jedan računar?
- ▶ Svaka kartica hoće 16 PCI Express linija za transport
- ▶ To je lepo, novi i9-9900K ima *ukupno* 16 PCI Express linija.
- ▶ Neki Xeon procesori imaju više, i Threadripper ima 32, ali, kao što se vidi, ukačiti dve kartice je *teško*
- ▶ Rešenje? NVLink
- ▶ NVLink je izuzetno visoko-performantni mehanizam za komunikaciju na male razdaljine.

Povezivanje

- ▶ NVLink može da povezuje kartice (i tako bi tipično i stavili dve kartice u prosečan računar), a i karticu i procesor ako to procesor podržava (IBM pravi procesore koji ovo mogu i oni su instrumentalni u funkcionisanju Summit mašine).
- ▶ Različite topologije su moguće.
- ▶ PCIExpress ima specifične svoje prednosti ako se koristi uz NVlink kroz RDMA funkcionalnost.
- ▶ RDMA omogućava da sa GPU-om komunicira memorija ili, još bolje, *mrežna kartica* apsolutno bez rada procesora. Ovo dramatično olakšava dizajn masivno paralelnih hibridnih heterogenih arhitektura visokih performansi.

Kako sve ovo programirati?

- ▶ NVidia ima CUDA API
 - ▶ Maksimum performansi
 - ▶ 0% portabilnosti
 - ▶ Prilično ružna sintaksa koja zahteva poseban NVidia kompajler
 - ▶ Odlični alati
- ▶ OpenCL
 - ▶ Poseban jezik
 - ▶ Maksimalno univerzalan
- ▶ OpenACC
 - ▶ Proširenje C-a preko pragmi u OpenMP maniru
 - ▶ Umereno univerzalan.
- ▶ I OpenACC i OpenCL žrtvuju nešto performansi za svoju brzinu.



OpenACC

UNIFORMNA TEHNOLOGIJA PRISTUPA AKCELERATORSKOM HARDVERU

Koja je veza između CUDA i ovoga?

- ▶ CUDA je specifična Nvidia tehnologija.
- ▶ OpenACC je otvorena specifikacija koja je namenjena da glatko podrži širok dijapazon akceleratorских uređaja.
- ▶ Šta karakteriše baš *akceleratorски* uređaj?
 - ▶ Posebna namena
 - ▶ Posebna hijerarhija memorije
 - ▶ Nečim ograničen transfer.
- ▶ OpenACC nam omogućava da pišemo optimizovan kod koji ovo uzima u obzir i sređuje transfer mesto nas.
- ▶ To znači da je promenljiva za nas jedan jedini objekat koji može biti u memoriji sistema i u memoriji akceleratorskog uređaja.

Opšta struktura

- ▶ OpenACC deli dosta osobina u svom dizajnu sa OpenMP i OpenMPI tehnologijama.
- ▶ Sa oba deli to što pokušava da bude, u što je većoj meri moguće, standardni programski jezik (to jest, C)
- ▶ Sa OpenMP deli oslanjanje na pragme.
- ▶ Kao i OpenMP, podrška mora da bude ubačena direktnu u kompajler.
- ▶ GCC 8 podržava OpenACC 2.5, GCC 7 OpenACC 2.0a

Nešto termina

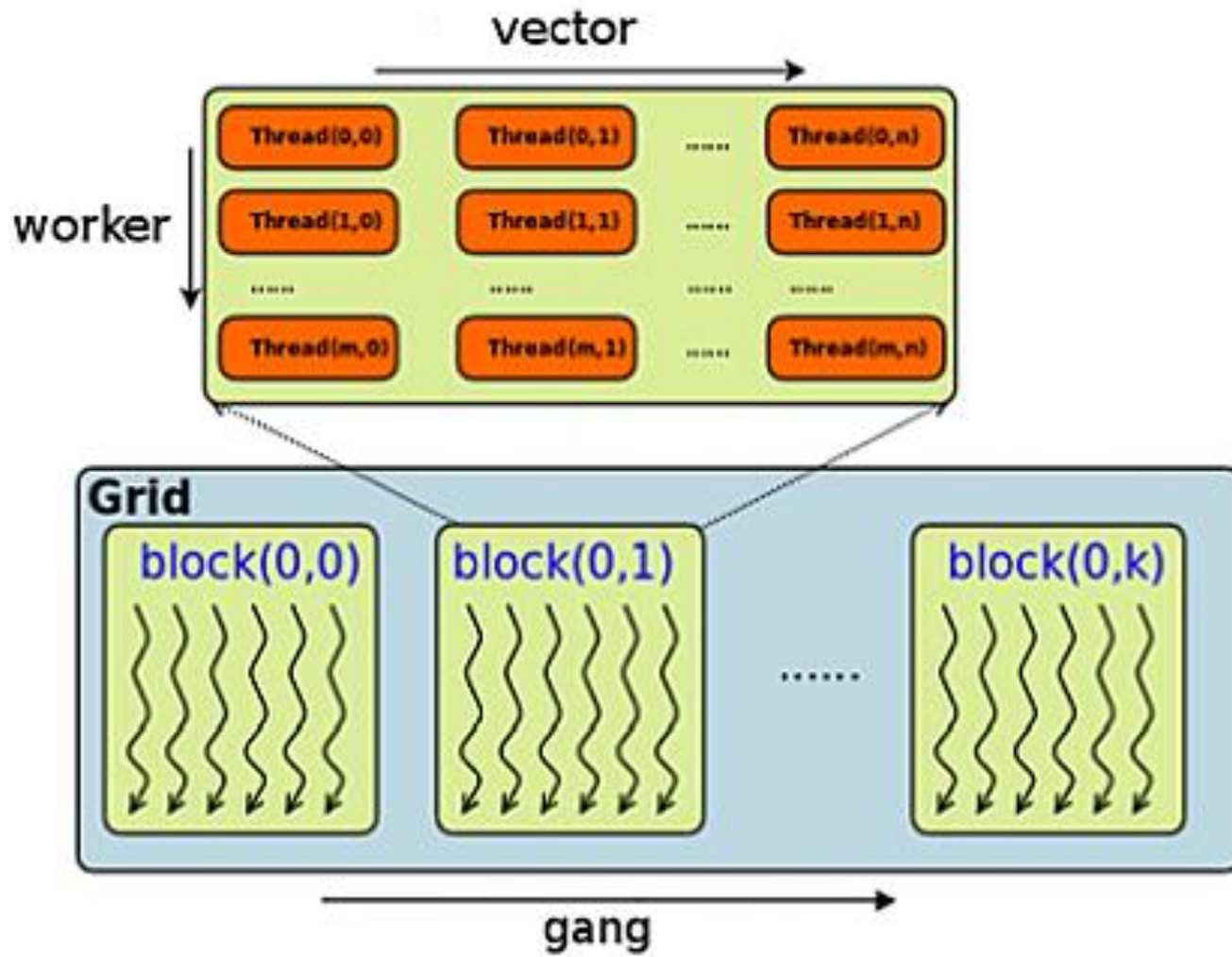
- ▶ Domaćin (eng. host)
 - ▶ Domaćin je računar na kome se izvršava kod i koji ima neki broj akceleratora
- ▶ Prebacivanje (eng. offloading)
 - ▶ Prebacivanje je mehanizam kojim se posebno specificirani delovi koda prebacuju na akcelerator.

Paralelizmi u OpenACC modelu

- ▶ OpenACC prepoznaje tri nivoa paralelizma kod akceleratora:
 - ▶ Grub (coarse grain)
 - ▶ Fin (fine grain), i
 - ▶ Individualan
- ▶ Grub paralelizam deli poslove na više resursa za proračun.
- ▶ Fin paralelizam tipičnu distribuira poslove na individualne procesne elemente.
- ▶ Individualan paralelizam je u okviru jednog PE, i izlaže paralelizam SIMD/vektor tipa direktno.
- ▶ Ovi nivoi u ACC-u se predstavljaju gang, worker, i vector paralelizmima.

Gang, Worker, Vector

- ▶ Gang (tim) je najviši nivo paralelizma za OpenACC model.
- ▶ Svaki uređaj akcelerator izvršava neki broj gang-ova koji imaju jedan ili više radnika (worker-a) koji izvršavaju ili individualne ili vektorske instrukcije.
- ▶ Vektorski paralelizam mora biti vrlo uniforman: ista stvar za više podataka u klasičnom SIMD maniru.
- ▶ Worker paralelizam je malo opušteniji ali deli iste resurse budući da svaki gang je, hardverski, definisan kao nešto što deli iste procesne resurse.
 - ▶ U NVidia svetu to znači da je Gang gotovo uvek vezan za jedan SM.
- ▶ Gang paralelizmi su nezavisni.



Gang,
Worker,
Vector

Režimi izvršavanja

- ▶ Izvršavanje OpenACC koda uvek počinje u Gang-Redundant režimu (GR). To je režim u kome svaki gang ima jednog radnika koji izvršava isti kod. Drugim rečima, nema paralelizma.
- ▶ Kada se u OpenACC kodu stigne to paralelnog segmenta prelazi se u Gang-Partitioned režim (GP) u kome su iteracije jedne petlje (ili, čak, više petlji) distribuirane između gang-ova, ali svaki gang i dalje ima samo jednog radnika koji radi na individualnim elementima u, tkzv. Worker-Single i Vector-Single režimima.
- ▶ Na GP nivou u WS/VS režimu mi smo veoma kao OpenMP i niti.
- ▶ Ako se to zatraži, moguće je aktivirati Worker-Partitioned režim gde se posao deli između radnika, te je u okviru radnika moguće zatražiti Vector-Partitioned režim

Režimi izvršavanja

- ▶ Može se o ovome misliti kao o OpenMP-u u kome je ugnježdavanje paralelizma ne samo moguće nego *neophodno* i to na tri nivoa.
- ▶ OpenMP niti su sve stvorene jednake, te nema performantnog razloga da preferiramo 3x2x2 niti u odnosu na 12 niti.
- ▶ U OpenACC postoji *hijerarhijska* podela na hardverskom nivou, te ukupne niti se najbolje dele na gang/worker/vector po tome koliko su podaci nad kojima radi nit vezani jedni za druge.
- ▶ Spuštati stvari na akcelerator nema puno smisla osim ako podaci nisu barem *malo* povezani i stoga prigodni. Spuštati stvari na vektorski nivo nema smisla osim ako (predvidivo) u pitanju nisu vektori.

OpenACC Hello World

```
#include <stdio.h>
#include <openacc.h>

int main(){
    printf("Supported OpenACC revision> %d.\n", _OPENACC);
    int count = acc_get_num_devices(acc_device_nvidia);
    printf("Found %d NVidia GPUs.\n", count);
    int n = acc_get_device_num(acc_device_nvidia);
    printf("default accelerator number is %d.\n", n);
    count = acc_get_num_devices(acc_device_host);
    printf("Found %d host processor.\n", count);
    n = acc_get_device_num(acc_device_host);
    printf("Default host processor number is %d.\n", n);
    return 0;
}
```

Kompajliranje OpenACC koda

```
veljko@HPC:~/c/acc1$ gcc hello.c -fopenacc -o hello
```

OpenACC sistemske promenljive

Promenljiva	Namena
ACC_DEVICE_TYPE	Tip uređaja koji se koristi za pokretanje koda. Opcije su NVIDIA, RADEON, i HOST
ACC_DEVICE_NUM	ID fizičkog akceleratoranskog uređaja koji se valja koristiti
ACC_PROFLIB	Biblioteka za profiliranje

OpenACC direktive

- ▶ Rade *neverovatno* slično kao u OpenMP-u.
- ▶ Sve počinju sa `#pragma acc`
- ▶ Zatim ide direktiva te onda klauzule koje parametrizuju direktivu.

Konstrukt paralelizma

- ▶ `#pragma acc parallel`
- ▶ Ovo se odnosi na blok koda i znači da se on izvršava u paraleli
- ▶ Podrazumevano je da se izvršava u GR (ne-baš-paralelnom) modu osim ako ne dodamo klauzule koje to spreče.
- ▶ Svaki paralelni blok se završava sa implicitnom sinhronizacijom.
- ▶ Paralelni blokovi **ne smeju da imaju grananje**.

Klauzule #acc parallel direktive

- ▶ `async[(integer)]`
 - ▶ Uklanja implicitnu barijeru na kraju i omogućava da procesor-domaćin radi stvari konkurentno sa kodom na GPU-u. Može da u zagradi ima brojku. Ta brojka je identifikator reda izvršavanja (activity queue) koji obrađuje stavke ovog bloka. Ta brojka se kasnije može koristiti kao parametar 'wait' klauzule da bi se omogućila sinhronizacija.
- ▶ `wait [(integer-list)]`
 - ▶ Blokira izvršavanje dok se navedeni redovi izvršavanja (kao brojevi u listi parametara) nisu kompletirali. Ako se ne navedu brojevi, čeka se da se sav asinhroni posao ne završi.

Klauzule #acc parallel direktive

- ▶ `num_gangs(integer)`
 - ▶ Traži (ali ne dobija garantovano) koliko gang-ova će se koristiti za podelu posla.
- ▶ `num_workers(integer)`
 - ▶ Kao gore, ali za radnike. Odnosi se samo na WP mod, naravno.
- ▶ `vector_length(integer)`
 - ▶ Kao gore, ali traži određeni broj vektorskih procesnih linija, u VP režimu. Na NVidia uređajima valja birati umnožak 32.

Kernels konstrukt

- ▶ `#pragma acc kernels`
- ▶ Odnosi se na blok, kao i `parallel`
- ▶ Ponašanje određuju klauzule
- ▶ Radi kao `parallel` (i prima iste klauzule) sa jednom ogromnom razlikom.
- ▶ Tamo gde je `parallel` zahtevao ručno podešavanje, kernels konstrukt analizira kod i *sam* distribuira posao.
- ▶ Drugim rečima ovo je magična 'make my code fast' direktiva koja zna da prepozna kada imamo, npr. tri ugnježdene for loop-a, da to podeli na gang/worker/vector nivou.

Upravljanje podacima

- ▶ Najvažnije usko grlo u programiranju akceleratora jeste u tome što je memorija akceleratora odvojena od memorije računara domaćina
- ▶ Sva komunikacija između ove dve memorije prolazi kroz magistralu koja je već preopterećena, skoro po definiciji.
- ▶ Ovo nije garantovano ponašanje: AMD ima uređaje za ubrzanje proračuna koji dele memoriju sa glavnim procesorom, te je overhead minimalan.
- ▶ Ali GPU-ovi su napravljeni tako kako jesu zbog potreba igara, i stoga, kada jednom učitate teksture i šejdere u radnu memoriju kartice, proračuni idu gotovo bez komunikacije sa ostatkom sveta.

Upravljanje podacima

- ▶ Automatizam je, u teoriji, moguć.
- ▶ U praksi? C/C++ je toliko kompleksan da je vrlo teško da statičkom analizom dođemo do zaključka koji deo memorije se kako koristi.
- ▶ Podrazumevano, OpenACC greši na stranu tačnog izvršavanja na račun brzine i kopira sve:
 - ▶ Sve strukture podataka se kopiraju sa domaćina na karticu
 - ▶ Radi se proračun
 - ▶ Kopiraju se svi podaci nazad.
- ▶ Automatsko kopiranje ove vrste samo radi bezbolno i implicitno ako se radi sa statički dimenzionisanim nizovima. Dinamički alocirana memorija se kopira rukom.

Ručno upravljanje memorijom

- ▶ Moguće je upravljati memorijom direktno kroz klauzule za upravljanje memorijom koje idu posle većine direktiva (parallel i kernel, recimo).
- ▶ Metod rada sa memorijom se bazira na brojanju referenci: svaka struktura memorije ili postoji u memoriji kartice ili ne. Ako postoji, broji se broj blokova u programu koji traži pristup toj strukturi kroz broj referenci.
- ▶ Čim se završi blok koji je koristio neku strukturu, broj referenci se smanji za jedan i ako stigne na 0, podaci se kopiraju nazad u glavnu memoriju.

Ručno upravljanje memorijom— klauzule

- ▶ `copy(variable-list)`
 - ▶ Copy klauzula specificira koji podaci trebaju datom paralelnom regionu. Promenljive koje nisu kopirane se kopiraju, a promenljive koje jesu se anotiraju sa povećanim brojem reference.
- ▶ `copyin(variable-list)`
 - ▶ Kao copy, ali se podaci kada broj referenci stigne do 0 ne kopiraju nazad, no se samo dealociraju.
- ▶ `copyout(variable-list)`
 - ▶ Kao copy, ali se ništa ne prenosi u memoriju. Broj referenci se povećava, i ako je 0 memorija se *alocira* ali se ništa ne prenosi na samu karticu. Kada broj referenci dostigne 0, podaci se kopiraju nazad.

Ručno upravljanje memorijom— klauzule

- ▶ `create(variable-list)`
 - ▶ Kao copy, ali niti prenosi podatke na karticu, niti ih vraća nazad.
 - ▶ Esencijalno alocira pomoćnu memoriju na kartici.

Specifikacija promenljivih

- ▶ variable-list se sastoji od promenljivih razdvojenih zarezima
- ▶ Promenljiva se sastoji od obaveznog identifikatora i opcionog dimenzionisanja
- ▶ Identifikator je ime promenljive
- ▶ Dimenzionisanje se sastoji od specifikacije delova strukture (multidimenzionalnog niza) koji se kopiraju i navodi se za svaku dimenziju kao specifikacija raspona
- ▶ Raspon se piše u uglastim zagradama i sastoji se od početnog indeksa i dužine razdvojene dvotačkom.
- ▶ Početna vrednost se može preskočiti (te je onda 0)
- ▶ Krajnja vrednost se može preskočiti na statički dimenzionisanim nizovima (te je onda ravna dužini niza).

Specifikacija promenljivih—primeri

- ▶ `a[5:t]`
 - ▶ Niz `a` počevši od 6-tog elementa sa ukupno `t` elemenata, tj. `a[5]`, `a[6]`, ..., `a[5+t-1]`
- ▶ `mat[:N][16:32]`
 - ▶ Region matrice `mat` koji obuhvata prvih `N` redova i sekcije kolona od po 32 elementa koje počinju sa 17-im elementom svakog reda.

Definisanje n-dimenzionalnih nizova u C/C++ na OpenACC-kompatibilan način

1. Statički dimenzionisani nizovi
 - ▶ Ograničenje: ako radimo sa ovim, naše specifikacije onoga što se prenosi *mora* da definiše kontinualni region memorije.
2. Pokazivači na statički dimenzionisane nizove
3. Statički alocirani nizovi pokazivača
4. Pokazivači na nizove pokazivača
5. Mešane alokacije

OpenACC i petlje: loop pragma

- ▶ #pragma acc loop
- ▶ Ide uvek ispred for petlje
- ▶ Mora biti ili kombinovana sa ili unutar parallel/kernel direktive
- ▶ Kernel direktiva ih pravi sama, ali možemo da mi ubacimo naše da kažemo sistemu tačno šta hoćemo.
- ▶ Kao i obično, klauzule određuju ponašanje ove pragme

Klauzule loop pragme

- ▶ `collapse(integer)`
 - ▶ Na koliko ugnježenih for petlji se odnosi ova direktiva. Podrazumevano je 1.
- ▶ `gang[(static:integer | *)]`
 - ▶ Aktivira paralelno izvršavanje po gang-ovima
 - ▶ Statički parametar nam omogućava da podesimo chunking kao i u OpenMP-u
 - ▶ * znači da chunking ostavljamo implementaciji
- ▶ `worker`
 - ▶ Aktivira paralelizam po worker-ima, tj. WP-mod

Klauzule loop pragma

- ▶ vector
 - ▶ Aktivira VP mod
- ▶ auto
 - ▶ Dobijemo šta inače radi kernel podrazumevano: statičku analizu koda i automatsku paralelizaciju.
- ▶ independent
 - ▶ Garantujemo kompajleru da je svaka iteracija naše petlje potpuno nezavisna od svake druge, te da se kod može paralelizovati mnogo više.

Klauzule loop pragme

- ▶ `reduction(operator:variable[,variable...])`
 - ▶ Kao i u svim okruženjima do sada, radimo sa redukcijom.
 - ▶ Operator može biti `+`, `*`, `max`, `min`, `/`, `|`, `&&`, `i | |`.
 - ▶ Promenljive koje se navode kao učesnici ne smeju a budu elementi niza ili strukture. To jest, samo obični skalari.

OpenACC i doseg promenljivih

- ▶ OpenACC jako vodi računa o dosegu: jako je bitno gde se promenljive deklarišu
- ▶ Na primer, sve promenljive u petlji su privatne za niti koja izvršava datu iteraciju petlje.
- ▶ Promenljive u bloku koji je markiran za VP su privatne za vektorsku liniju izvršavanja.
- ▶ Promenljive u bloku koji je markiran za WP su privatne za svakog radnika, ali deljene kroz vektorske linije izvršavanja.
- ▶ Možemo da iznudimo ovo ponašanje kroz 'private' klauzulu.

Eksplicitna sinhronizacija u OpenACC

- ▶ Valja je izbeći.
- ▶ Ali ponekad *mora*.
- ▶ E, pa, kada mora onda je proces ovakav:
- ▶ `#pragma acc atomic`
- ▶ Definiše statement koji je atomski, to jest, ne može se prekinuti.
- ▶ Tip atomskog izvršavanja se definiše klauzulom

Atomske klauzule

- ▶ read
 - ▶ Garantovan atomski pristup promenljivama sa desne strane operatora dodele.
- ▶ write
 - ▶ Garantovan atomski pristup promenljivama sa leve strane operatora dodele.
- ▶ update
 - ▶ Garantovano i čitanje i pisanje, ali samo u fiksnim formama koje koriste read-modify-write sekvencu kao što je prefix i postfix inkrement i dekrement kao i svi operatori forme `op=`
- ▶ capture
 - ▶ Poseban slučaj kada hoćemo da zaštitimo update-tip operacije sa desne strane znaka jednako, a sa leve vrednost koja hvata vrednost modifikovane promenljive ili pre ili posle modifikacije tj. `a = i++`;