Elektrotehniški vestnik 72(2): 91–96, 2005
Electrotechnical Review, Ljubljana, Slovenija

Načrtovanje porazdeljene arhitekture za simultano izvajanje programskih bremen

Sašo Greiner, Janez Brest, Viljem Žumer

Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, Smetanova 17, 2000 Maribor, Slovenija

E-pošta: saso.greiner@uni-mb.si

Povzetek. Učinkovito porazdeljevanje izvajanja programskega bremena je zahtevno opravilo tako s stališča modeliranja problema kot dejanske implementacije. Za porazdeljevanje splošnega bremena smo načrtovali abstraktno grozdno (cluster) arhitekturo, ki omogoča preprosto aplikacijo eksplicitnih mehanizmov paralelizacije na dejanski problem. V ta namen smo izdelali programski vmesnik, s katerim modeliramo programsko breme za porazdeljeno izvajanje na heterogenem računalniškem sistemu. Arhitektura zagotavlja transparentnost porazdeljenega izvajanja, implicitne ter eksplicitne sinhronizacijske mehanizme in komunikacijo po standardu za prenos sporočil MPI. Bistvo arhitekture in prednost pred sorodnimi različicami je prav v njeni abstraktnosti, ki omogoča osredotočen pogled na problematiko, medtem ko je nivo dejanskega porazdeljenega izvajanja odmaknjen – zanj v celoti skrbita arhitektura in programski vmesnik. Čeprav je komunikacija zasnovana z MPI, le-ta na stopnji aplikacijskega vmesnika sploh ni viden. Zaradi visoke stopnje sočasnosti smo izvajalno okolje na posameznem vozlišču implementirali z večnitnim ogrodjem. Niti služijo za sočasno izvajanje uporabniških in sistemskih (režijskih) poslov. Sistemski nivo arhitekture smo zasnovali robustno, kar pomeni, da lahko ob izpadu posameznega vozlišča ustrezno ukrepamo. Zaradi abstraktne zasnove arhitektura podpira množico vozliščnih topologij: gospodar-suženj, obroč, večnivojsko drevesno topologijo in druge.

Ključne besede: porazdeljeni sistemi, paralelno izvajanje, računalniške arhitekture, grozdne arhitekture

Designing distributed architecture for concurrent program execution

Extended abstract. Efficient parallelisation of program execution is an exacting task both from problem modelling viewpoint as well as that of actual implementation. For distributing general, non-specific program tasks we have developed an abstract cluster architecture that enables straightforward application of explicit parallelisation mechanisms to the selected problem. We have also implemented a C++ application programming interface (API) which serves as a tool for modelling the task executed in a heterogeneous computing system. Architecture guarantees transparency of parallel execution, implicit and explicit synchronisation mechanisms and communication through the widely accepted message passing interface standard (MPI). The main advantage in comparison with other contemporary systems for parallel execution based on MPI is the abstractness of architecture which allows greater focus on the problem itself and leaves out all the unnecessary details, such as synchronisation, task management, etc. Even though the communication model is based on MPI, the actual functionality of MPI within the application interface is not visible at all. To achieve the highest attainable degree of concurrency, we employ multithreaded execution environment on every node within the cluster. Threads are used for concurrent execution of user-defined and system tasks. Each node has its own execution environment (Fig. 2) and together they constitute a global execution engine. The execution environment within a single node consists of three independently executing threads: the one with the highest priority executes the assigned task, one executes actions mapped to user messages, and one maintains consistent state of the entire system. Message passing has been realised in three ways: interruptible, uninterruptible, and acknowledge-based (Fig. 1). In addition to local synchronisation which takes place on every node there is also a "global" synchronisation process on the level of entire architecture. System layer was designed with roboustness in mind, which means the architecture can be made immune to node fails. Abstract design of architecture core allows for a multitude of well known topologies such as master-slave, ring, tree, and others. Nodes which serve as basic units in cluster architecture have also been designed in an abstract manner. This means they can represent actual processing nodes or dispatcher nodes which can collect commands and messages and then forward them to remote nodes.

Prejet 9. junij, 2004
Odobren 18. januar, 2005

Key words: distributed systems, parallel execution, computer architectures, cluster architectures

1 Uvod

Proces paralelizacije je ponavadi vezan na domeno problema in je v splošnem že sam po sebi kompleksen problem [7]. Treba je izdelati sinhronizacijske in razvrščevalne mehanizme [14], ki ne bodo vezani na nobeno domeno, temveč bodo operirali na celi množici domen. Porazdeljeno (komponentno) izvajanje v okviru komercialnih sistemov je "najudobneje" vpeljala javanska arhitektura z vmesniki CORBA in RMI [12], vendar se java zaradi svojih slabosti, tj. hitrost izvajanja v virtualnem okolju, v masovnem paralelnem procesiranju ni posebno uveljavila. Prav zato so se razvila javanska razvojna ogrodja, ki omogočajo večjo učinkovitost porazdeljenega izvajanja [11].
Pristop, ki smo ga ubrali za razvoj tovrstne arhitekture, je abstrahiranje v strukturi in funkcionalnosti. Samo z abstrakcijo je namreč mogoče zajeti širok spekter problemov, nad katerimi izvajamo proces paralelizacije. Grozdno (cluster) arhitekturo smo zasnovali tako, da je vsak avtonomni sistem v arhitekturi predstavljen kot vozlišče z neko funkcionalnostjo. Sinhronizacija med posameznimi vozlišči poteka prek komunikacijskih kanalov. Po istih poteh (logično različnih) se prenašajo tudi podatki, ki so specifični glede na problem. Za izvedbo komunikacijskih mehanizmov smo uporabili standard MPI, ki je soroden PVM (Parallel Vector Machine) [8], vendar raziskave kažejo, da je MPI integriran v bistveno več porazdeljenih sistemov [6, 10] kot PVM.
Ker je stopnja sočasnosti pogojena z učinkovito izrabo procesnih in podatkovnih virov, smo na posameznem vozlišču uvedli večnitno izvajanje. Pri arhitekturi, ki ni vezana na dotično breme, je treba dejansko izvajanje bremena ločiti od režijskih poslov. Med režijske posle smo uvrstili sinhronizacijo in vse vrste razpošiljanja ter prejema sporočil. Sporočila smo razdelili na sistemska, ki so bremenu nevidna, in uporabniška, ki se lahko pošiljajo na zahtevo. Zaradi učinkovite izrabe procesnih virov je pošiljanje sistemskih sporočil implementirano neprekinjevalno. Večnitno okolje na vozlišču skrbi za karseda sočasno obdelavo režijskih poslov in bremena. Hkrati pa takšna zasnova povzroča precejšno kompleksnost celotnega izvajalnega okolja. Zaradi več, sočasno izvajajočih se tokov programa je namreč treba sinhronizacijo izvajati lokalno na posameznem vozlišču in hkrati na ravni celotne arhitekture. Gledano na arhitekturo kot celoto imamo torej koncept izvajalnega okolja s simultano večnitnostjo [4].
Pojem vozlišča smo opredelili povsem abstraktno in s tem omogočili, da z vozliščem predstavimo poljubno procesno entiteto, ki lahko dejansko izvaja breme ali pa služi zgolj za posredovanje zahtev drugim vozliščem. Slednje pride prav pri strukturiranju večnivojskih arhitektur z logično topologijo drevesa. Takšne arhitekture so primerne za veliko vozlišč, kjer že sama režija pomeni veliko časovno zahtevnost. Z abstraktno predstavitvijo arhitekture je mogoče snovati povsem specifične topologije, izmed katerih je treba poudariti gospodar-suženj (master-slave), obroč (ring) ter hibridne topologije. Zelo uporabna je tudi necentralizirana topologija, kjer je vsako vozlišče avtonomen sistem in operira neodvisno od drugih.
V drugem poglavju je opisana predstavitev vozlišč in njihova vloga v porazdeljenem sistemu. Tretje poglavje opisuje komunikacijske mehanizme med vozlišči in sporočila. V četrtem poglavju je podana zasnova večnitnega izvajalnega okolja posameznega vozlišča in celotne arhitekture. Naslednje poglavje je programski aplikacijski vmesnik in uporabo razredov. Šesto poglavje konča razpravo.

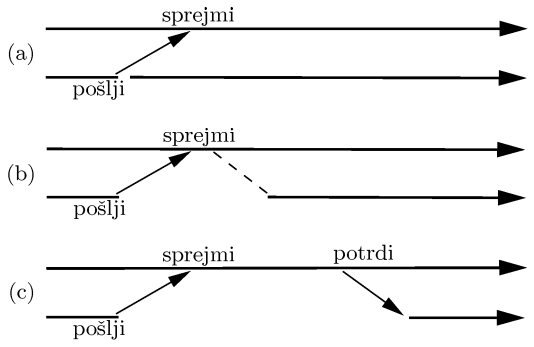
2 Vozlišča

Vozlišče (node) je osnovna procesna enota porazdeljenega sistema. Ker je predstavljeno abstraktno, lahko sodeluje pri izvajanju bremena ali pa igra vlogo "prenosnika" zahtev skupini vozlišč. Vozlišča se lahko razvrščajo v skupine za sodelovanje pri različnih problemih. Zaradi različnih vlog, se vsako vozlišče v vsakem trenutku nahaja v določenem stanju. Vsako vozlišče, ki sodeluje pri izvajanju bremena, mora biti živo. Mrtva vozlišča so tista, ki se fizično nahajajo v topologiji sistema, vendar niso del logične arhitekture. Živa vozlišča so lahko aktivna ali neaktivna. Aktivna so tista, ki dejansko sodelujejo pri izvajanju bremena. Poudariti velja, da je aktivnost posameznega vozlišča kontekstno odvisna. Neaktivno vozlišče v kontekstu izvajanja bremena α je lahko aktivno v kontekstu β, pri čemer sta α in β dva različna konteksta izvajanja ($\alpha \neq \beta$) oz. dve različni bremeni. Stanji, ki zadevata izvajanje, sta "zaposleno" in "prosto". Prvo pomeni, da je vozlišče zaposleno z izvajanjem bremena, drugo pa, da je na voljo za izvajanje. Razumljivo je, da mora biti vozlišče, ki ga želimo obremeniti, prosto. Arhitektura s katerokoli topologijo omogoča iskanje prostega vozlišča na višji abstraktni stopnji tako, da se naslednje prosto vozlišče določi avtomatično. Izbira je torej povsem transparentna, saj arhitektura v vsakem trenutku zagotavlja prosto vozlišče, če le-to v sistemu obstaja. Ne glede na to, ali vozlišče izvaja breme ali ne, režijski posli tečejo nemoteno. Vozlišče mora biti ob vsaki akciji sinhronizirano z vsemi sodelujočimi vozlišči. Ustrezati mora zahtevam, ki nastajajo na

sistemski ravni in tako ohranjati konsistentno stanje celotne arhitekture.
Manipulacija vozlišč poteka lokalno ali oddaljeno. Slednji primer je tipičen za topologijo gospodar-suženj, kjer gospodar operira nad oddaljenimi vozlišči. V takšnem primeru je treba imeti ustrezne pravice. Vozlišče lahko zaradi nekega razloga izpade in tako povzroči nekonsistenco v stanju sistema. Izpad posameznega vozlišča povzroči rekonfiguracijo celotne arhitekture. Ker se le-ta zgodi na sistemski ravni je za samo izvajanje bremena transparentna.

3 Komunikacija

Komunikacija med posameznimi vozlišči poteka prek komunikacijskih kanalov, ki se ustvarijo v času inicializacije sistema. Od topologije arhitekture je odvisno, kako se postavijo komunikacijski kanali in kako vozlišča komunicirajo med seboj [2]. Komunikacija je realizirana transparentno glede na mrežno topologijo, saj je zasnovana na svežnju protokolov TCP/IP. Sistemska raven temelji na arhitekturno neodvisnem standardu za pošiljanje sporočil MPI (Message Passing Interface) [3, 9]. Komunikacija je realizirana s konceptom sporočil. Vsako sporočilo, ki se lahko sprejme, se mora v fazi inicializacije registrirati na vozlišču. Sporočila so zaradi visoke stopnje fleksibilnosti, podobno kot vozlišča, zasnovana povsem abstraktno, saj specifično sporočilo v celoti implementira programer bremena. Ker sporočilo prispe z neko nedeterministično zakasnitvijo, ima vsako vozlišče sporočilno vrsto, v kateri se le-ta nabirajo. Tako prejem kot oddaja sta zaradi ohranjanja visoke stopnje paralelnosti realizirana neprekinjevalno oz. neblokirajoče, kar pomeni, da se izvajanje nadaljuje takoj po pošiljanju oz. prejemu. Seveda arhitektura omogoča pošiljanje uporabniških sporočil tudi prekinjevalno ali s potrditvijo na sprejemni strani. Implementirano sinhronizacijsko semantiko [1] vseh treh metod prikazuje slika 1.

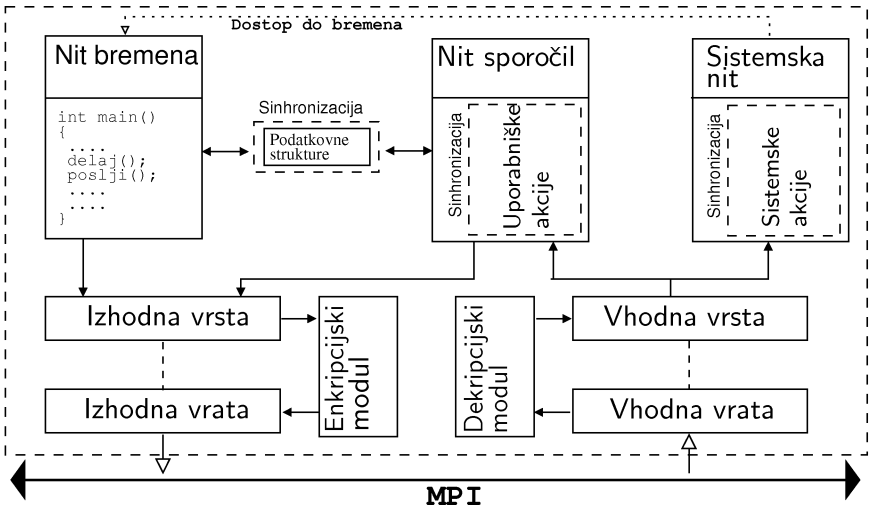


Slika 1. Mehanizmi pošiljanja v (a) neprekinjevalnem, (b) prekinjevalnem in (c) načinu s potrditvijo
Figure 1. Sending mechanisms for (a) uninterruptible, (b) interruptible, and (c) acknowledge-based modes

Visoka stopnja sočasnosti pri sprejemanju je dosežena z uporabo niti, v kateri se nahaja rutina za prejem in klasifikacijo vseh registriranih sporočil. Najpogosteje se uporablja pošiljanje sporočila iz enega v drugo vozlišče, torej eden enemu (unicast). Sistem podpira tudi način eden več vozliščem (multicast) in eden vsem (broadcast).
Sporočila pomenijo zgolj neko logično strukturo podatkov. Delijo se na sistemska in uporabniška. Sistemska so predefinirana in se uporabljajo za režijo in vzdrževanje konsistentnega stanja sistema. Uporabniško definirana sporočila so problemsko specifična tako po tipu kot po informaciji, ki jo prenašajo.

4 Izvajalno okolje

Izvajalno okolje je bistveni del arhitekture. Sestavljajo ga izvajalni mehanizmi na posameznih vozliščih. Konsistenca izvajanja je dosežena z uporabo sinhronizacijskih mehanizmov v okviru vozlišč in na ravni celotne arhitekture. Sinhronizacija izvajanja je dosežena posredno prek sistemskih in uporabniških sporočil. Na posameznem vozlišču je sinhronizacija dostopa do skupnih entitet realizirana z uporabo mehanizmov medsebojnega izključevanja (mutual exclusion) in semaforjev (semaphores). Funkcionalnost, ki je vezana na posamezno sporočilo, je omogočena s preslikavo signala prihoda sporočila na uporabniško akcijo. V kateremkoli trenutku med izvajanjem bremena lahko prispe neomejeno število registriranih sporočil. Sinhronizacijski mehanizem v tem primeru zagotavlja konsistenco med izvajanjem glavnega bremena in akcije kot posledice prejetega sporočila. Slika 2 prikazuje izvajalno okolje na posameznem vozlišču.
Izvajalno okolje je sestavljeno iz treh niti. Nit, v kateri teče breme, ima v razvrščevalniku najvišjo prioriteto. Nit z nižjo prioriteto skrbi za izvedbo akcij, ki so vezane na registrirana uporabniška sporočila. Ker je do skupnih podatkovnih struktur smiselno dostopati tako znotraj bremena kot iz akcij sporočil, je treba dostop sinhronizirati. Zaradi možnosti prepletanja uporabniških akcij je treba tudi te sinhronizirati. Nit z najnižjo prioriteto, tj. sistemska nit, opravlja sistemske akcije, ki so lahko redefinirane. Poleg obdelave sistemskih akcij ta nit skrbi še za konsistentno stanje celotnega sistema, saj ob izpadu nekega vozlišča sinhrono osveži vse sezname, ki so vezani na vsa vozlišča arhitekture. Razvrščanje niti po prioritetnem načinu je zelo pomembno, saj omogoča bistveno hitrejše odzivne čase na realno časovne dogodke kot npr. klasična strategija FCFS (First Come First Serve). Sistemski niti smo zaradi možnosti direktne manipulacije omogočili neposreden dostop do bremena. Za izvedbo niti smo uporabili



Slika 2. Arhitektura izvajalnega okolja na vozlišču Figure 2. Execution environment on node

standardne POSIX niti (pthreads). Z uporabo dejanske večprocesorske arhitekture SMP (symmetric multiprocessing) [5] z deljenim pomnilnikom lahko na vozlišču transparentno dosežemo povsem realno simultano izvajanje, saj posamezne niti tečejo na lastnih procesorjih. S tem se zmanjša tudi število preklopov kontekstov niti.

Uporabniška sporočila se lahko pošiljajo neposredno iz bremena ali znotraj obdelave nekega že prejetega sporočila. V slednjem primeru lahko sporočila verižimo (message chaining). Oddana sporočila se najprej pošljejo v izhodno vrsto, od tam pa na izhodna vrata, kjer se oddajo. Če je zahtevan kriptiran prenos, se vsebina sporočila filtrira skozi enkripcijski modul. Filtriranje je transparentno. Analogno kriptiranju je dekriptiranje, kjer se vsebina filtrira v osnovno obliko. Kriptiranje poteka po algoritmih *3DES* (trikratni DES s tremi različnimi ključi) ali *blowfish*, katerega osnovna prednost pred 3DES je hitrost. Vzpostavitev logične arhitekture poteka pravtako prek kriptiranih kanalov z uporabo parov privatnih in javnih ključev [13]. Sporočila se sprejemajo na vhodnih vratih in se nato prenesejo v vhodno vrsto, od koder prožijo sistemske in uporabniške akcije.

Funkcionalno je izvajalno okolje na vseh vozliščih identično, edina razlika se lahko pokaže v zmogljivosti, saj so posamezna vozlišča na splošno heterogena.

5 Aplikacijski programski vmesnik

Da bi bilo preprosto aplicirati porazdeljeno izvajanje na izbrani program, smo načrtovali abstraktni programski aplikacijski vmesnik (API). Aplikacijski vmesnik je zasnovan z objektnimi koncepti v jeziku C++ in v osnovi služi kot povezovalni sloj med nizkonivojskim logičnim slojem in programsko abstrakcijo. Med logičnim in fizičnim nivojem je standardni vmesnik za pošiljanje sporočil v heterogenem porazdeljenem sistemu MPI. Bistvena prednost dobro definiranega abstraktnega sloja je v transparenci heterogenosti, kar pomeni, da programer ne vidi arhitekturne raznolikosti. Aplikacijski vmesnik je zasnovan na razmeroma visoki programski ravni, ki skriva in enkapsulira vse vitalne podrobnosti delovanja. Ker je definiran na aplikacijski ravni, omogoča paralelizem na ravni grobe razčljenjenosti programskega bremena. Ker je treba paralelno izvajanje načrtovati eksplicitno, govorimo o eksplicitni paralelnosti.

Vozlišča so predstavljena z abstraktnim razredom **AbstractNode**, ki definira funkcionalnost, ki jo je treba implementirati na posameznem vozlišču. Konkretna vozlišča so torej predstavljena kot objekti, nad katerimi so definirane konkretne funkcionalnosti za poizvedovanje o stanju vozlišča, iskanje prostih vozlišč in prirejanje bremena vozlišču.

S podobnim pristopom predstavimo sporočila. Vsako uporabniško sporočilo mora implementirati abstraktne metode razreda **AbstractMessage**.

Dejansko breme mora implementirati abstraktne metode razreda **AbstractTask**.

Uporaba vmesnika za porazdeljeno izvajanje bremena je zelo preprosta in učinkovita. Najprej se ustvarijo objekti sodelujočih vozlišč in sporočila, s katerimi želimo operirati (npr. sporočilo za začetek dela in signalizacijo zaključka):

```
myNode = new Node();

msgBegin = new BeginMessage();
msgComplete = new CompleteMessage();
```

Sporočila registriramo na vozliščih:

```
myNode->listenTo( msgComplete );
myNode->listenTo( msgBegin );
```

Ustvarimo uporabniško breme (MyTask) in začnemo izvajati:

```
myNode->assignTask( new MyTask() );
myNode->execute();
```

Ob koncu izvajanja je treba vozlišče odstraniti takole:

```
myNode->shutdown();
```

ali

```
myNode->shutdownAll();
```

Uporabniška sporočila je treba definirati tako, da jih izpeljemo iz abstraktnega sporočila in dodamo specifične atribute in metode. Pošiljanje poteka iz izvornega vozlišča na eno ali več ciljnih vozlišč:

```
msgBegin->dispatch(myNode->getFreeNode());
```

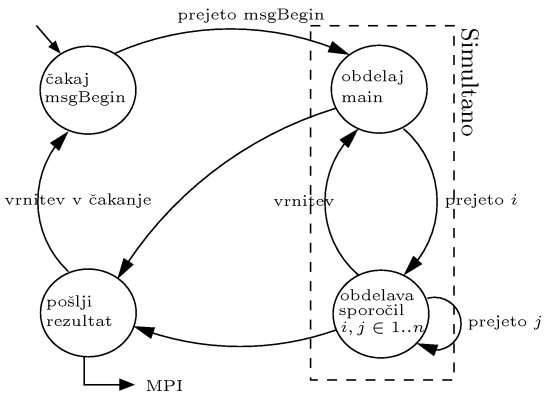
Vsem pošljemo sporočilo takole:

```
msgBegin->broadcast();
```

Breme modeliramo z razredom AbstractTask, kjer implementiramo glavno metodo **main** in metode za uporabniške akcije prejetih sporočil:

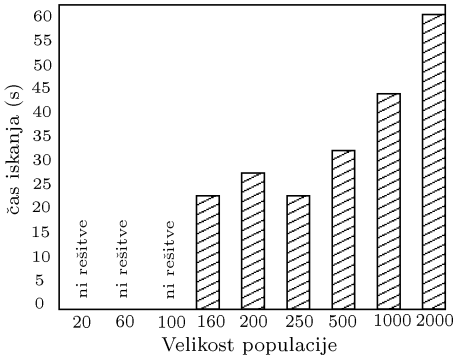
```
class MyTask : public AbstractTask {
  void onReceive(AbstractMessage msg){
      ...
      // programski kod akcije
      ...
  }
  void main() {
     ...
     wait( msgBegin );
     // kod glavnega bremena
     ...
  }
}
```

Diagram stanj za tipičen model izvajanja prikazuje slika 3.

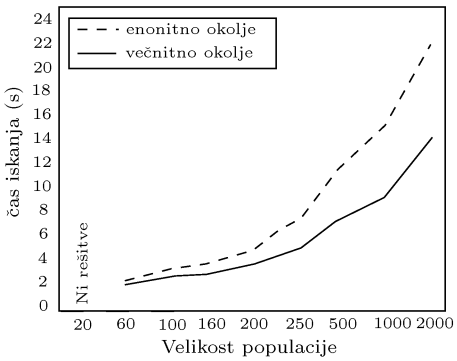


Slika 3. Diagram stanj
Figure 3. State transition diagram for a typical execution model

Arhitekturo smo med drugim uporabili za realizacijo problema simbolične regresije, ki je časovno zelo zahteven algoritem. Iz podanih funkcijskih vrednosti je treba poiskati funkcijo v simbolični obliki. Regresija se izvaja kot genetski algoritem, katerega populacija sestoji iz genetskih programov, ki opisujejo matematične funkcije [15]. Algoritem smo implementirali za klasično enoprocesorsko izvedbo in ga nato priredili za izvedbo na opisani porazdeljeni arhitekturi. Ker je genetski algoritem nedeterminističen, rešitev ni vedno najdena. Slika 4 prikazuje čase iskanja skozi 50 generacij v odvisnosti od velikosti populacije na enoprocesorskem sistemu. Čase ekvivalentnih primerov, ki so bili testirani na porazdeljeni arhitekturi, prikazuje slika 5. Izvajanje smo realizirali v enonitnem okolju s prekinjevalno strategijo pošiljanja sporočil in s polno funkcionalnostjo arhitekture, ki omogoča večnitno izvajalno okolje. Topologija arhitekture je bila gospodar-suženj z osmimi aktivnimi vozlišči.



Slika 4. Izvajanje simbolične regresije na enoprocesorski arhitekturi
Figure 4. Symbolic regression on uniprocessor architecture



Slika 5. Porazdeljena simbolična regresija v enonitnem in večnitnem okolju
Figure 5. Distributed symbolic regression in a single and multi-threaded environment

6 Sklep

Predstavili smo arhitekturo za porazdeljeno izvajanje splošnih programskih bremen. Prednost arhitekture je v njeni abstraktni strukturi, ki omogoča boljše načrtovanje porazdeljevanja programskega bremena, medtem ko so za problem nepomembne podrobnosti, kot sta sinhronizacija in upravljanje posla, prepuščene sami arhitekturi. Zato je mogoče zelo učinkovito modeliranje programskega bremena, saj načrtovalec le-tega ne vidi morebitne arhitekturne heterogenosti, sistemske ravni razpošiljanja sporočil ter sinhronizacije med režijskimi posli in bremenom. Abstraktne komponente omogočajo večjo prilagodljivost za različne problemske domene. Načrtovan je bil aplikacijski programski vmesnik, ki služi kot vmesna raven med arhitekturo in aplikacijo. Vmesnik smo implementirali kot sveženj abstraktnih in konkretnih razredov v jeziku C++. Proces modeliranja bremena smo razdelili na modeliranje dejanske problematike bremena in komunikacijo s pošiljanjem sporočil med posameznimi vozlišči. Ogrodje vozlišč je zaradi visoke stopnje sočasnosti izvajanja uporabniških in sistemskih poslov izvedeno večnitno. Za realizacijo večnitnega okolja izvajalnega okolja smo uporabili standardne niti POSIX. Arhitektura je bila testirana na kopici operacijskih sistemov Linux in BSD. Razširitev na druge platforme je transparentna, saj je vmesnik implementiran v jeziku C++ in vezan na standard za pošiljanje sporočil MPI. Nadaljnje delo vključuje optimizacijo obstoječe večnivojske arhitekture za veliko aktivnih vozlišč. Večnivojska arhitektura ima topologijo drevesa s posredniškimi vozlišči v notranjosti drevesa, kjer so poseben izziv sinhronizacijski mehanizmi ob izpadih notranjih vozlišč.

7 Literatura

[1] M. L. Scott, Programming Language Pragmatics, *Morgan Kaufmann Publishers*, 2000.

[2] S. Greiner, J. Brest, V. Žumer, Grozdna arhitektura za porazdeljeno izvajanje programskega bremena, *ERK 2003*, Ljubljana, Slovenija, september 2003.

[3] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. MPI, The Complete Reference, volume 2, The MPI Extensions, *MIT Press*, 1998.

[4] A. Kvas, Računalnika arhitektura s simultano večnitnostjo, magistrsko delo, FERI Maribor, 2001.

[5] D. A. Bader and Joseph J., SIMPLE: A Methodology for Programming High Performance Algorithms on Clusters of Symmetric Multiprocessors (SMPs), *Journal of Parallel and Distributed Computing*, 1999, pp. 92-108.

[6] P. Hadjidoukas and E. Polychronopoulos and T. Papatheodorou, Integrating MPI and Nanothreads Programming Model, *10th Euromicro Workshop on Parallel, Distributed and Network-Based Processing*, 2002.

[7] D. Bader and Bernard M.E. Moret and P. Sanders, Algorithm Engineering for Parallel Computation, citeseer.nj.nec.com/article/bader02algorithm.html, 2002.

[8] A. Geist and A. Beguelin and Jack Dongarra and W. Jiang and R. Manchek and V. Sunderam, PVM Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing, *MIT Press*, 1994.

[9] W. D. Gropp, E. Lusk, N. Doss, and A. Skjellum, A high peformance, portable implementation of MPI message passing interface standard, *Parallel Computing*, 22(6), 1996.

[10] I. Foster and N. Karonis, A Grid-Enabled MPI: Message Passing in Heterogeneous Distributed Computing Systems, *Proceedings of SC'98*, 1998.

[11] B. Carpenter and V. Getov and Glenn Judd and Anthony Skjellum and Geoffrey Fox, MPJ: MPI-like message passing for Java, *Concurrency: Practice and Experience*, 2000, pp. 1019-1038.

[12] J. Maassen and R. Van Nieuwpoort and R. Veldema and H. E. Bal and T. Kielmann and Ceriel J. H. Jacobs and Rutger F. H. Hofman, Efficient Java RMI for parallel programming, *Programming Languages and Systems*, 2001, pp. 747-775.

[13] V. Shoup, On formal models for secure key exchange, *Theory of Cryptography Library Record*, 99-12, 1999.

[14] J. Brest, V. Žumer, Aproksimacijski algoritem za statično razvrščanje opravil na večprocesorskem računalniku, *Elektrotehiški vestnik, 2000*, letnik 67, št. 2, str. 138-144.

[15] Z. Michalewicz, Genetic Algorithms + Data Structures = Evolution Programs, *Springer Verlag*, New York, 1996.

Sašo Greiner je diplomiral leta 2002 in je trenutno asistent na Fakulteti za elektrotehniko, računalništvo in informatiko Univerze v Mariboru. Njegova raziskovalna področja vključujejo objektno usmerjene programske jezike, prevajalnike, računalniške arhitekture ter spletno računalništvo v okviru informacijskih sistemov.

Janez Brest je diplomiral leta 1995, magistriral leta 1998 in doktoriral leta 2001 na Fakulteti za elektrotehniko, računalništvo in informatiko Univerze v Mariboru. Od leta 1994 je zaposlen v Laboratoriju za računalniške arhitekture in jezike, kjer se ukvarja s spletnim programiranjem, s paralelnim in porazdeljenim procesiranjem s poudarkom na razvrščanju opravil. Njegovo področje dela so tudi programski jeziki, ukvarja pa se tudi z optimizacijskimi raziskavami.

Viljem Žumer je redni profesor na Fakulteti za elektrotehniko, računalništvo in informatiko Univerze v Mariboru. Vodi Inštitut za računalništvo in Laboratorij za računalniške arhitekture in jezike. Področja, s katerimi se ukvarja, so programski jeziki, paralelno in porazdeljeno procesiranje ter računalniške arhitekture.