UNIVERSITATEA TEHNICĂ „GHEORGHE ASACHI” DIN IAȘI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
Domeniul: Calculatoare și Tehnologia Informației
Programul de studii: Tehnologia Informației

PROIECT DE DIPLOMĂ

Coordonator științific:
Ș.l.dr.ing. Silviu Dumitru PAVĂL

Absolvent:
Ioan CÎRJĂ

Iași, 2025

UNIVERSITATEA TEHNICĂ „GHEORGHE ASACHI” DIN IAȘI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
Domeniul: Calculatoare și Tehnologia Informației
Programul de studii: Tehnologia Informației

Gestionarea Colaborativă a Distribuției și Execuției Proceselor într-un Cluster OpenMPI

PROIECT DE DIPLOMĂ

Coordonator științific:
Ș.l.dr.ing. Silviu Dumitru PAVĂL

Absolvent:
Ioan CÎRJĂ

Iași, 2025

DECLARAȚIE DE ASUMARE A AUTENTICITĂȚII PROIECTULUI DE DIPLOMĂ

Subsemnatul _____,
legitimat cu ____ seria _____ nr. _____ , CNP_____
autorul lucrării _____

_____

_____

elaborată în vederea susținerii examenului de finalizare a studiilor de licență, programul de studii _____ organizat de către Facultatea de Automatică şi Calculatoare din cadrul Universității Tehnice „Gheorghe Asachi" din Iaşi, sesiunea _____ a anului universitar _____, luând în considerare conținutul Art. 34 din Codul de etică universitară al Universității Tehnice „Gheorghe Asachi" din Iaşi (Manualul Procedurilor, UTI.POM.02 - Funcționarea Comisiei de etică universitară), declar pe proprie răspundere, că această lucrare este rezultatul propriei activități intelectuale, nu conține porțiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislației române (legea 8/1996) şi a convențiilor internaționale privind drepturile de autor.

Data Semnătura

Cuprins

Gestionarea Colaborativă a Distribuției și Execuției Proceselor într-un Cluster OpenMPI

Ioan CÎRJĂ

Rezumat

Proiectul de diplomă/lucrarea de disertație va conține un scurt rezumat (abstract – termenul în engleză) de maximum o pagină, care expune ideile principale ale tezei, contribuția autorului la realizarea temei propuse precum și concluziile obținute. Formulați ideile principale ale lucrării în propoziții sau fraze cât mai simple, fără amănunte nesemnificative. Încercați să atingeți următoarele idei:

- contextul și motivația alegerii temei;
- obiectivele principale;
- metodologia utilizată;
- soluția propusă;
- rezultatele esențiale și principalele concluzii.

Rezumatul ar trebui să fie între 150-250 de cuvinte și să fie urmat de cuvinte-cheie (*keywords*, 5-7 termeni relevanți).

Recomandare neoficială: Rezumatul se compune, de obicei, după ce ați terminat redactarea lucrării de licență/disertație, după finalizarea capitolului Concluzii

Capitolul 1. Introducere

În contextul dezvoltării tehnologice hardware/software și al extinderii ariilor de aplicabilitate ale acestora, cerințele privind performanța și optimizarea în procesarea informațiilor sunt în continuă creștere în domenii precum cercetare științifică, inteligență artificială, simulări numerice, prelucrare de imagini, analiză a datelor, sau alte tipuri de procesare care, de obicei, necesită un set de algoritmi bine definiți. În aceste cazuri, abordările tradiționale bazate pe calcul și procesare secvențială - desfășurată pe un singur procesor sau nucleu din cadrul unei unități de procesare - pot da dovadă de ineficiență, astfel creându-se posibilitatea apariției unor timpi mari de execuție, utilizare ineficientă a resurselor disponibile sau alte blocaje. Aceste limitări reprezintă unul dintre motivele principale care au condus la dezvoltarea paradigmelor de calcul paralel și distribuit. Acestea oferă un mod diferit și eficientizat de a prelucra sarcinile din cadrul unui program și au capacitatea, în cazul utilizării corecte, de a crește performanța unei aplicații, comparativ cu varianta secvențială. Acest lucru a impus trecerea de la arhitecturi monolitice spre arhitecturi distribuite și paralele care să utilizeze la maxim capabilitățile celor două paradigme. Astfel, clusterele de calcul, alcătuite din noduri multiple interconectate prin rețele cu latență redusă au devenit o soluție des abordată pentru creșterea performanței prin fragmentarea unui volum mare de lucru în sarcini ce pot fi executate independent.

Apariția paradigmelor menționate anterior a condus, de asemenea, la dezvoltarea unui set de standarde și librării care să faciliteze, la nivel software, comunicarea în medii distribuite și paralelizate. Astfel, au fost dezvoltate o serie de soluții care facilitează paralelism, însă fiecare a propus propriul set de protocoale de comunicare și mecanisme de sincronizare. Din cauza acestei diversități, au apărut dificultăți precum portabilitate scăzută, lipsă de interoperabilitate și complexitate operațională. În consecință, proiectele de cercetare și aplicațiile industriale au suferit costuri de dezvoltare ridicate și întârzieri.

La începutul anilor ’90, a fost dezvoltat un nou standard, numit MPI (Message Passing Interface) [1], de către o serie de cercetători și universități, menit să rezolve problemele menționate anterior și să ofere un mod standardizat de a facilita programare paralelă în medii distribuite, devenind rapid un standard de facto asociat calculului paralel și distribuit. Au urmat o serie de implementări ale standardului MPI, care au luat rapid popularitate și au facilitat programe optimizate prin paralelism.

În mod tradițional, interacțiunea cu un sistem distribuit, precum un cluster de calcul, presupune o serie de configurări și utilizarea intensivă a liniei de comandă, ceea ce poate constitui o barieră pentru utilizatorii fără experiență în administrarea sistemelor distribuite. La nivel general, în cadrul clusterului, configurările pot fi constituite din setări la nivel hardware și de rețea, stabilirea numărului și tipurilor de noduri, alerea tehnologiilor de rețea și a parametrilor de transport, introducerea unui sistem de fișiere distribuit. De asemenea, este necesară stabilirea mediilor de execuție pentru tehnologiile utilizate, introducerea unui sistem de management al resurselor cu diferite politici de planificare și escaladare, aspecte ce țin de securitate și access, politici de monitorizare ș întreținere, iar exemplele pot continua. Din acest motiv, a apărut nevoia de a introduce instrumente care să abstractizeze, să simplifice astfel de interacțiuni și să ofere un mod centralizat de monitorizare și utilizare a resurselor. Astfel de instrumente se concentrează pe obținerea rezultatelor dorite cât mai rapid, cu un număr cât mai mic de configurări necesare. Tehnologiile utilizate, împreună cu arhitectura aleasă pentru dezvoltarea unor aplicații care să faciliteze accesul la resurse distrbuite, precum clusterele pot varia, însă lucrarea de față urmărește impactul serviciilor web.

Serviciile web pot fi definite ca un set de funcționalități expuse prin intermediul unor API-uri standardizate într-o rețea, accesibile prin protocolul HTTP(S) și descrise în formate precum REST sau SOAP. Ele permit sistemelor distribuite să comunice și să facă schimb de date într-un mod agnostic față de limbajul de programare sau platforma pe care sunt implementate. În practică, un serviciu web definește o colecție de operațiuni descrise formal prin documente sau specificații care pot fi apelate

de orice client compatibil. Această abstractizare face ca detaliile de implementare să fie ascunse și să poată fi modificate fără a perturba clienții, câtă vreme contractul de date rămâne neschimbat.

SOA (Service Oriented Architecture) este o paradigmă prin care aplicațiile sunt definite ca un set de servicii autonome, independente tehnologic, care interoperează pentru a facilita logica de afaceri a unei aplicații. Un element cheie în SOA este Enterprise Service Bus (ESB), care asigură rutarea mesajelor pe conținut, transformări de date în formate incompatibile, politici de securitate și calitate a serviciului și oferă, practic, o modalitate de decuplare a producătorilor de servicii de consumatorii lor.

SOAP (Simple Object Access Protocol) este un protocol utilizat în cadrul SOA și definește modalități de comunicare bazate pe XML și contracte WSDL, unde sunt descrise operațiunile și tipurile de date.

REST (Representational State Transfer) a fost definit de către Roy Fielding în 2000, ca fiind un stil arhitectural în construirea aplicațiilor distribuite bazate pe servicii web. În cadrul REST, fiecare entitate de domeniu devine o resursă identificată printr-un URI (Uniform Resource Identifier), iar operațiile asupra acestor resurse se realizează prin verbe HTTP standard, fără a impune un format de mesaje sau un fișier de descriere obligatoriu. Clientul nu manipulează niciodată obiectul direct, ci doar reprezentări ale acestuia (de obicei JSON sau XML). O noțiune importantă privind modul de operare REST este aceea că acest stil arhitectural este fără stare. Fiecare cerere HTTP trebuie să conțină toate informațiile necesare, iar răspunsul este complet detașat de cele anterioare. Avantajul este că serverele pot fi scalate orizontal foarte ușor, iar defectele unui nod nu afectează starea altuia. Acest lucru este benefic pentru aplicațiile care abstractizează sisteme distribuite precum clusterele, deoarece performanța și utilitatea acestora este direct proporțională cu capacitatea de scalare.

Din perspectiva utilizării MPI [1], serviciile web pot fi folosite pentru a automatiza procesul de distribuire sarcinilor, alocare al resurselor, execuție și monitorizare al rezultatelor. Astfel, procesul utilizării calculului distribuit și paralelizat poate fi simplificat, și se poate pune accentul pe dezvoltarea programului efectiv, sau pe analiza dorită, fără a ține cont de alte configurări. Acest lucru poate fi benefic și poate aduce optimizări vizibile în utilizarea la scară largă, în cadrul cercetărilor științifice, unde obținerea unui rezultat într-un timp cât mai mic este apreciată.

Un alt avantaj ce poate fi ușor evidențiat în dezvoltarea unor servicii web care să interacționeze cu un cluster este reprezentat de punerea la dispoziție a clienților a unui set de resurse computaționale și facilitarea utilizării lor. Clienți unor astfel de aplicații pot dispune de putere computațională adițională provenită din utilizarea infrastructurii hardware a clusterului. În funcție de capacitatea hardware a acestuia, se poate efectua un calcul complex, precum simularea unui algoritm într-un mod paralelizat și distribuit, care nu ar putea fi posibilă utilizând doar unitatea de calcul a clientului.

Dezvoltarea soluțiilor de interacțiune, vizualizare și monitorizare a calculului distribuit, precum procesele MPI, nu este frecvent întâlnită. Aceasta implică dezvoltarea și mentenanța unor resurse software și infrastructuri capabile să gestioneze calculul distribuit. Cu toate acestea, utilizarea serviciilor web oferă o nouă perspectivă în gestionarea proceselor MPI. Astfel, tema proiectului constă în dezvoltarea unor servicii web, alături de o interfață grafică pentru facilitarea accesibilității crescute în utilizarea unei implementări MPI, și anume OpenMPI, în cadrul unui cluster alcătuit din 21 de unități de procesare, capabile să execute procesare paralelizată și distribuită cu un nivel de performanță ridicat.

În cele ce urmează, capitolul II prezentă o serie de concepte teoretice privind modul de funcționare al MPI în cadrul clusterului și abordări similare ideii prezentate. Capitolul III prezintă în detaliu soluția propusă, din perspectiva arhitecturii, al componentelor și interoperabilitatea dintre acestea, precum și modul de comunicare cu clusterului asociat aplicației. Ulterior, în capitolul IV se vor prezenta o serie de rezultate preliminare obținute în utilizarea aplicației. În final, teza prezintă o serie de concluzii privind abordarea prezentată, precum și câteva direcții pentru viitoarele etape de dezvoltare.

Capitolul 2. Fundamentarea teoretică și documentarea bibliografică

2.1. Calculul paralel și distribuit

Calculul paralel și calculul distribuit sunt două paradigme fundamentale în contextul actual al dezvoltării informatice și al tehnologiilor emergente. Scopul general este efectuarea unui volum mare de procesări și calcule complexe într-un mod eficientizat, divizând sarcina principală în mai multe etape sau unități de execuție. Acestea, însă, diferă în modul de operare, structură și utilizare.

Calculul distribuit se referă la utilizarea unei rețele de unități de procesare care interoperează, cu scopul de a rezolva o sarcină comună. Unitățile, numite noduri, colaborează prin schimb de mesaje, împart resursele și sarcinile de procesare, și fiecare gestionează o parte a algoritmului sau a aplicației. Aceste noduri pot funcționa independent, dar acționează împreună ca un sistem unificat.

Procesarea paralelă presupune alocarea concurentă a unui set bine definit de sarcini către un set de procesoare sau nuclee de execuție din cadrul unei unități de calcul cu scopul de a fi executate simultan. Se pot distinge două moduri de abordare în utilizarea paralelismului: utilizarea memoriei partajate sau a memoriei distribuite.

În paralelismul bazat pe memorie partajată, toate unitățile de procesare, de obicei fiind reprezentate de mai multe fire de execuție în cadrul unui singur proces, accesează aceeași locație de memorie în cadrul execuției unui program. Comunicarea se realizează eficient, deoarece nu este necesar transferul de date utilizând o rețea. Este necesară, însă, implementarea unor mecanisme de sincronizare, precum semafoarele, pentru a controla eficient accesul la date și a preveni condițiile de cursă sau blocaje în cadrul execuției. De asemenea, scalabilitatea poate deveni dificilă, o dată cu creșterea numarului de fire de execuție. Este potrivit pentru sarcini care presupun comunicări frecvente și sincronizări în cadrul unei singure unități de procesare.

Paralelismul bazat pe memorie distribuită se referă la distribuirea mai multor sarcini către procese independente în cadrul unui program, fiecare având propriul spațiu de memorie și propriul set de date. Sincronizarea și comunicarea între acestea se realizează prin transmiterea de mesaje în cadrul unei rețele. Procesele sunt independente din punctul de vedere al resurselor utilizate, facilitând astfel simplitate in gestiunea și izolarea erorilor și posibilitatea scalării programelor. Ca și dezavantaje se remarcă complexitatea crescută și posibilitatea apariției latenței în comunicarea, din cauza faptului este necesară transmiterea de mesaje explicite între procese. Este potrivit pentru procesarea pe scară largă, precum modelarea vremii, antrenarea modelelor de inteligență artificială și high-performance computing (HPC).

Cele două abordări ale paralelismului pot fi combinate pentru a obține un nivel ridicat de performanță. În astfel de cazuri, paralelismul este obținut pe două niveluri: paralelism bazat pe memorie partajată în interiorul unui singur nod și paralelism bazat pe memorie distribuită în raport cu celelalte noduri din rețea.

În cazul în care optimizarea și eficiența constituie factori de interes înalt în abordarea unei probleme de procesare, cele două paradigme prezentate anterior pot coexista. Astfel, în cadrul unui cluster format din mai multe unități de procesare, putem aborda calculul distribuit, împărțind o sarcină către mai multe noduri, iar ulterior putem paraleliza fiecare nod, folosind un set de nuclee/procesoare locale.

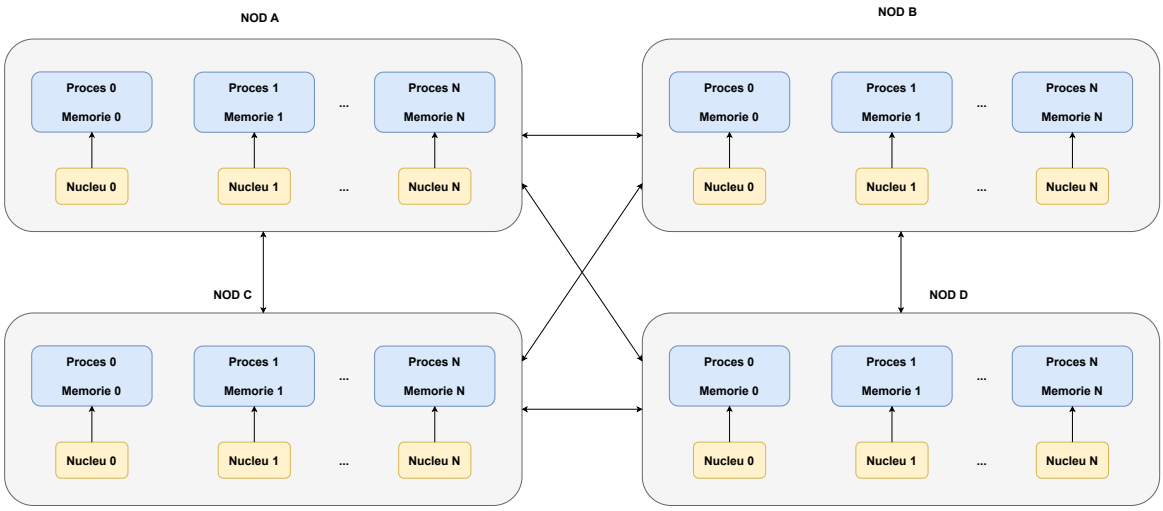

Figura 2.1. Procesare paralelă și distribuită

2.2. Standardul MPI

Odată ce utilizarea calculului paralel și distribuit a devenit mai frecvent întâlnită s-a remarcat lipsa unui mod standardizat pentru facilitarea comunicării între procese. Aceasta a rezultat în apariția unei serii de incompatibilități atât la nivel software, cât și hardware, precum și o lipsă de portabilitate. Din această cauză, programarea paralelă și portabilă a devenit dificilă. Astfel, la începutul anilor '90 a fost dezvoltat MPI, o soluție de standardizare și facilitare a programării paralele bazate pe memorie distribuită în cadrul clusterelor sau a supercomputerelor.

MPI [1] este un standard deschis, conceput pentru sincronizarea, transferul de date și controlul asupra proceselor ce rulează în paralel. Prima implementare a standardului MPI 1.x a fost MPICH, din cadrul Argonne National Laboratory (ANL) și Universitatea Mississippi. Ulterior au apărut pe piața implementări precum OpenMPI, LAM/MPI, LA-MPI, FT-MPI Microsoft MPI etc. Fiecare implementare se diferențiază prin nivelul de optimizare, suportul pe care îl oferă pentru diferite arhitecturi hardware, modul de configurare și performanța obținută în cadrul aplicațiilor.

OpenMPI, utilizat în cadrul lucrării de față, este o implementare open source a standardului, rezultată în urma colaborării mai multor instituții, cercetători și parteneri din industrie. Soluția software se concentrează pe aspectul calității și al performanței în cadrul aplicațiilor lansate în producție [2].

Standardul MPI definește procedurile și funcțiile utilizate pentru comunicarea între procese și specifică modul de operare al acestora. Implementările concrete, cum este OpenMPI, oferă codul efectiv care respectă aceste specificații și gestionează buffer-ele interne, optimizările pentru rețele de mare viteză şi suportul pentru diverse arhitecturi hardware.

2.2.1. Fundamentele comunicării MPI

MPI permite două tipuri de transmitere de mesaje: comunicare între procese individuale, numită comunicare punct-la-punct, și comunicare colectivă, între grupuri de procese. Comunicarea punct-la-punct are loc atunci când două procese interschimbă mesaje direct, fiecare cu rol de expeditor sau receptor specificând explicit destinaţia (rank-ul) şi identificatorul mesajului (tag-ul). Comunicarea colectivă implică simultan toate procesele dintr-un communicator, fără a desemna un expeditor sau un receptor singular; astfel, operaţiile cum ar fi `broadcast`, `scatter`, `gather` se desfăşoară cu toţi participanţii făcând schimb de date pe grupuri întregi de procese. Funcţiile colective sunt imple-

mentate intern, optimizat, şi nu necesită ca fiecare proces să apeleze explicit o funcţie de trimitere sau primire pentru fiecare destinaţie.

Comunicarea dintre procesele MPI poate fi realizată atât sincron, cât și asincron. Comunicarea sincronă, realizată cu funcții precum `MPI_Send` sau `MPI_Recv` expeditorul şi/sau receptorul rămân blocaţi pana la confirmarea primirii datelor. Comunicarea asincronă nu necesită o astfel de așteptare, ci permite procesului să continue execuţia imediat după iniţierea transmisiei, fără a aştepta finalizarea transferului; în acest caz, sincronizarea se face ulterior prin apeluri care verifică starea comunicării.

Gruparea proceselor se realizează utilizând topologii fizice și virtuale. Topologiile fizice reprezintă legăturile fizice, efective, dintre nuclee, procesoare și nodurile care execută calcul paralelizat. Topologiile virtuale constituie o structură abstractă, definită de programator, care organizează procesele MPI într-o anumită formă pentru a optimiza comunicarea, precum topologii de tip hipercub, plasă, inel, arbore, etc. Ele sunt utilizate în maparea structurii logice a proceselor asupra topologiei fizice. Topologiile virtuale oferă un nivel de abstractizare logică peste comunicarea dintre procese și o pot face mai ușor de gestionat, deoarece elimină, în unele cazuri, nevoie de a ține evidența manual asupra corespondenței sender-receiver din cadrul unei aplicații.

2.2.2. Joburile MPI

Numim un job MPI o instanță de execuție a unei aplicații paralele care utilizează standardul MPI pentru comunicarea între procese distribuite pe mai multe unități de calcul. Un job MPI este format dintr-un anumit număr de procese, fiecare cu un identificator unic, numit rank, care rulează același program și cooperează prin mesaje explicite pentru a rezolva o sarcină comună de calcul. Pentru a instanția un job MPI, avem nevoie de un executabil, rezultat în urma compilării codului MPI.

În cadrul unui cluster OpenMPI, invocarea utilitarului `mpirun` declanșează, în primă fază validarea parametrilor de lansare, după care se deschid canale de comunicație către fiecare nod specificat, unde daemonii locali `orted` sunt inițiați pentru a orchestra lansarea proceselor, iar plasarea fizică pe nuclee se realizează automat sau în funcție de configrație, dacă aceasta este specificată. După crearea efectivă a proceselor. În momentul apelului `MPI_Init`, aplicația trece în stadiul de comunicații punct-la-punct și colective, iar la final, dupa apelul `MPI _Finalize`, resursele sunt eliberate, daemonii `orted` sunt opriți, iar codurile de ieșire și datele despre execuție sunt raportate.

Definim, mai jos, o serie de parametri comuni folosiți pentru a lansa joburi MPI în execuție, folosind OpenMPI:

Parametru	**Descriere**
`-x ENV_VAR=VALUE`	Exportă o variabilă de mediu către procesele MPI.
`-hostfile path`	Specifică nodurile disponibile pentru execuţie.
`-host host1,host2,...`	Listează direct nodurile pe care să ruleze procesele.
`-np N`	Defineşte numărul de procese care vor fi lansate.
`-npernode N`	Numărul de procese MPI pe fiecare nod.
`-oversubscribe`	Permite rularea mai multor procese decât nucleele disponibile.
`-map-by slot,pe=N`	Distribuie procesele cu un anumit număr de sloturi (PE) per proces.
`-mca btl tcp, self`	Alege transportul TCP + loopback în mesagerie.
`-mca oob_tcp_dynamic_ipv4_ports PORTS`	Porturile TCP dinamice pentru OOB.

Parametru	Descriere
`-mca oob_tcp_static_ipv4_ports PORTS`	Specifică porturile TCP statice pentru comunicația OOB.
`-report-pid path`	Scrie PID-ul procesului `mpirun` într-un fişier.
`-report-bindings`	Afişează pe ce CPU/socket a fost fixat fiecare proces.
`-tag-output`	Prefixează fiecare linie de output cu rank-ul procesului.
`-timestamp-output`	Adaugă timestamp fiecărei linii de output.
`-prefix path`	Specifică prefixul de instalare Open MPI.
`-bind-to core`	Leagă fiecare proces MPI de un nucleu.
`-oversubscribe`	Permite rularea mai multor procese decât nucleele disponibile.
`-display-map`	Afişează harta de distribuţie a proceselor.
`-output-filename path`	Specifică fişierul de ieşire pentru loguri şi erori.
`-cpu-set x,y,z`	Alocă procesele specifice exclusiv pe core-urile specificate.

Tabelul 2.1. Descrierea parametrilor utilizaţi în comanda `mpirun`

2.3. Utilizarea modelului REST

Gestionarea joburilor MPI folosind servicii web care utilizează noțiuni ale standardului REST, presupune expunerea joburilor MPI ca resurse conceptuale accesibile prin URI-uri şi manipulabile prin metode HTTP standard (GET, POST, PUT, DELETE). Crearea unui job MPI se realizează printr-un simplu POST care furnizează un JSON complet auto-conținut cu parametrii doriți, iar interogarea stării sau anularea jobului corespund unor apeluri GET sau DELETE la URI-ul dedicat resursei.

În loc să se ofere clientului un mecanism direct de conectare la nodurile clusterului sau de pornire a daemonilor `orted`, interfața REST prezintă o abstracție uniformă: clientul transmite prin HTTP(S) un document JSON care descrie toate parametrii necesari execuției MPI (numărul de procese, fișierul hostfile, opțiunile de mapare și binding pe nuclee, variabilele de mediu necesare etc.). În acest cadru, nu este nevoie să se specifice vreun detaliu privind modul exact în care serverul negociază porturile TCP, inițiază procesele orted sau alocă fiecare proces MPI pe un nucleu fizic: toate aceste operațiuni rămân ascunse după nivelul API-ului REST, iar clientul nu interacționează decât cu reprezentări JSON și coduri de stare HTTP standard.

Fiecare cerere de trimitere a unui job conține toate datele despre acesta care sunt necesare pentru ca cererea să fie procesată de server. Ca răspuns se returnează un cod HTTP standard, dar care diferă în funcție de rezultatul jobului în urma execuției pe cluster sau a potențialelor erori.

La nivel intern, serverul trebuie însă să stabilească o conexiune SSH către nodul master al clusterului pentru a lansa efectiv comanda `mpirun`. Practic, API-ul poate fi văzut ca un middleware de abstractizare și vizualizare între client și cluster. Un dezavantaj al introducerii unui strat adițional între client și cluster poate fi introducerea timpilor de latență suplimentari înainte de demararea efectivă a jobului MPI, datorați propagării cererii prin HTTP(S) și SSH.

2.4. Soluții existente

Utilizarea tehnologiilor și serviciilor web pentru a expune interfețe menite să simplifice procesul de configurare și utilizare a unui sistem distribuit este abordată în multe lucrări științifice și articole, însă extinderea acestei idei asupra tehnologiei MPI reprezintă, totuși, un subiect care nu este abordat pe scară largă. În [3] este descrisă utilizarea serviciilor web în dezvoltarea unei aplicații care să faciliteze accesul studenților la sisteme distribuite și să îi sprijine în învățarea programării parale-

lizate, folosind MPI, în cadrul unor cursuri. Se prezintă cum aplicația poate ajută studenții să evite interfețele low-level, și cum aceștia pot trimite diferite joburi spre execuție către un cluster și apoi vor fi notați de către profesori, scopul fiind facilitarea unui proces optimal de învățare. Referința [4] prezintă, într-o manieră asemănătoare, utilizarea unei interfețe grafice în predarea calculului distribuit în cadrul unui cluster către studenți în cadrul unui curs dedicat sistemelor de operare. Ideea de a susține, din punct de vedere academic, studenții, în învățarea programării paralele folosind aplicații intermediare este abordată și în [5], unde se descrie experiența implementării unui sistem care să ofere acces de la distața studenților la calculatoare de înaltă performanță, însă lucrarea se concentrează pe aspecte comparative dintre MPI și OpenMP. Similar, lucrarea [6] se concetrează pe cum poate îmbunătăți performanța academică privind paralelismul o platformă web.

Limitarea principală pe care sistemele menționate mai sunt o tratează este reprezentată de potențialele dificultăți pe care studenți le pot avea în utilizarea programării paralele și a librăriilor aferente, alături de nevoia existenței unei infrastructuri fizice efective care să fie pusă la dispoziția lor în cadrul cursurilor care se concentrează pe calcul paralelizat și distribuit cu tehnologii precum MPI, dar și alternative bazate pe memorie partajată, ca OpenMP. Clasa de studenți țintită în mod special este reprezentată de studenții din cicluri universitare inferioare, care încă nu au suficientă experiență pentru a lucra cu ușurință cu sisteme complexe, precum un cluster, în cadrul liniei de comandă. Utilizarea unui astfel de sistem poate avea însă și alt public țintă.

Capitolul 3. Soluția propusă

3.1. Specificații privind aplicația propusă

Dezvoltarea unei soluții bazate pe servicii web, care să faciliteze accesul la un cluster OpenMPI și să permită lansarea de joburi spre execuție și să poate vizualiza rezultatele în timp real presupune utilizarea unei combinații de tehnologii care să trateze diferitele probleme ce pot apărea. În primul rând, avem nevoie de un sistem de gestiune al utilizatorilor. Aplicația este realizată cu scop general, și nu țintind un public anume, precum studenții, deși poate veni și în ajutorul acestora. Aceasta trebuie să interacționeze cu clusterul într-un mod controlat pentru a putea preveni eventualele blocaje. Pentru acest lucru s-a introdus, în cadrul serviciilor web, logică de autentificare și autorizare, care să restricționeze accesul la aplicație în cazul unui utilizator necunoscut, iar la nivelul interfeței utilizator, aplicația trebuie să conțină secțiuni dedicate pentru crearea unui cont și pentru autentificare.

Clusterul necesită configurări prestabilite și instalarea tehnologiilor necesare și setarea unor metode de autentificare, precum SSH. Aplicația va comunica doar cu nodul master, urmând ca distribuția resurselor și utilizarea lor să fie realizată prin intermediul acestui nod. Se necesită de asemenea, valabilitate din punctul de vedere al conexiunii în rețea, spațiu de stocare aferent joburilor pe cluster, memorie RAM, etc. Aplicația trebuie să monitorizeze dacă nodurile sunt valabile în mod constant și să actualizeze interfața utilizator, pentru a evita cazul în care se trimite cererea de creare a unor procese către noduri inactive fizic. Atunci când utilizatorul va crea un job, se trimite o cerere de conectare către nodul master din cluster; dacă conexiunea este realizată cu succes, jobul poate fi instanțiat.

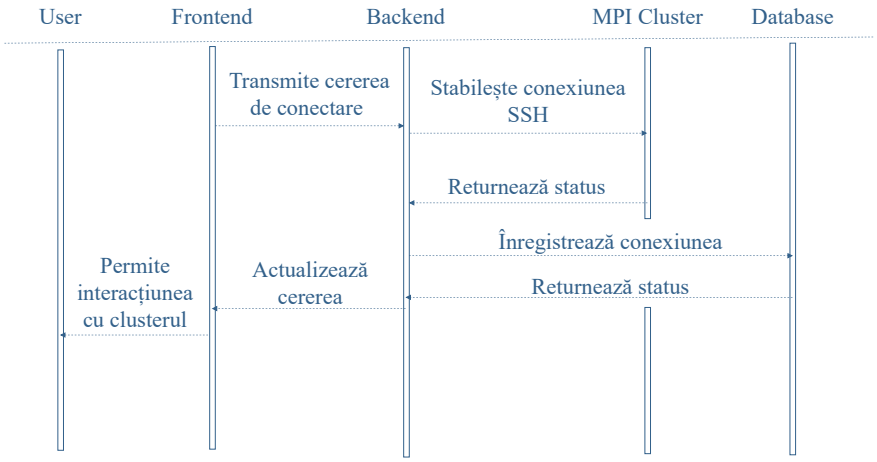

Figura 3.1. Conectarea la cluster prin SSH

La nivelul interfeței utilizator este necesar ca aplicația să restricționeze accesul la resurse dacă un client nu deține drepturile de acces corespunzătoare. În urma autentificării, utilizatorul va putea naviga pe pagini dedicate încărcării unui job MPI, unde va putea configura un job și a-l trimite în execuție. Configurația unui job este următoarea:

- Nume și descriere.
- Executabilul propriu-zis, rezultat în urma compilării codului MPI.
- Distribuția și numărul proceselor pe nodurile din cluster (Exemplu: 5 procese pe nodul A, 3 procese pe nodul B, cu un total de 8 procese)
- Parametri de configurare, precum: `map-by`, `rank-by`, `oversubscribe`, `bind-to`, `display-map`, etc.

- Opțiuni adiționale secundare, precum notificarea atunci când un job este finalizat

Pentru a putea menține evidența joburilor și a facilita vizualizarea rezultatelor trebuie ca fiecare job să poate fi corect clasificat, în funcție de starea sa. Din momentul instanțierii unui job MPI în cadrul aplicației, acestuia i se asociază una dintre următoarele stări, la un moment dat:

- pending - jobul a fost salvat cu succes, iar aplicația așteaptă eliberarea resurselor sau respectarea restricțiilor accesului la resurse ale unui cont utilizator
- running - jobul a fost trimis către cluster și execuția a pornit; se așteaptă rezultatele
- completed - jobul a fost finalizat cu succes, rezultatele sunt disponibile în interfața utilizator
- failed - jobul a întâmpinat erori în procesul de execuție și a fost terminat
- killed - utilizatorul a decis oprirea unui job și terminarea proceselor din cadrul acestuia

La nivelul serviciilor web, acestea trebuie să fie capabile să getioneze mai multe cereri de creare a unui job, din partea unor utilizatori diferiți. De asemenea, trebuie să gestioneze cazuri de supraîncărcarea ale clusterului sau să le prevină pentru a menține utilizarea acestuia. Astfel, aplicația trebuie să conțină un algoritm care să evalueze starea clusterului și să decidă dacă execuția unui job are loc sau nu. Dacă execuția are loc, serviciile web trebuie să trimită datele aferente unui job către nodul master din cluster, care va lansa jobul MPI în execuție. Supraîncărcarea clusterului cu un număr prea mare de joburi la un moment dat, provenite de la unul sau mai mulți utilizatori este oprită prin gestionarea unor permisiuni asociate fiecărui cont din cadrul aplicației. Acest lucru este realizat de către un cont administrator.

Serviciile web trebuie să gestioneze logica de interacțiune cu clusterul OpenMPI, să comunice în timp real rezultatele și să monitorizeze permanent starea joburilor. Pentru acest lucru s-au folosit o combinație de strategii de implementare, precum alocarea de resurse dedicate pentru procesarea și așteptarea rezultatelor unui job, dar și implementarea unor servicii recurente care interoghează starea și resursele clusterului pe parcursul rulării aplicației.

Backendul monitorizează în mod constant următoarele metrici pentru a deduce disponibilitatea clusterului de a prelua joburile unui utilizator specific:

- Numărul maxim de procese simultane permis pentru un utilizator.
- Numărul maxim de procese per nod, per utilizator, pentru a preveni supraîncărcarea resurselor locale.
- Numărul maxim de joburi care pot fi în execuție simultan pentru un utilizator.
- Numărul maxim de joburi aflate în așteptare în coadă pentru un utilizator.
- Durata maximă permisă pentru rularea unui job.
- Lista nodurilor pe care utilizatorul are dreptul să ruleze joburi.
- Numărul maxim de noduri ce pot fi alocate unui singur job.
- Numărul total de joburi (active sau în așteptare) permis simultan pentru un utilizator în sistem.

La nivel de ansamblu, backendul monitorizează utilizarea resurselor ținând cont de toți utilizatorii în același timp. Astfel, verifică:

- Numărul total de joburi aflate în execuție simultan pe întregul cluster.

- Gradul de utilizare a fiecărui nod, în funcție de numărul total de procese active.
- Numărul de joburi aflate în așteptare la nivelul întregului cluster.
- Numărul total de procese active (indiferent de utilizator) în raport cu capacitatea globală a clusterului.

De îndată ce un job este salvat, acesta va fi marcat cu starea pending. Backendul trebuie să interogheaze recurent starea joburilor și a clusterului, pentru a putea lua o decizie privind momentul în care jobul va trece în starea running. Mai exact, algoritmul de planificare verifică simultan condițiile menționate mai sus, iar dacă acestea sunt îndeplinite, jobul va putea fi executat.

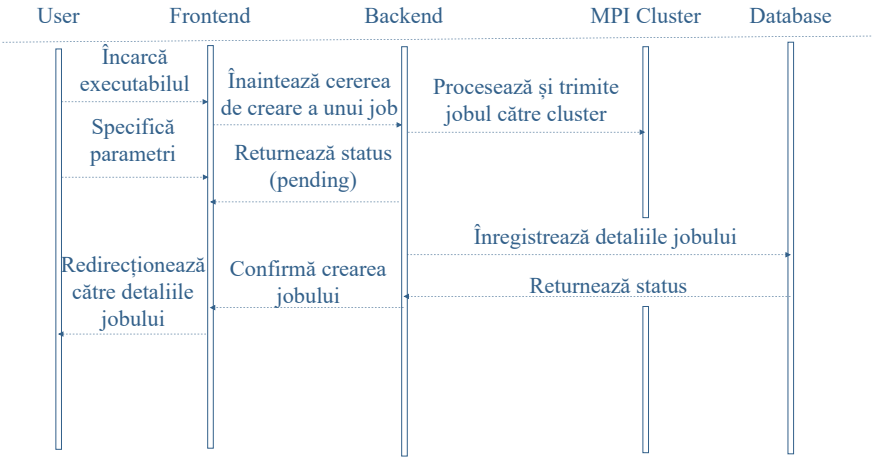

Figura 3.2. Încărcare unui job MPI

Interfața trebuie să fie reactivă și să permită actualizarea în timp real a statusului fiecărui job, nu doar apeluri REST clasice. În timp ce endpoint-urile REST acoperă operațiunile CRUD (creare, citire, actualizare, ștergere) pentru joburi și metadate, acestea nu pot furniza imediat informații despre progresul intern al unui job sau despre erorile de configurare și execuție apărute pe cluster. De aceea, pe lângă apelurile REST standard, este necesară integrarea unui canal de comunicare asincron (WebSockets) care transmite mesajele de progres și alertele de eroare către client. Astfel, atunci când un job trece de la starea "pending" la "running" sau apare o greșeală la generarea hostfile-ului ori în timpul execuției mpirun, notificarea ajunge instant în browser, iar statusul este actualizat.

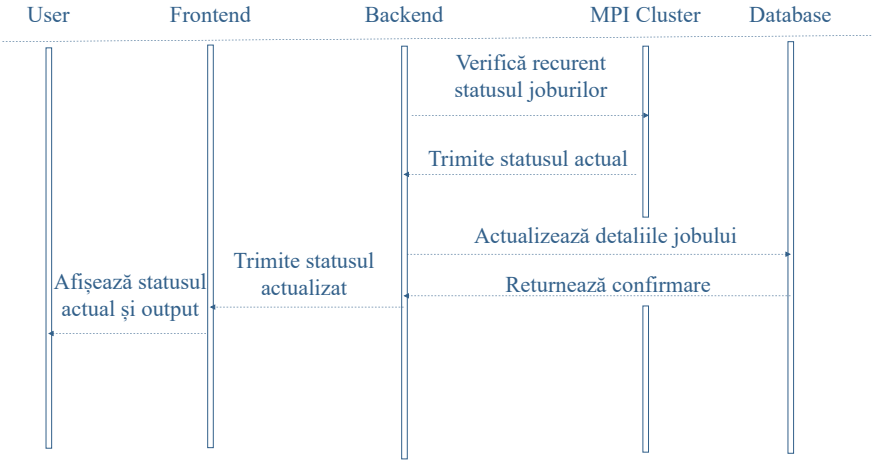

Figura 3.3. Verificarea statusului joburilor MPI

3.2. Arhitectura aplicației și tehnologiile utilizate

Arhitectura aplicației urmează un model client-server în care interfața grafică și microserviciile [7] comunică prin protocoalul standard HTTP(S) și un canal asincron pentru actualizări în timp real. Toate solicitările provenind din aplicația frontend trec mai întâi printr-un punct central de rutare (API Gateway), care validează cererile și le direcționează către serviciul potrivit de pe backend. În paralel, pentru afișarea progresului joburilor MPI și rapoarte instant, este deschis un canal WebSocket între frontend și backend, asigurând astfel că starea și erorile apar imediat în interfață fără reîncărcări suplimentare.

Pe partea de frontend, am utilizat Angular pentru a construi o aplicație reactivă, modulară, capabilă să afișeze formulare de configurare și liste de joburi cu statusuri dinamice. Angular comunică cu backend-ul prin apeluri REST și folosește WebSocket-uri pentru notificări în timp real. Autentificarea și autorizarea utilizatorilor se face pe baza tokenurilor JWT (JSON Web Tokens).

Pe partea de backend, responsabilitățile sunt distribuite între șase microservicii separate, fiecare construit cu FastAPI. Fiecare serviciu se ocupă de o funcționalitate clară – autentificare, orchestrarea joburilor MPI, monitorizare, scanare de fișiere executabile, gestionare utilizatori și raportare de metrici – și comunică cu celelalte strict prin API-uri REST.

Conexiunile către cluster-ul de calcul MPI folosesc SSH programatic și permit lansarea comenzilor necesare fără ca utilizatorii să interacționeze direct cu linia de comandă. Astfel, backend-ul garantează securitate, flexibilitate și posibilitatea de a răspunde rapid la schimbările din mediul distribuit.

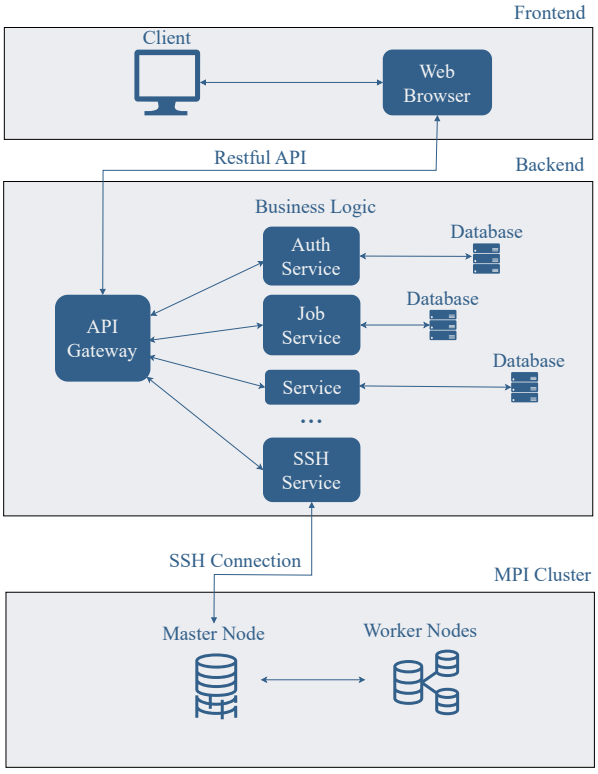

Figura 3.4. Arhitectura generală a aplicației

Pentru persistența datelor am folosit două soluții non-relaționale, Firebase Realtime Database și MongoDB. Firebase Realtime Database se ocupă de gestiunea conturilor utilizator. MongoDB este utilizat pentru procesarea joburilor, stocarea datelor unui job, încărcarea clusterului la un moment dat din punctul de vedere al joburilor, valabilitate nodurilor din punct de vedere hardware, etc. Ambele folosesc formate JSON standardizate, ceea ce mi-a permis să adăug câmpuri noi pe parcurs fără

operațiuni de migrare. În plus, în acest proiect nu avem nevoie de interogări complexe SQL/PL SQL, așa că nu beneficiem la maxim de capacitățile relaționale.

Aplicația client–server trimite toate solicitările de lansare și monitorizare a joburilor MPI către un singur nod „master” (C00). Din moment ce utilizatorul configurează un job prin interfața web, microserviciul de orchestrare deschide o conexiune SSH exclusiv către C00 pentru a executa comanda mpirun. Astfel, mpirun rulează numai pe C00, fără a expune direct celelalte noduri. În funcție de setările de topologie și numărul de procese definit de utilizator, C00 invocă daemon-ul `orted` pentru a iniția procesele MPI pe nodurile secundare.Comunicarea dintre C00 și nodurile secundare se realizează prin porturi TCP, definite dinamic de către backend , stabilite după cum urmează: înainte de lansarea jobului, C00 calculează un interval de porturi (de exemplu 5000–5100) și transmite acele valori în parametrii MCA (Modular Component Architecture) ai Open MPI. Nodurile secundare, la rândul lor, deschid conexiuni TCP către C00 pe porturile respective pentru a negocia setările de comunicare internă. În acest fel, straturile superioare ale aplicației nu trebuie să știe nimic despre configurația internă a clusterului: ele interacționează doar cu C00 prin SSH, iar Open MPI și orted se ocupă automat de restul operațiunilor de distribuție și comunicare inter-noduri.

Implementarea serviciilor web urmărește o reprezentare bazata pe standardul REST, cu un model client-server și comunicări uniforme de tip HTTP(S) (GET, POST, PUT, DELETE) și reprezentări standard ale resurselor, precum JSON. Solicitările cleintului trebuie să conțină toate informațiile necesare pentru a putea fi înțelese și procesate de server, iar răspunsurile oferite de acesta vor conține coduri de stare HTTP pentru detalierea rezultatului solicitării. microservicii cleint server http aplciatie web etc

3.3. Aspecte legate de securitate

Din cauza faptului că aplicația expune accesul la un cluster de calcul, trebuie să avem în vedere faptul că utilizarea acestuia poate fi supusă unor riscuri de securitate. Toate fișierele binare încărcate de utilizatori sunt scanate automat prin VirusTotal, iar doar executabilele aprobate sunt transferate pe nodul master și lansate prin SSH.

Autentificarea utilizatorilor se realizează prin token-uri JWT, astfel încât fiecare cerere către API-uri trebuie să conțină un token valid și nerestricționat. Conexiunea SSH folosită de microserviciu rulează sub un cont cu drepturi minime, care nu are permisiuni de administrare și poate doar să inițieze procese mpirun și orted; în felul acesta, chiar dacă cheia SSH este compromisă, nimeni nu poate modifica configurații de sistem sau instala pachete noi pe nodurile clusterului.

În paralel, există un cont de administrator cu privilegii speciale prin care se pot suspenda conturi de utilizator, seta limite de joburi, procese și timp de execuție la nivel de cluster, iar orice modificare a acestor politici este propagată instant în regulile de validare ale microserviciilor.

3.4. Dezvoltarea interfeței utilizator

3.5. Dezvoltarea microserviciilor

3.5.1. Componenta de gestiune a joburilor

Cel mai complex microserviciu este cel care gestionează încărcarea joburilor și vizualizarea rezultatelor. Pe lângă logica de încărcare a unui job în cadrul clusterului, acesta implementează monitorizarea continuă a statusului joburilor la nivelul bazei de date și al clusterului și implementează algoritmul de comparație care decide dacă un job poate fi instanțiat sau nu pentru a preveni supraalocarea.

3.5.1.1. Serviciul recurent de monitorizare

În momentul în care microserviciul destinat încărcării joburilor în cluster și așteptării rezultatelor este pornit, acesta porneste un serviciu secundar, care rulează în paralel, folosind `create_task`

din librăria `asyncio`. Scopul acestuia este de a decide efectiv dacă, la un moemnt dat, clusterul este disponibil să preia execuția unui job marcat ca `pending` în baza de date. Practic, acest serviciu trimite joburile către cluster, bazându-se pe decizii luate în funcție de limitările setate privind starea generală a clusterului, și anume: numărul joburilor aflate în starea `pending` și `running`, utilizarea și cererea fiecărui nod în parte și utilizarea totală a clusterului. Algoritmul rulează recurent, la un interval de monitorizare numit `MONITOR_INTERVAL`. Pentru a ne asigura că rezultatele execuției unui job pot fi obținute cât mai rapid, fără a introduce latențe suplimentare, și practic, a obține un răspuns în timp real, valoarea intervalului de monitorizare trebuie să fie foarte mică.

Algoritmul 3.1 Programarea joburilor în stare `running`

procedure SCHEDULEPENDINGJOBS
  MAX_RUNNING ← limită joburi în execuție
  MAX_NODE_USAGE ← limită procese per nod
  MAX_PENDING ← limită joburi în așteptare
  MAX_TOTAL_USAGE ← limită utilizare totală cluster
  loop
    așteaptă `MONITOR_INTERVAL` secunde
    pendingJobs ← obține toate joburile cu status `pending`
    runningJobs ← obține toate joburile cu status `running`
    clusterUsage ← calculează utilizarea procese pe fiecare nod
    $totalUsage \leftarrow \sum_{n \in clusterUsage} clusterUsage[n]$
    if $totalUsage \geq MAX\_TOTAL\_USAGE$ **then**
      continue ▷ S-a atins utilizarea totală maximă
    if $|runningJobs| \geq MAX\_RUNNING$ **then**
      continue ▷ S-a atins numărul maxim de joburi `running`
    if $|pendingJobs| \geq MAX\_PENDING$ **then**
      continue ▷ S-a atins numărul maxim de joburi `pending`
    jobStarted ← False
    for all *job* ∈ *pendingJobs* **do**
      if *jobStarted* = True **then**
        break ▷ S-a lansat deja un job în această iterație
      decodează *hostfile* din job → construire dicționar *nodeRequest* de forma {nod: sloturi}
      exceedsLimit ← False
      for all $(node, slots) \in nodeRequest$ **do**
        current ← $clusterUsage[node]$
        if $current + slots > MAX\_NODE\_USAGE$ **then**
          exceedsLimit ← True
          break
      if *exceedsLimit* = True **then**
        continue ▷ Sari peste acest job
      lansează asincron job-ul: `execute_job_in_background(job_id, jobDTO)`
      jobStarted ← True

Inițial, bucla începe așteptând `MONITOR_INTERVAL` secunde, iar după acestea pregătește toate datele necesare, făcând cereri către baza de date. Datele obținute constituie cea mai actuală stare a clusterului și a joburilor, deoarece fiecare interacțiune client-joburi-cluster este imediat înregistrată în baza de date, iar datele sunt actualizate continuu. Ulterior algoritmul compară dacă a fost atins vreun maxim dintre limitările setate și, în caz afirmativ, încetează această iterație și continuă cu următoarea, și astfel nu creează niciun job până când toate condițiile sunt satisfăcute.

Ulterior, algoritmul încearcă să instanțieze cel mai vechi job asociat utilizatorului, care încă este în starea `pending`. Joburile păstrează o ordine cronologică în baza de date și în logica de instanțiere, astfel că acestea vor fi luate în ordine din baza de date. Urmează ca datele jobului să fie analizate, mai exact ce noduri specifică configurația acestuia, urmând să fie verificate dacă sunt disponibile. Dacă da, jobul este instanțiat asincron, folosind funcția `execute_job_in_background`.

3.5.1.2. Serviciul de comunicare SSH

Microserviciul care gestionează joburile este singurul care interacționează direct cu clusterul, iar acest lucru se realizează prin comunicarea SSH. În cadrul acestei componente a fost create un "serviciu" care expune o clasă care implementează o serie de funcționalități care implică comunicarea cu clusterul, printre care: conectare (connect), trimiterea unei comenzi SSH execute_command și afișarea rezultatului, trimiterea fișierelor executabile și a fișierelor hostfile, care specifică cum să fie mapate procesele pe noduri, cererea de terminare forțată a unui job și funcții utilitare, de curățare și eliminarea datelor de pe cluster asociate unui job.

Datele unui job sunt salvate într-un loc dedicat în cadrul nodurilor care participă la acel job, iar numele acestora este adnotat cu identificatori unici pentru fiecare job, pentru a evita conflicte dintre fișierele de la ma multe joburi. După execuția unui job, datele acestuia rămân, ca și istoric în cadrul clusterului, atât timp cât utilizatorul păstrează istoric joburilor din cadrul interfeței utilizator.

3.5.1.3. Comunicarea HTTP

Capitolul 4. Testarea aplicației și rezultate experimentale

Capitolul 5. Concluzii

În această lucrare am prezentat proiectarea și implementarea unei platforme web pentru gestionarea și executarea joburilor MPI pe un cluster distribuit. Am pornit de la nevoia de a abstractiza interacțiunea directă cu nodurile de calcul și de a oferi o interfață unificată prin care utilizatorii să poată încărca executabile, configura parametrii de execuție și monitoriza în timp real starea joburilor.

Bibliografie

[1] MPI Forum, "MPI standard documentation," https://www.mpi-forum.org/docs/, 2025, Ultima accesare: 03.06.2025.

[2] R. L. Graham, G. M. Shipman, B. W. Barrett, R. H. Castain, G. Bosilca, and A. Lumsdaine, "Open mpi: A high-performance, heterogeneous mpi," in *2006 IEEE International Conference on Cluster Computing*, 2006, pp. 1–9.

[3] O. Sukhoroslov, "Building web-based services for practical exercises in parallel and distributed computing," *Journal of Parallel and Distributed Computing*, vol. 118, 03 2018.

[4] H. Lin, "Teaching parallel and distributed computing using a cluster computing portal," 05 2013, pp. 1312–1317.

[5] H. Freitas, "Introducing parallel programming to traditional undergraduate courses," 10 2012.

[6] M. Nowicki, M. Marchwiany, M. Szpindler, and P. Bała, "On-line service for teaching parallel programming," vol. 9523, 12 2015, pp. 78–89.

[7] G. Blinowski, A. Ojdowska, and A. Przybyłek, "Monolithic vs. microservice architecture: A performance and scalability evaluation," *IEEE Access*, vol. 10, pp. 20 357–20 374, 2022.

[8] A. Archip, C.-M. Amarandei, P.-C. Herghelegiu, C. Mironeanu, and E. şerban, "Restful web services – a question of standards," in *2018 22nd International Conference on System Theory, Control and Computing (ICSTCC)*, 2018, pp. 677–682.