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

Cuprins

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

Ioan CÎRJĂ

Rezumat

Lucrarea de față prezintă proiectarea și implementarea unei platforme web, cu o arhitectură client-server bazată pe microservicii, destinată gestionării și execuției proceselor OpenMPI pe un cluster distribuit. Am pornit de la nevoia de a abstractiza interacțiunea directă cu nodurile de calcul, iar scopul a fost de a oferi o interfață unificată prin care utilizatorii să poată încărca executabile, configura parametrii de execuție OpenMPI și monitoriza în timp real starea proceselor, într-un mod controlat, rezultatele finale ale execuției fiind disponibile ulterior pentru vizualizare.

Prin acest lucru am dorit să ofer o perspectivă relativ nouă privind utilizarea MPI, cu un nivel de accesibilitate sporit și să realizez o aplicație care partajează resurse computaționale utilizatorilor, într-un mediu web. Deși funcționalitățile din cadrul aplicației nu acoperă în totalitate capacitățile standardului MPI și a librăriilor care îl implementează, totuși aceasta oferă suport în utilizarea rapidă, accesibilă, fără configurări inițiale, a implementării OpenMPI în cadrul unui cluster.

Cuvinte cheie: job MPI, OpenMPI, microservicii, platformă web, cluster

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 procesare secvențială, care se desfășoară pe un singur nucleu din cadrul unei unități de procesare, pot da dovadă de ineficiență, astfel creându-se posibilitatea apariției unor timpi mari de execuție, a unei utilizări ineficiente a resurselor disponibile sau a altor 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 susținut 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 care le-au utilizat 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 programarea 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, instalarea tehnologiilor utilizate pe noduri, introducerea unui sistem de fișiere distribuit, iar exemplele pot continua. De asemenea, este necesară stabilirea mediilor de execuție pentru tehnologiile utilizate, introducerea unui sistem de management al resurselor cu diferite politici de planificare, gestionarea aspectelor ce țin de securitate și access și introducerea de politici de monitorizare și întreținere. 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, de la aplicații desktop, până la soluții bazate pe servicii 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*) [2] 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 ESB (*Enterprise Service Bus*), 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*) [3] a fost definit de către Roy Fielding în 2000, ca fiind un stil arhitectural în construirea aplicațiilor 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, sau 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 o simulare numerică avansată, într-un mod paralelizat și distribuit, care nu ar putea fi posibilă utilizând doar unitatea de calcul a clientului, din cauza problemelor de performanță.

Dezvoltarea soluțiilor de interacțiune, vizualizare și monitorizare a calculului distribuit și paralelizat, precum procesele MPI, nu este frecvent întâlnită. Aceasta implică dezvoltarea și menținerea unor resurse software capabile să gestioneze calculul distribuit, prin comunicarea cu nodurile clusterului și crearea de procese. Cu toate acestea, utilizarea serviciilor web oferă o nouă perspectivă în gestionarea proceselor MPI, deoarece standardul descrie un mod de lucru strâns legat de hardware, și nu un mediu web. 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 multiple 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 OpenMPI în cadrul clusterului, abordări similare ideii prezentate și o serie de concepte privind serviciile web. Capitolul III prezintă în detaliu soluția propusă, din perspectiva arhitecturii, al componentelor și interoperabilitatea dintre acestea, precum și modul de comunicare cu infrastructura clusterului asociat aplicației. Ulterior, în capitolul IV se vor prezenta o serie de rezultate preliminare obținute în utilizarea aplicației. În final, teza prezintă concluzii privind abordarea prezentată, câteva direcții pentru viitoarele etape de dezvoltare și o serie de noțiuni aprofundate.

Capitolul 2. Fundamentarea teoretică și documentarea bibliografică

2.1. Calculul paralel și distribuit

Calculul paralel și calculul distribuit [4] 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 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 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, 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 HPC (*High-Performance Computing*).

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 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 locale.

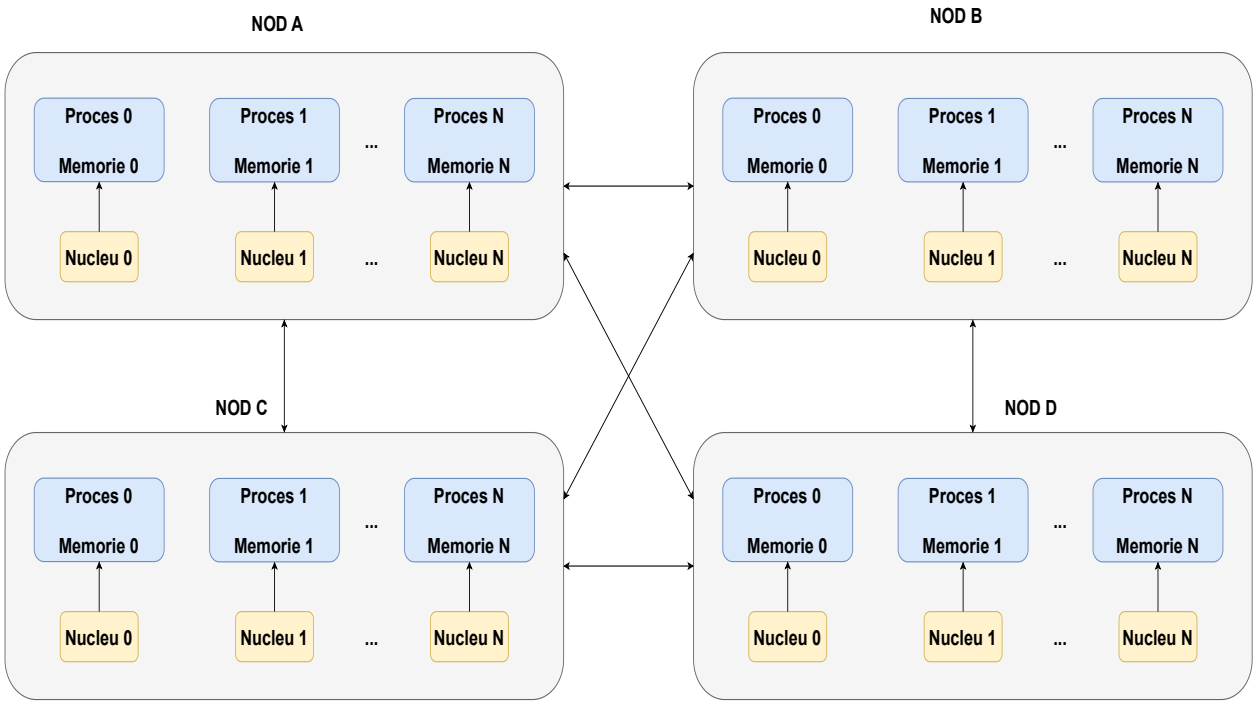

Figura 2.1. Exemplu de procesare paralelă și distribuită

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 [5], 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 [6].

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 implementate 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. În 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. 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 [4], 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 configurație, 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 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 nucleu 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 OpenMPI.
`-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 nucleele 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 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, un subiect care nu este abordat pe scară largă. Totuși, în [7] este descrisă utilizarea serviciilor web în dezvoltarea unei aplicații care să faciliteze accesul studenților la sisteme distribuite și să îi sprijine în învățarea programării paralelizate, 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 [8] prezintă, într-o manieră asemănătoare, utilizarea unei interfețe grafice în predarea calculului distribuit, 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 [9], 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 [10] prezintă 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 oferi suport oricărui client care dorește acces rapid la resurse computaționale, expuse din perspectiva MPI.

Capitolul 3. Soluția propusă

3.1. Specificații privind aplicația propusă

Dezvoltarea unei soluții bazate pe servicii web care utilizează noțiuni REST, accesibile printr-o interfață grafică, care să faciliteze accesul la un cluster OpenMPI, să permită lansarea de joburi în execuție și să poată vizualiza rezultatele în timp real presupune utilizarea unei strategii de implementare capabile să gestioneze o comunicare între utilizator și infrastructura hardware, și care să fie compatibilă cu caracterul asincron al acestei interacțiuni, sau cu alte 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, așa cum este menționat în referințele [7], [9], [8], [10] deși poate veni în ajutorul lor. Aceasta trebuie să interacționeze cu clusterul într-un mod controlat pentru a putea preveni eventualele blocaje. Pentru acest lucru este necesară introducerea de logică de autentificare, autorizare și monitorizare. Aplicația trebuie să conțină o secțiune dedicată pentru autentificare și să restricționeze accesul la resurse dacă un client nu deține drepturile de acces corespunzătoare.

Clusterul OpenMPI necesită o configurare inițială, instalarea tehnologiilor necesare și setarea unei metode de conectare cu resurse externe. Aplicația va comunica doar cu nodul master, urmând ca distribuția resurselor și utilizarea lor să fie realizată prin intermediul acestuia. De asemenea, 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 incapabile să gestioneze sarcina. 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.

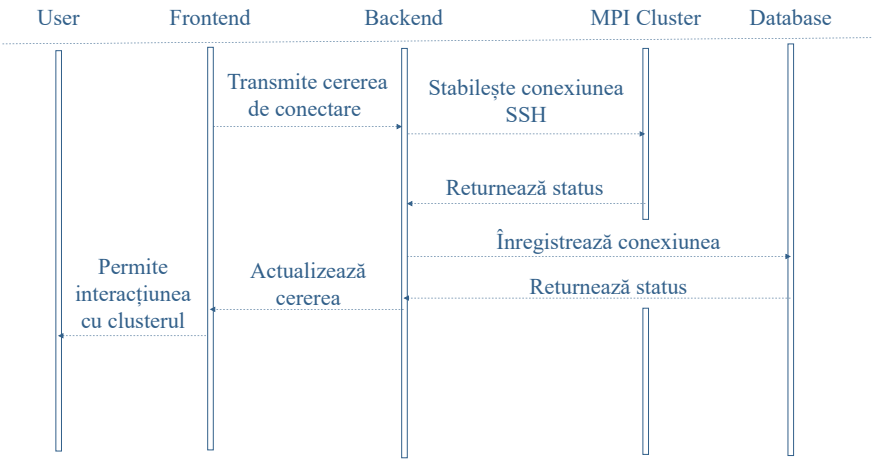

Figura 3.1. Conectarea la cluster prin SSH

Punctul central al aplicației este reprezentat de joburile MPI. În urma autentificării, aplicația trebuie să dispună utilizatorului o pagină dedicată încărcării unui job MPI în cluster, de unde va putea crea o configurație și o va putea trimite în execuție. Configurația minimală a unui job MPI este următoarea:

- Nume și descriere;
- Executabilul propriu-zis, rezultat în urma compilării codului MPI;
- Numărul și distribuția proceselor pe nodurile din cluster (de exemplu: 5 procese pe nodul A, 3 procese pe nodul B, etc.);

- Parametri de configurare, precum: `map-by`, `rank-by`, `oversubscribe`, `bind-to`, `display-map`, variabile de mediu.
- Opțiuni adiționale secundare, precum notificarea atunci când un job este finalizat.

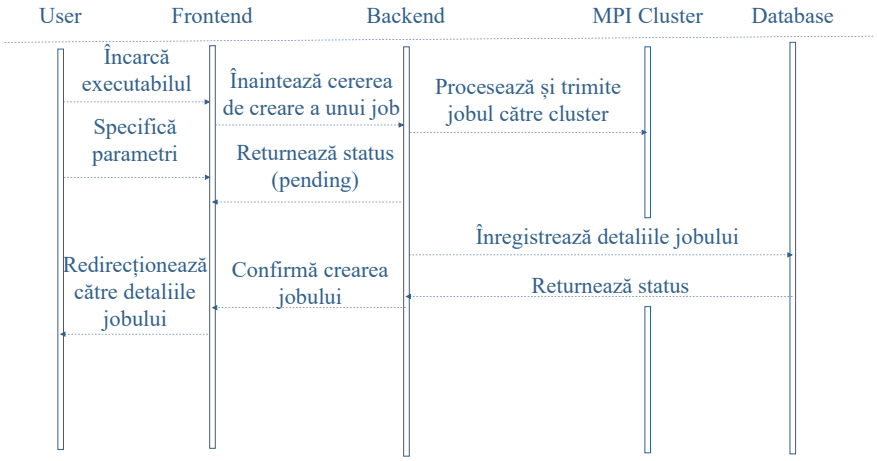

Figura 3.2. Încărcare unui job MPI

Pentru a putea menține evidența joburilor și a facilita vizualizarea rezultatelor trebuie ca fiecare job să fie monitorizat și 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 trimiterea în execuție, în funcție de utilizarea clusterului și de restricțiile accesului la resurse ale unui cont utilizator;
- Running - jobul a fost trimis către cluster și execuția a pornit, așteaptându-se rezultatele;
- Completed - jobul a fost finalizat cu succes, iar 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 aferente acestuia;

Aplicația trebuie să fie capabilă 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, datele aferente unui job trebuie trimise către nodul master din cluster.

Supraîncărcarea clusterului cu un număr prea mare de joburi la un moment dat, provenite de la un singur utilizatori trebuie, de asemenea, oprită, iar acest lucru poate fi realizat prin gestionarea unor permisiuni asociate fiecărui cont din cadrul aplicației, centralizate de către un cont administrator.

Serviciile web trebuie să monitorizeze în mod constant clusterul pentru a deduce disponibilitatea acestuia de a prelua joburile unui utilizator specific. Astfel, definim următoarele metrici:

- Numărul maxim de procese simultane permis pentru un utilizator;
- Numărul maxim de procese per nod, per utilizator;
- Numărul maxim de joburi care pot fi în execuție simultan pentru un utilizator;
- Numărul maxim de joburi aflate în așteptare 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;

Pentru monitorizarea per ansamblu, ținând cont de utilizarea resurselor de către toți utilizatorii în același timp, se definesc următoarele metrici:

- 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. Serviciile web trebuie să interogheaze continuu 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 trebuie să verifice simultan condițiile menționate mai sus, iar dacă acestea sunt îndeplinite, jobul va putea fi executat.

Interfața trebuie să fie reactivă și să permită actualizarea în timp real a statusului fiecărui job, iar acest lucru nu poate fi realizat doar servicii care conțin noțiuni REST clasice. În timp ce rutele REST acoperă operațiunile de tip CRUD (creare, citire, actualizare, ștergere) pentru joburi, 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 utilizarea stilului arhitectural REST, este necesară integrarea unui canal de comunicare asincron și bidirecțional 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 eroare în procesarea jobului, notificarea ajunge în timp real la client, iar statusul este actualizat.

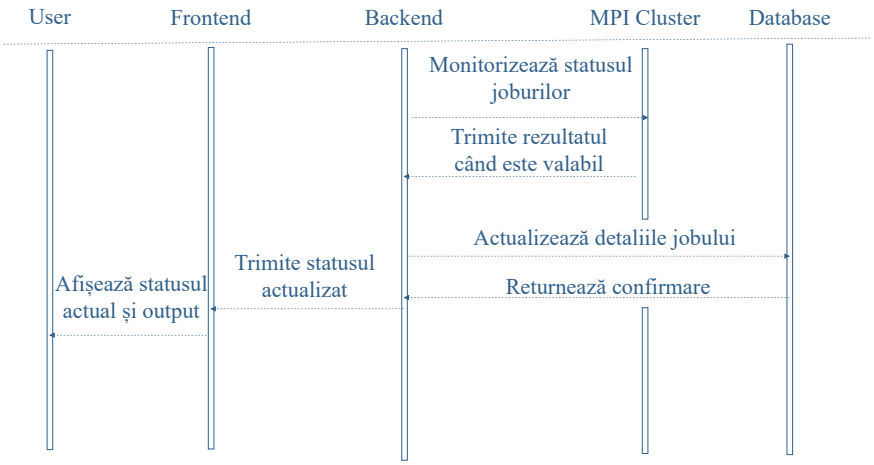

Figura 3.3. Verificarea statusului joburilor MPI

3.2. Arhitectura aplicației și tehnologiile utilizate

Arhitectura aplicației urmează un model client-server, bazat pe microservicii [2] [11]. Implementarea acestora urmărește o reprezentare bazată pe o serie de noțiuni ale stilului arhitectural REST, propus de Fielding, folosind comunicări uniforme prin HTTP(S) (GET, POST, PUT, DELETE) și reprezentări standard ale resurselor, precum JSON. Solicitările clientului sunt fără stare și conțin 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. Toate solicitările provenind din aplicația grafică trec mai întâi printr-un punct central de rutare, care validează cererile și le direcționează către microserviciul potrivit.

Pe lângă comunicarea prin protocolul standard HTTP(S), se utilizează un canal asincron pentru actualizări în timp real. Astfel, pentru afișarea progresului joburilor MPI și raportarea în timp real, este deschis un canal WebSocket între interfață și microservicii, asigurând astfel că starea sau erorile joburilor apar imediat în interfață fără reîncărcări suplimentare.

Pentru interfața utilizator, s-a 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 microserviciile prin apeluri REST și folosește WebSocket-uri pentru notificări în timp real. Autorizarea utilizatorilor se face pe baza tokenurilor JWT (*JSON Web Tokens*).

La nivel de server, responsabilitățile sunt distribuite între cinci microservicii separate, fiecare construit cu FastAPI, utilizând limbajul Python. Fiecare serviciu se ocupă de o funcționalitate specifică, și anume: autentificare și gestiunea permisiunilor asupra clusterului, orchestrarea joburilor MPI și monitorizare, monitorizarea stării fizice a clusterului, actualizarea interfeței cu starea fizică a clusterului, și agregarea celorlalte microservcii.

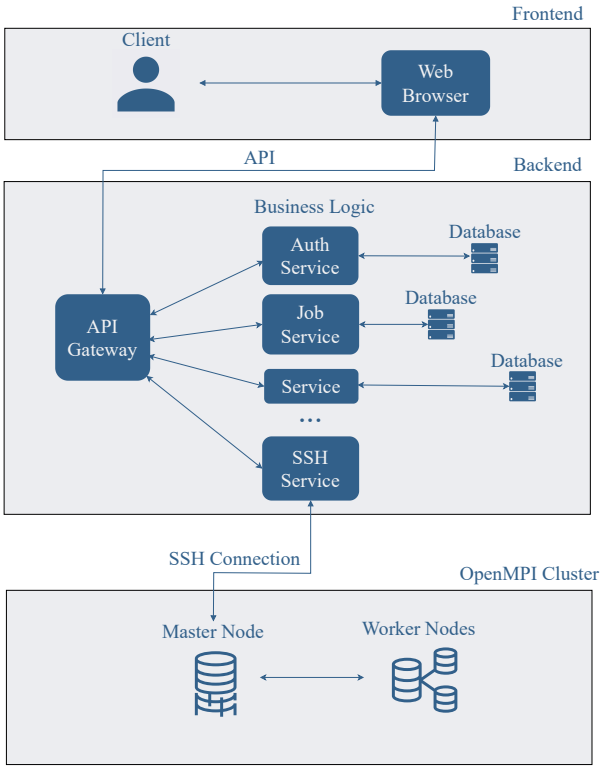

Figura 3.4. Arhitectura generală a aplicației

Conexiunile cu infrastructura clusterului de calcul folosesc SSH și permit lansarea comenzilor necesare fără ca utilizatorii să interacționeze direct cu linia de comandă. Pentru acest lucru s-au utilizat librăriile `paramiko` [12] şi `asyncssh` [13].

Componentele de gestiune a joburilor și de monitorizare a clusterului conțin sarcini care se execută în fundal, în mod paralel și izolat. Acest lucru este necesar pentru a menține reactivitatea acestora la cererile HTTP(S) și a evita comportamentele blocante cauzate de monitorizarea continuă a bazei de date sau a clusterului. Pentru acest lucru s-a utilizat librăria `asyncio` [14], împreună cu funcționalitatea `BackgroundTask` din cadrul FastAPI.

Pentru persistența datelor s-a folosit o soluție non-relațională. MongoDB este utilizat pentru procesarea și stocarea datelor unui job, monitorizarea încărcarii clusterului la un moment dat, valabilitate nodurilor din punct de vedere hardware, gestiunea conturilor utilizator și a drepturilor acestora asupra clusterului. Utilizarea unui format JSON standardizat ca modalitate de persistență a permis adăugarea unor câmpuri noi pe parcurs fără operațiuni de migrare. În plus, în acest proiect nu avem nevoie de interogări complexe, așa că nu beneficiem la maxim de capacitățile relaționale, motiv pentru care MongoDB constituie o opțiune validă pentru persistență.

Aplicația client–server trimite toate solicitările de lansare și monitorizare a joburilor MPI către un singur nod master, numit *C00*. Din moment ce utilizatorul configurează un job prin interfața web, microserviciul de gestiune a joburilor deschide o conexiune SSH către *C00* pentru a executa comanda `mpirun`. Astfel, comenzile de lansare a joburilor 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 microservicii, stabilite după cum urmează: înainte de lansarea jobului, se calculează un subinterval de porturi dintr-un interval predefinit și se transmit acele valori în parametrii OpenMPI. 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 OpenMPI și orted gestionează operațiunea de distribuție și comunicare între noduri.

3.3. Dezvoltarea microserviciilor

Fiecare microserviciu urmează o structură stratificată, după cum urmează: nivelul `Controller` expune rutele REST, sau canale WebSocket și validează, autorizează, folosind token-ul JWT, sau redirecționează cererile HTTP(S); nivelul `Service` conține logica de afaceri și gestionează sarcinile asincrone, iar nivelul `Repository` izolează și implementează accesul la date. Datele sunt transferate între nivele folosind structuri DTO (*Data Transfer Object*).

3.3.1. 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 un mecanism de monitorizarea continuă a statusului joburilor la nivelul bazei de date și al clusterului și implementează un algoritm de comparație care decide dacă un job poate fi instanțiat sau nu pentru a preveni supraalocarea și a facilita un acces controlat la nivel de cluster.

Încărcarea unui job are loc atunci când se primește o cerere HTTP de tip POST către URI-ul `api/jobs/upload-job` care conține, în format JSON, configurația acestuia, alături de executabilul MPI și fișierul `hostfile`, care conține distribuția proceselor pe noduri, codificate în `Base64`:

```
{
    "_id": "string",
    "jobName": "string",
    "jobDescription": "string",
    "beginDate": "string",
    "endDate": "string",
    "fileName": "string",
    "fileContent": "string",
```

```
    "hostFile": "string",
    "hostNumber": "integer",
    "numProcesses": "integer",
    "allowOverSubscription": "boolean",
    "environmentVars": "string",
    "displayMap": "boolean",
    "rankBy": "string",
    "mapBy": "string",
    "status": "string",
    "output": "string",
    "alertOnFinish": "boolean",
    "user_id": "string",
    "userEmail": "string"
}
```

Listing 3.1. Datele unui job trimise prin HTTP(S)

În urma cererii POST, microserviciul realizează intern, o cerere GET către microserviciul care gestionează conturile clienților, la `/api/users/quotas`, pentru a obține limitările utilizatorului respectiv privind utilizarea clusterului. Ulterior se verifică care este starea joburilor utilizatorului respectiv, izolat de restul clienților, iar apoi se compară datele pentru a decide dacă se poate instanția acel job.

```
if job_data.numProcesses > int(quota['max_processes_per_user']):
    raise HTTPException(status_code=status.HTTP_403_FORBIDDEN,
     ↪  detail="Exceeded max processes per user!")

if len(running_jobs_for_user) > int(quota['max_running_jobs']):
    raise HTTPException(status_code=status.HTTP_403_FORBIDDEN,
     ↪  detail="Max running jobs limit reached!")

if len(node_request) > int(quota['max_nodes_per_job']):
    raise HTTPException(status_code=status.HTTP_403_FORBIDDEN,
     ↪  detail="Exceeded max nodes per job!")

for node, slots in node_request.items():
    if slots > int(quota['max_processes_per_node_per_user']):
        raise HTTPException(status_code=status.HTTP_403_FORBIDDEN,
         ↪  detail=f"{node} exceeds the max process count per node!")

    if node not in allowed_nodes_with_prefix:
        raise HTTPException(status_code=status.HTTP_403_FORBIDDEN,
         ↪  detail=f"Node {node} is not allowed!")
```

Listing 3.2. Exemplu de verificare a cotelor utilizatorului

Exemplul prezentat anterior reprezintă o secțiune din logica din `Controller`. Pe baza corpului cererii POST, se reconstruiește fișierul `hostfile` și se verifică dacă se depășește cererea de procese pentru un nod sau se utilizează noduri care nu sunt permise acestui utilizator. În caz afirmativ, va fi returnat un cod de eroare, iar jobul nu va fi creat.

3.3.1.1. Monitorizare continuă a joburilor

În momentul în care microserviciul destinat încărcării joburilor în cluster și vizualizării rezultatelor este pornit, acesta inițializează o sarcină secundară, care rulează în paralel, folosind `create_task`

din librăria `asyncio`. Scopul acesteia este de a decide dacă, la un moment dat, clusterul este disponibil să preia execuția unui job marcat ca `pending` în baza de date. Practic, acestă sarcină 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 continuu, la un interval de monitorizare numit `MONITOR_INTERVAL`, iar în cadrul fiecărui ciclu, încearcă să pornească execuția unui singur job MPI aflat în starea `pending` în cluster. 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`

```
1:  procedure SCHEDULEPENDINGJOBS
2:      MAX_RUNNING ← limită joburi în execuție
3:      MAX_NODE_USAGE ← limită procese per nod
4:      MAX_PENDING ← limită joburi în așteptare
5:      MAX_TOTAL_USAGE ← limită utilizare totală cluster
6:      loop
7:          așteaptă MONITOR_INTERVAL secunde
8:          pendingJobs ← obține toate joburile cu status pending
9:          runningJobs ← obține toate joburile cu status running
10:         clusterUsage ← calculează utilizarea procese pe fiecare nod
11:         totalUsage ← |runningJobs| + |pendingJobs|
12:         if totalUsage ≥ MAX_TOTAL_USAGE then
13:             continue                          ▷ S-a atins utilizarea totală maximă
14:         if |runningJobs| ≥ MAX_RUNNING then
15:             continue                  ▷ S-a atins numărul maxim de joburi running
16:         if |pendingJobs| ≥ MAX_PENDING then
17:             continue                  ▷ S-a atins numărul maxim de joburi pending
18:         jobStarted ← False
19:         for all job ∈ pendingJobs do
20:             if jobStarted = True then
21:                 break                      ▷ S-a lansat deja un job în această iterație
22:             decodează hostfile din job → construire dicționar nodeRequest de forma {nod: sloturi}
23:             exceedsLimit ← False
24:             for all (node, slots) ∈ nodeRequest do
25:                 current ← clusterUsage[node]
26:                 if current + slots > MAX_NODE_USAGE then
27:                     exceedsLimit ← True
28:                     break
29:             if exceedsLimit = True then
30:                 continue                                  ▷ Sari peste acest job
31:             execute_job_in_background(job_id, jobDTO)     ▷ Lansează asincron jobul
32:             jobStarted ← True
```

Inițial, bucla începe așteptând `MONITOR_INTERVAL` secunde, iar după acestea pregătește toate datele necesare, făcând cereri către baza de date. Fiecare interacțiune client-joburi-cluster din cadrul aplicației este imediat înregistrată în baza de date, acestea fiind actualizate continuu. Astfel, se obține cea mai actuală stare a clusterului și a joburilor. Ulterior algoritmul compară dacă a fost atins vreun maxim dintre limitările setate, iar în caz afirmativ, încetează această iterație și continuă cu următoarea,

și astfel nu creează niciun job până când toate condițiile sunt satisfăcute.

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. În caz afirmativ, jobul este instanțiat asincron, folosind corutina `execute_job_in_background`, prezentată mai jos.

3.3.1.2. Trimiterea unui job către cluster

Corutina `execute_job_in_background` din interiorul clasei `JobService` este cea care realizeaza agregarea datelor trimise de utilizator și crearea unui job efectiv la nivel de cluster. De îndată ce funcția este apelată, jobul va fi marcat cu starea `running`, iar interfața va fi notificată pentru a actualiza vizual statusul. Urmează să se stabilească o conexiune SSH către nodul master, să se definească un set de porturi TCP pentru comunicarea între noduri dintr-un set de porturi gata predefinit, în mod dinamic, și să se defineacă locațiile fișierelor la nivelul nodurilor. Concret, se creează o cale unică identificată print-un id, de forma `/home/mpilauncher/jobs/job_uuid4` unde vor fi salvate toate fișierele utilizate de un job. Urmează validarea parametrilor trimiși de către utilizator prin HTTP, către `/api/jobs` și formatarea lor în comanda propriu-zisă `mpirun` care va fi lansată în cadrul clusterului de către nodul master. Fișierul executabil și fișierul hostfile, care sunt utilizate în comandă, sunt decodate din `Base64` în fișiere temporare care vor fi trimise către cluster. Fiecare fișier este redenumit astfel încât să includă identificatorul jobului înainte de a fi trimis prin SSH. Acest lucru se realizează pentru organizarea eficientă a fișierelor în cadrul clusterului și evitarea conflictelor provenite de la fișierelor cu același nume sau a căilor inexistente.

După ce fișierele sunt trimise și salvate către cluster, începe asamblarea comenzii `mpirun` și executarea sa. Se folosesc funcționalitățile `asyncio.wait_for` și `asyncio.to_thread` pentru a asigura o execuție non-blocantă, cu un timeout specific fiecărui user, în funcție de permisiunile acestuia. Dacă jobul nu este finalizat în timp util, acesta este marcat cu starea `failed`, iar rezultatele nu vor fi disponibile, chiar dacă jobul nu conține erori. Dacă, în schimb, rezultatul este obținut în timp util, datele sunt propagate către interfață și puse la dispoziția clienților folosind WebSockets.

```
1   async def execute_job_in_background(self, job_id: str, job_data:
     ↪  JobUploadDTO, timeout: int):
2   # ... -> se definesc un set de inițializări
3       try:
4           success = self.repository.update_job(job_id, updated_data)
5           if success:
6               self.notify_frontend(job_id, "running", "", "")
7
8               ssh_service = SSHService(SSH_HOST, SSH_PORT, SSH_USERNAME,
                 ↪  SSH_PASSWORD)
9               ssh_service.connect()
10              port_range = self.allocate_port_range(job_id)
11
12  # ... -> se validează și se pregătesc date pentru cluster
13              scan_result = self.scan_file_with_virustotal(temp_exe_path)
14              if scan_result != 0:
15                  raise Exception("File flagged by VirusTotal")
16
17              await ssh_service.send_file_to_hosts(job_id, temp_exe_path,
                 ↪  temp_hostfile_path)
18              mpirun_cmd = (
19                  f"mpirun {env_vars_str or ''} "
```

```
                f"-hostfile {remote_path_host} "
                f"-np {job_data.numProcesses} "
                f"--mca oob_tcp_dynamic_ipv4_ports {port_range} "
                f"--report-pid {remote_pid_path} "
                f"--output-filename {output_path}"
            )
            if job_data.mapBy:
                mpirun_cmd += f" --map-by {job_data.mapBy}"
            if job_data.rankBy:
                mpirun_cmd += f" --rank-by {job_data.rankBy}"
            if job_data.displayMap:
                mpirun_cmd += " --display-map"
            if job_data.allowOverSubscription:
                mpirun_cmd += " --oversubscribe"
            mpirun_cmd += f" {remote_path_exe}"

            try:
                output = await asyncio.wait_for(
                    asyncio.to_thread(ssh_service.execute_command,
                                      mpirun_cmd,
                                      remote_path_exe,
                                      remote_path_host),
                    timeout=timeout
                )

# ... -> se stabilește rezultatul și statusul
            self.update_job_status_and_output(job_id, job_data_db)
            self.notify_frontend(job_id, job_data_db.status,
             ↪  job_data_db.output, job_data_db.endDate)
            if job_data.alertOnFinish:
                self.send_job_status_email(job_id, job_data_db)
```

Listing 3.3. Secvențe din implementarea metodei `execute_job_in_background`

La final, se realizează o operație de curățare, unde fișierele temporare create sunt eliminate și porturile rezervate jobului sunt eliberate.

3.3.1.3. 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 creată clasa `SSHService` care implementează o serie de funcționalități care implică comunicarea cu clusterul, printre care: conectare, trimiterea unei comenzi SSH 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.

Mai jos este prezentată funcția `execute_command`, care este utilizată pentru a trimite o comandă `mpirun` către cluster. Funcția primește ca parametri locațiile de pe cluster unde se află fișierul executabil și fișierul `hostfile` asociat unui job și comanda `mpirun`. Pentru execuția propriu-zisă a comenzii se utilizează `self.client.exec_command`, unde `client` este o instață a `SSHClient` a librăriei `paramiko`.

```
def execute_command(self, command: str, remote_path_exe: str,
 ↪  remote_path_host:str):
```

```
    try:

        chmod_command = f"chmod +x {remote_path_exe}"
        stdin, stdout, stderr = self.client.exec_command(chmod_command)

        chmod_command = f"chmod +x {remote_path_host}"
        stdin, stdout, stderr = self.client.exec_command(chmod_command)

        stdin, stdout, stderr = self.client.exec_command(command)
        command_output = stdout.read().decode('utf-8')
        command_error = stderr.read().decode('utf-8')

        if command_error:
            raise Exception(f"Error executing command:
             ↪ {command_error}")
        return command_output

    except Exception as e:
        raise Exception(f"Error executing command: {str(e)}")
```

Listing 3.4. Trimiterea unei comenzi către cluster

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

Pentru a trimite fișierul executabil către joburi se folosesc corutinele `send_file_to_host` și `send_file_to_hosts`. Prima corutină menționată are rolul de a trimite date către un singur nod din cadrul clusterului utilizând `asyncssh.connect`. Aceasta creează o conexiune cu nodul respectiv și execută comenzile bash necesare pentru a crea directoarele și a transmite fișierele unui job. Ulterior, corutina `send_file_to_hosts` parsează fișierul `hostfile`, se asigură că nodul master este enumerat, întrucât acesta trebuie să conțină mereu fișierele joburilor, și crează câte o sarcină paralelă de trimitere de date către fiecare nod utilizat, pe care le așteaptă, folosind `asyncio.gather`.

```
async def send_file_to_host(self, fqdn, job_id, local_exe,
 ↪ local_hostfile):
    remote_job_dir = f"{BASE_DIRECTORY}/job_{job_id}"
    try:
        async with asyncssh.connect(fqdn, port=self.port,
         ↪ username=self.username, password=self.password,
         ↪ known_hosts=None) as conn:

            await conn.run(f"mkdir -p {remote_job_dir}", check=True)

            remote_exe_path =
             ↪ f"{remote_job_dir}/{os.path.basename(local_exe)}"
            await asyncssh.scp(local_exe, (conn, remote_exe_path))

            remote_hostfile_path =
             ↪ f"{remote_job_dir}/hostfile_{job_id}.txt"
            await asyncssh.scp(local_hostfile, (conn,
             ↪ remote_hostfile_path))

```

```
14              await conn.run(f"chmod +x {remote_exe_path}", check=True)
15
16      except Exception as e:
17          raise Exception(f"Error sending files to {fqdn}: {str(e)}")
18
19  async def send_file_to_hosts(self, job_id, local_exe, local_hostfile):
20      try:
21          with open(local_hostfile, 'r') as hostfile:
22              hosts = [line.strip().split()[0] for line in hostfile if
                 ↪  line.strip()]
23
24          if not hosts:
25              raise ValueError("Hostfile is empty or invalid!")
26
27          if master_node not in hosts:
28              hosts.insert(0, master_node)
29
30          tasks = []
31          for host in hosts:
32              fqdn = f"{host.lower()}.cs.tuiasi.ro"
33              tasks.append(self.send_file_to_host(fqdn, job_id,
                 ↪  local_exe, local_hostfile))
34
35          await asyncio.gather(*tasks)
36
37      except Exception as e:
38          raise Exception(f"Failed to send files to hosts: {str(e)}")
```

Listing 3.5. Trimiterea fișierelor către cluster

3.3.2. Componenta de gestiunea a conturilor utilizator

Acest microserviciu gestionează acțiunile centrate pe utilizator, precum înregistrarea, autentificarea, stabilirea rolului utilizatorului în cadrul aplicației și permisiunile la nivel de cluster. La primirea unei cereri POST către URI-ul `/api/signup` se creează un cont utilizator, cu rolul `base`, ceea ce înseamnă că va putea instanția și vizualiza joburi, însă nu are dreptul de a-și modifica permisiunile la nivel de cluster. În cadrul bazei de date se stochează datele completate de utilizator atunci când a creat contul, la care se adaugă automat câmpurile care constituie drepturile sale de a crea joburi MPI. Parola este criptată utilizând `hashlib.sha256`, ceea ce face necesară stocarea unui câmp adițional, `salt`, pentru decriptarea parolei. De asemenea se stochează subdocumentul `suspensions`, care este utilizat pentru a decide dacă un utilizator are drept de logare sau nu. Subdocumentul conține dată de început a suspendării și timpul de suspendare în minute.

```
1   {
2     "_id": "ObjectId",
3     "username": "string",
4     "email": "string",
5     "password": "string",
6     "salt": "string",
7     "max_processes_per_user": "number",
8     "max_processes_per_node_per_user": "number",
9     "max_running_jobs": "number",
10    "max_pending_jobs": "number",
```

```
    "max_job_time": "number",
    "allowed_nodes": "string",
    "max_nodes_per_job": "number",
    "max_total_jobs": "number",
    "suspensions": [
      {
        "id": "string",
        "user_id": "ObjectId",
        "username": "string",
        "email": "string",
        "start_date": "string",
        "suspend_time": "number"
      }
    ],
    "rights": "string"
  }
```

Listing 3.6. Datele aferente unui utilizator

Autentificarea se realizează accesând o cerere POST către `/api/login`, iar ca răspuns se returnează token-ul JWT, utilizat pentru autentificarea cererilor HTTP ulterioare.

Microserviciul gestionează, de asemenea, operațiile efectuate de administrator asupra conturilor utilizator privind limitările acestora în utilizarea aplicației. Utilizând endpoint-uri prefixate cu `/api/admin/`, administratorul realizează operațiuni de tip CRUD asupra câmpurilor stocate în baza de date aferente unui cont de utilizator, menționate mai sus.

3.3.3. Componenta de monitorizare a stării clusterului

Componenta de monitorizare a stării generale a clusterului este separată de logica gestionării stării joburilor și este constituită dintr-un microserviciu care operează intern, fără a expune un `Controller` cu rute ce pot fi accesate de entități externe.

În momentul pornirii acestuia, folosind decoratorul `@app.on_event("startup")`, se utilizează `asyncio.create_task`, pentru a instanția sarcina `run_ssh_cycle_and_notify`. Aceasta este o corutină responsabilă de gestionarea unui ciclu continuu de monitorizare a stării nodurilor clusterului prin intermendiul unei conexiuni SSH. La începutul execuției, se obține bucla de evenimente `asyncio`, care permite coordonarea operațiilor asincrone. Apoi se creează o instanță a clasei `MonitorService`, care conține logica de conectare și actualizare a stării nodurilor în baza de date. Metoda de conectare prin SSH este realizată folosind librăria `paramiko` și este blocantă. Din această cauză, a fost delegată către un executor separat, folosind `loop.run_in_executor` pentru a preveni blocajele.

Ulterior, se creează o buclă infinită care apelează metoda `update_node_statuses_in_db`. Aceasta trimite comanda prezentată mai jos către fiecare nod și, iar dacă primește un răspuns înapoi de la nod, marchează nodul ca fiind activ. Procesul se realizează pentru fiecare nod, urmând ca statusurile să fie agregate și actualizate în baza de date.

În final, blocul `finally` se încheie conexiunea SSH, tot cu ajutorul unui executor, pentru a preveni blocarea buclei de evenimente în timpul închiderii.

```
ssh -o ConnectTimeout=5 -o StrictHostKeyChecking=no mpi.cluster@{node} uptime
```

```
async def run_ssh_cycle_and_notify():
    loop = asyncio.get_event_loop()
    ssh_service = MonitorService(HOST, PORT, USERNAME, PASSWORD)
```

```
4
5       await loop.run_in_executor(None, ssh_service.connect)
6
7       try:
8           while True:
9               ssh_service.update_node_statuses_in_db()
10              await asyncio.sleep(MONITOR_INTERVAL)
11      finally:
12          await loop.run_in_executor(None, ssh_service.close)
```

Listing 3.7. Implementarea metodei `run_ssh_cycle_and_notify`

```
1   def update_node_statuses_in_db(self):
2       node_status_dtos = []
3       for node in MPI_HOSTS:
4           status = self.run_ssh_command(node)
5
6           node_key = node.split('.')[0]
7           formatted_node_key = node_key.upper()
8
9           node_status_dtos.append(NodeStatusDTO(formatted_node_key,
            ↪  status))
10
11      self.repo.update_all_node_statuses(node_status_dtos)
```

Listing 3.8. Implementarea metodei `update_node_statuses_in_db`

3.3.4. Componenta de broadcast a datelor privind clusterul

Componenta de broadcast este responsabilă cu transmiterea periodică a stării fizice a nodurilor clusterului către interfața utilizator, folosind o conexiune WebSocket. Similar cu microserviciul de monitorizare a clusterului, funcționarea este asincronă, pe baza unei sarcini care rulează în fundal, și care este pornită la inițializarea aplicației FastAPI. Datele provenite de la această componentă sunt utilizate la nivelul interfeței utilizator pentru a preveni joburi către noduri care nu sunt disponibile la un moment dat din punct de vedere fizic. Astfel, microserviciul realizează o monitorizare a clusterului diferită ca logică comparativ cu monitorizarea joburilor.

La inițializarea aplicației, în funcția marcată cu `@app.on_event("startup")` se pornește o sarcină asincronă care execută corutina `broadcast_node_statuses`. Metoda utilizează clasa BroadCastRepository pentru a extrage periodic datele actuale privind starea nodurilor și a le trimite, periodic, către conexiunea WebSocket, accesibilă folosind `@router.websocket("/ws")`.

Microserviciile de montorizare la nivel de cluster și de broadcast utilizează o bază de date comună pentru a evita datele duplicate, dar se mențin ca microservicii independente pentru a separa logica de comunicare continuă folosind websocket-uri de conexiunile asincrone realizate prin SSH folosind executori pentru interogarea clusterului și a asigura simplitate în implementare.

```
1   async def broadcast_node_statuses():
2       repo = BroadCastRepository()
3       while True:
4           try:
5               node_statuses = repo.get_all_node_statuses()
6               if not node_statuses:
7                   await asyncio.sleep(10)
```

```
                continue
            node_status_data = json.dumps(node_statuses)
            for websocket in status_active_connections:
                try:
                    await websocket.send_text(node_status_data)
                except WebSocketDisconnect:
                    status_active_connections.remove(websocket)
            await asyncio.sleep(MONITOR_INTERVAL)
        except Exception as e:
            logging.error(f"Error while broadcasting: {e}")
            await asyncio.sleep(10)
```

Listing 3.9. Implementarea metodei `broadcast_node_statuses`

3.3.5. Componenta de agregare a microserviciilor

Componenta de agregare a microserviciilor este creată cu scopul de a facilita accesul centralizat la nivelul microserviciilor. Interfața utilizator trimite cererile HTTP(S) doar către această componentă, iar aceasta redirecționează cererea către microserviciul dedicat gestionării acelei sarcini, în funcție de ruta folosită. Redirecționarea unei cereri este realizată folosind corutina `forward_request`. Aceasta preia cererile realizate prin intermediul interfeței web și folosește `httpx.AsyncClient` pentru a trimite o cerere către microserviciul care expune ruta din `target_url`.

```
async def forward_request(request: Request, target_url: str):
    async with httpx.AsyncClient() as client:
        body = await request.body()
        headers = dict(request.headers)
        method = request.method
        try:
            response = await client.request(
                method, target_url, content=body, headers=headers
            )

            return JSONResponse(content=response.json(),
             ↪  status_code=response.status_code)
        except httpx.RequestError as e:
            raise HTTPException(status_code=500, detail=f"Error
             ↪  communicating with service: {str(e)}")
```

Listing 3.10. Implementarea metodei `forward_request`

3.4. Dezvoltarea interfeței utilizator

Interfața utilizator este constituită din componente Angular care facilitează interacțiunea cu clusterul OpenMPI, prin microservicii, într-un mod interactiv. Accesând paginile web, utilizatorul consumă rutele REST expuse de microservicii pentru operații CRUD și dechide canale de comunicație WebSocket pentru actualizări în timp real ale statusurilor joburilor.

Stările interne ale interfeței sunt actualizate folosind mecanisme specifice Angular, precum observabile `RxJs` pentru a facilita reactivitatea aplicației la datele provenite de la microservicii. Cererile HTTP(S) includ un antet de autorizare constituit din token-ul JWT stocat în local storage. Acesta este adăgat automat folosind un interceptor Angular. Navigarea pe paginile web este controlată folosind Angular Guards, componente care interceptează navigarea pe o anumită pagină și o opresc în cazul accesului neautorizat.

3.4.1. Paginile de autentificare

Intrarea în sistem se realizează accesând pagina de logare sau cea de creare a unui cont, disponibile la `/api/login` și `/api/signup`. Acestea expun, pentru utilizatori, formulare prin care se pot conecta la un cont creat anterior, pentru a continua crearea joburilor, sau pentru a crea un cont nou. În urma autentificării, se va crea și stoca un token JWT, utilizat pentru a accesa restul aplicației.

3.4.2. Pagina de încărcare a unui job

După ce autentificarea a fost efectuată cu succes, utilizatorul este redirecționat către pagina de încărcare a unui job, accesibilă la `/api/upload-jobs` unde are acccesul la un formular cu ajutorul căruia va crea un job. Pagina expune opțiunile `mpirun`, distribuția proceselor pe noduri, opțiunea pentru încărcarea unui executabil și butoane de încărcare sau salvare a unei configurații. Utilizatorul poate plasa cursorul peste setări pentru a primi mai multe informații despre acestea. După ce datele au fost validate, butonul "Create Job" devine disponibil, iar utilizatorul poate crea jobul MPI.

3.4.3. Pagina de vizualizare a rezultatelor

După ce formularul de încărcare a unui job este completat și trimis, interfața grafică afișează o animație de încărcare, apoi redirecționează utilizatorul către pagina de vizualizare a rezultatelor, disponibilă la `/api/status`. Aici este afișat istoricul joburilor asociate acelui utilizator și statusul lor. În funcție de datele provenite de la WebSocket, statusul joburilor este actualizat. Practic, gestiunea joburilor și actulizarea interfeței simulează comportamentul unei cozi de mesaje, unde fiecare job este procesat doar în anumite condiții.

Asupra unui job se pot efectua un set de acțiuni la nivel de interfață, printre care enumerăm: încetarea forțată a execuției unui job, salvarea locală a configurației unui job, eliminarea acestuia din istoric și din cluster. Vizualizarea rezultatelor poate fi realizată apăsând butonul `expand`.

3.4.4. Pagina dashboard

Pentru a vizualiza metrici agregate despre joburile sale, sau limitările contului său, utilizatorul poate naviga la pagina dashboard, accesibilă la `/api/dashboard`. Pagina realizează cereri HTTP(S) similare cu pagina care afișează statusul joburilor, însă efectuează o serie de agregări ale datelor pentru a crea și dispune metrici informative pentru utilizator. Scopul acestei pagini este de a prezenta limitările contului utilizator și a îmbunătăți experiența de utilizare în cadrul aplicației.

3.4.5. Pagina administrator

Pagina administrator este disponibilă doar prin introducerea credențialelor contului administrator în momentul logării. Scopul acestea este de a vizualiza și modifica cotele unui utilizator sau de a crea suspendări și a bloca accesul la aplicație. Pagina expune tabele ce conțin date despre utilizatori și joburi MPI, asupra cărora se pot efectua modificări.

Capitolul 4. Testarea aplicației și rezultate experimentale

4.1. Lansarea aplicației și elemente de configurare

Înainte ca aplicația să fie pornită, clusterul trebuie să poată fi contactat. Pentru acest lucru am setat o conexiune VPN pe mașina gazdă, astfel încât tot traficul SSH să poate fi transmis către rețeaua internă a clusterului. Microserviciul care gestionează joburile și microserviciul care monitorizează dacă nodurile sunt active sunt singurele care interacționează cu clusterul. Acestea au nevoie de un cont utilizator, pentru acces SSH, în mod special către nodul master pentru a porni un job folosind `mpirun`, dar și către nodurile secundare, pentru a transmite executabilul.

Parametri de configurare ai aplicației sunt centralizați în fișiere .env la nivel de microserviciu. Aici sunt stocate setări precum: hostnames pentru fiecare nod, portul SSH, directorul de bază unde vor fi salvate datele pe noduri, valoare intervalelor de monitorizare și alte credențiale.

Înainte de a porni aplicația web, trebuie să instalăm Python, împreună cu dependețele necesare, specificate în fișierul `requirements.txt` al fiecărui microserviciu, și framework-ul Angular. Lansarea microserviciilor se realizează utilizând utilitarul `uvicorn`, iar interfața utilizator este pornită folosind comanda `ng serve`.

Pentru partea de persistență, trebuie menționat că instanţele MongoDB rulează în containere Docker şi trebuie pornite înainte de lansarea microserviciilor.

Deși configurarea propriu-zisă a clusterului nu face parte din tema lucrării de față, este important de menționat necesitatea acesteia, întrucât aplicația web este dependentă de disponibilitatea clusterului de a executa joburi MPI.

4.2. Testarea sistemului

Modalitatea principală de testare a sistemul este testarea funcțională, concentrându-se pe validarea end-to-end a funcționalităților fiecărui microserviciu. Pentru depanare s-a utilizat Postman, cu scopul de simula cereri HTTP(S) și a verifica răspunsurile rutelor expuse de microservicii.

Pentru testarea modului de gestionare a joburilor multiple, implementat utilizând algoritmul de comparație, vom crea joburi MPI cu durate de execuție diferite, care vor fi instanțiate în același timp. Pentru acest lucru au fost create mai multe executabile MPI care au un timp de execuție controlat și prelugit prin utilizarea funcției `sleep` a librăriei `unistd.h` în cadrul programului MPI. Astfel, se simulează joburi complexe, și se poate testa modul în care joburile trec din starea `pending` în starea `running` și dacă se respectă limitările generale privind utilizarea clusterului din funcția `monitor_pending_jobs`, prezentată anterior.

Pentru testarea cotelor impuse unui cont utilizator, vom trimite în execuție joburi MPI care au configurări complexe, care încalcă limitările impuse acelui utilizator. Rezultatul așteptat este respingerea acelui job și afișarea unui mesaj informativ.

Pentru a simula comportamentul utilizatorilor în cadrul aplicației web trebuie să accesăm paginila de creare a unui cont și apoi să ne autentificăm. Vom testa modulul de autentificare, creând conturi multiple, prin introducerea de credențiale valide și invalide și vom urmări comportamentul interfeței și răspunsul serverului.

La nivelul interfeței utilizator, vom crea un set de joburi MPI, cu distribuții diferite de procese, și executabile diferite, pentru a testa fluxul de încărcare. Utilizăm fișiere prea mari sau cu un format neacceptat și introducem diferite forme de input pentru a verifica validarea acestora. Pentru testarea configurației unui job, selectăm diferite combinații de parametri și distribuții pe noduri. Alegem cazuri în care utilizăm un singur nod, toate nodurile, număr maxim de procese pe un nod, pentru a verifica funcționalitatea permisiunilor utilizatorului la nivel de cluster. Se realizează, de asemenea, teste funcționale privind autorizarea corectă a accesului către paginile web ale aplicației.

După crearea unui număr mare de joburi MPI, navigăm pe pagina de vizualizare a rezultatelor și observăm modul de execuție al acestora. Joburile simple se execută rapid, iar rezultatul este imediat disponibil. În funcție de cotele setate, joburile trec treptat din starea `pending` în starea `running` și sunt trimise către cluster. Fiecare job prezintă un indicator care reprezintă starea jobului la un moment dat, schimbându-se în funcție de aceasta.

Testăm, de asemenea, funcționarea acțiunilor aferente unui job: descărcarea executabilului sau a fișierului hostfile, salvarea configurației, repornirea jobului, terminarea forțată și eliminarea din istoric și din cluster. În urma execuției, vizualizăm rezultatele.

Pentru testarea paginii destinate administratorului, modificăm valorile cotelor unui utilizator și adăugăm suspendări, apoi verificăm impactul asupra contului utilizator. Acesta își poate verifica cotele în cadrul paginii dashboard.

4.3. Aspecte legate de securitate și scalabilitate

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. Astfel, executabilele utilizate în cadrul aplicației web pot conține elemente malware, care pot afecta activitatea clusterului. Acest lucru este gestionat prin trimiterea fișierelor executabil încărcate de utilizatori către un API extern, numit VirusTotal, urmând ca doar executabilele aprobate să fie transferate pe nodul master și lansate prin SSH.

Modul de autentificare SSH utilizat în implementarea microserviciilor funcționează într-o manieră neprivilegiată, fără drepturi de administrator, iar permisiunile de utilizare a clusterului sunt limitate. De asemenea, serverul validează comenzile trimise prin SSH pentru a preveni potențiale comportamente nedorite.

După cum a fost prezentat anterior, autentificarea și autorizarea utilizatorilor la nivelul aplicației web se realizează prin token-uri JWT, facilitate de microserviciul de autentificare în urma validării credențialelor unui utilizator. Token-ul conține câmpuri standard (sub, iat, exp, iss), precum și câmpuri ce stabilesc rolul și cotele unui utilizator.

Din perspectiva gestionării cotelor utilizatorilor la nivelul aplicației web, se utilizează un cont administrator, care are privilegii speciale în privind utilizarea acesteia, comparativ cu utilizatorii normali. Introducerea acestuia a fost realizată cu scopul de a crește nivelul de control asupra utilizatorilor în cadrul aplicației. Utilizarea clusterului este dependentă de setările pe care administratorul le stabilește, deoarece acestea sunt luate în considerare în cadrul microserviciului de gestiune a joburilor. În plus, administratorul poate restricționa accesul unui cont în cadrul aplicației temporar sau permanent.

Arhitectura aleasă, reprezentată de un model client-server bazat pe microservicii oferă un potențial de scalabilitate ridicat datorită modului de izolare a responsabilităților. Aplicația poate fi scalată orizontal pentru a permite mai multor utilizatori să o folosească în același timp. Însă, capacitatea clusterului de a executa un număr de joburi și a produce rezultate satisfăcătoate, în timp util, rămâne aceeași. Din această cauză, scalarea orizontală a aplicației presupune scalarea clusterului, atât vertical, prin îmbunătățirea infrastructurii hardware a unui nod, cât și orizontal, prin adăugarea mai multor noduri care să poate prelua joburi.

4.4. Aspecte legate de încărcarea procesorului, memoriei, limitări în ce privește transmisia datelor/comunicarea

Din cauza faptului că microserviciile creează pentru fiecare job MPI trimis către cluster un fir de execuție dedicat, pentru a aștepta rezultatele, sistemul se poate dovedi consumator de resurse în cazuri de utilizare intensă. Încărcarea depinde de numărul total de joburi pe care utilizatorul îl permite pe cluster, acesta fiind numărul maxim de fire de execuție pe care microserviciul de gestiune al joburilor îl poate crea la un moment dat pentru așteptarea rezultatelor, în caz de utilizare maximă a clusterului.

Cotele setate de administrator inflențează comportamentul clusterului în cazul aplicației și poate limita numărul de utilizatori care folosesc clusterul la un moment dat. Spre exemplu, presupunând că administratorul permite doar 30 de procese active în același timp pe cluster, dacă avem 3 conturi utilizator care crează procese la un moment dat, iar cota acestora este de 10 procese pe nod, capacitatea a fost atinsă, iar joburile ulterioare ale acestora sau joburile altora utilizatori vor fi pornite după eliberarea clusterului. În schimb, dacă avem 6 utilizatori cu o cotă de 5 procese, mai mulți utilizatori vor beneficia de preluarea instantanee a jobului MPI.

Un alt factor care influențează utilizarea resurselor de către aplicație este dimensiunea executabilului utilizat, întrucât acesta va fi stocat sub o reprezentare `Base64`, care poate crește rapid în dimensiune. De vreme ce acesta este stocat în baza de date, dimensiunea acestuia poate influența dimensiunea bazei de date.

Mai jos se prezintă o serie de metrici privind resursele utilizate de către instațele containerelor utilizate pentru persistența datelor în urma unei utilizări cu încărcare medie:

Tabelul 4.1. Metrici privind containerele MongoDB

Nume	**CPU (%)**	**Utilizare/Limitare Memorie**	**Citire/Scriere pe Disc**
AuthDB	0.48	191.4 MB / 7.61 GB	4.21 MB / 9.72 MB
JobDB	0.47	190.5 MB / 7.61 GB	2.88 MB / 10.1 MB
ClusterStatusDB	0.43	202.1 MB / 7.61 GB	5.88 MB / 11.5 MB

Deși relativă, măsurarea timpului de răspuns al rutelor REST poate constitui o modalitate de a măsura latențele întâlnite de utilizator. Se observă că valorile sunt relativ în parametri normali. Cel mai lung timp de așteptare apare atunci când se trimite un job către cluster, datorită transmiterii datelor de la interfața web către cluster, prin microservicii, însă acesta este tratat la nivel de interfață prin afișarea unei animații de încărcare. Pot apărea latențe adiționale în urma validării token-ului JWT, din cauza faptului că aceasta se realizează de către microservicii, în urma unei cereri HTTP(S) provenite de la interfață.

Tabelul 4.2. Timpul de răspuns aproximativ al unor rute din cadrul microserviciilor

Endpoint	**Time (ms)**
/api/login	797
/api/signup	613
/api/profile	679
/api/upload	1160
/api/kill	567
/api/admin/users	8
/api/clearhistory	343
Mesaje WebSocket	∼10–50

4.5. Utilizarea sistemului și rezultate experimentale

În cele ce urmează, se va prezenta un fir logic de utilizare a aplicației, din perspectiva unui utilizator final și a administratorului. Inițial, acesta este direcționat către pagina de autentificare. În cazul în care nu are un cont deja creat, va trebui să acceseze butonul de creare a unui cont, care îl va redirecționa către pagina dedicată.

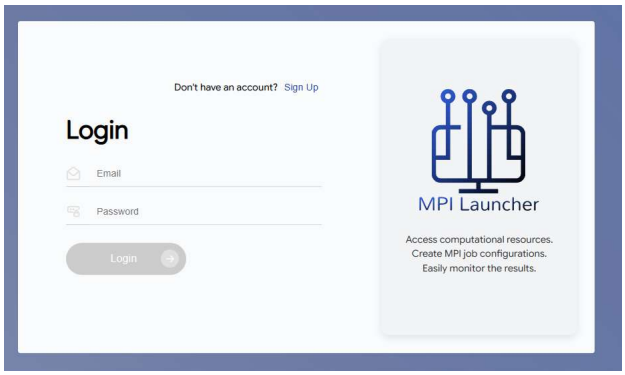

Figura 4.1. Paginare de autentificare

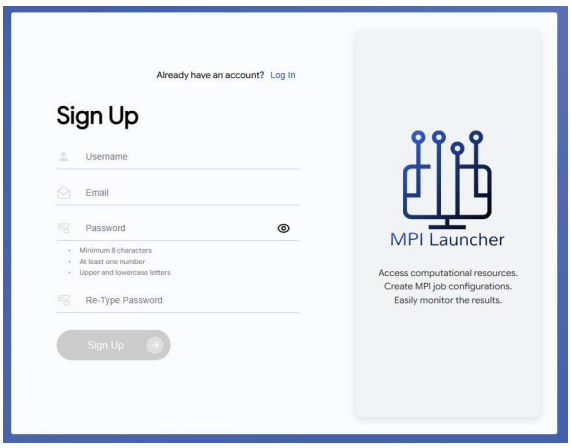

Figura 4.2. Pagina de creare a unui cont

După crearea unui cont, utilizatorul este redicționat către pagina de creare de joburi MPI. Aici trebuie să specifice metadate despre job, precum nume, și descriere, să încarce executabilul MPI în browser și să specifice parametrii de configurație MPI, iar apoi, să selecteze distribuția proceselor pe noduri.

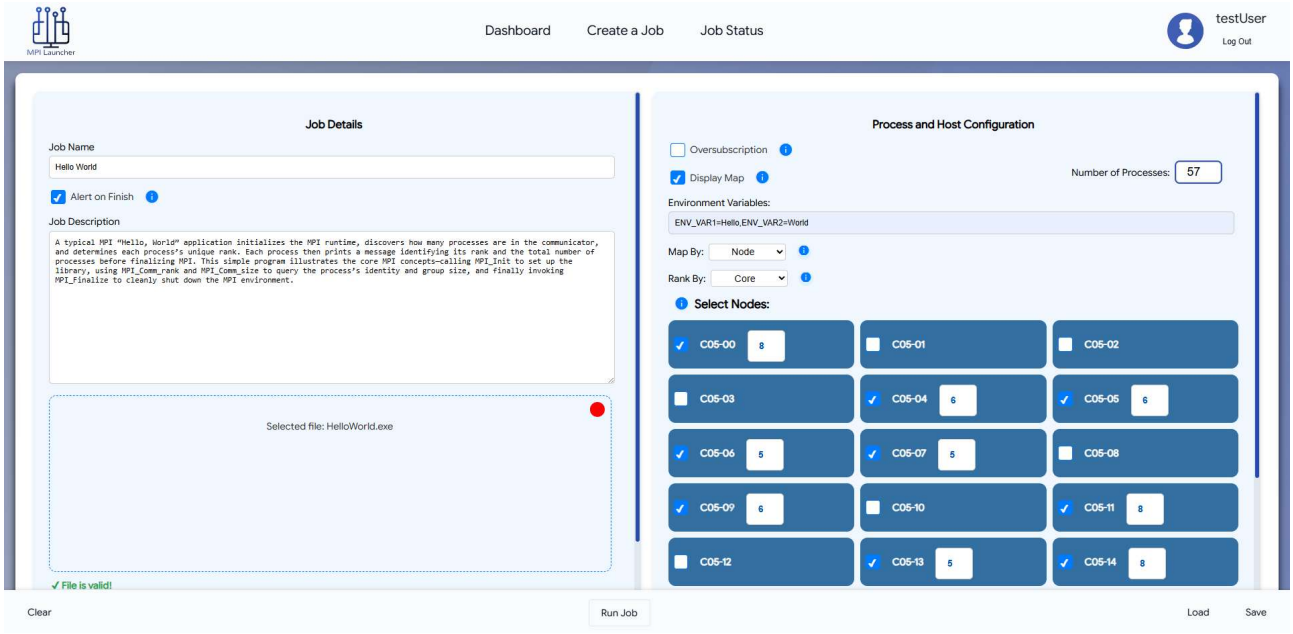

Figura 4.3. Crearea unui job MPI

După crearea unui job MPI, acesta va fi redirecționat către pagina de afișare a rezultatelor. Joburile sunt afișate sub formă de listă, în ordine cronologică, pentru fiecare dintre ele fiind vizibil numele, statusul său în acel moment, și o serie de butoane care oferă utilități, menționate anterior. Utilizatorul are opțiunea de a șterge istoricul joburilor sale, setare care se va propaga inclusiv în cadrul clusterului.

Figura 4.4. Istoricul joburilor MPI

Pentru a vizualiza rezultatele unui job, utilizatorul poate selecta butonul `expand`. Aici sunt afișate toate datele acelui job, se poate redescărca execitabilul sau fișierul `hostfileaaa`, iar la final este prezentat rezultatul transmis de către cluster.

Figura 4.5. Vizualizarea rezultatelor unui job MPI

Pagina dashboard oferă date despre joburile create până în acel moment sub formă de grafic și permite vizualizarea restricțiilor privind utilizarea clusterului:

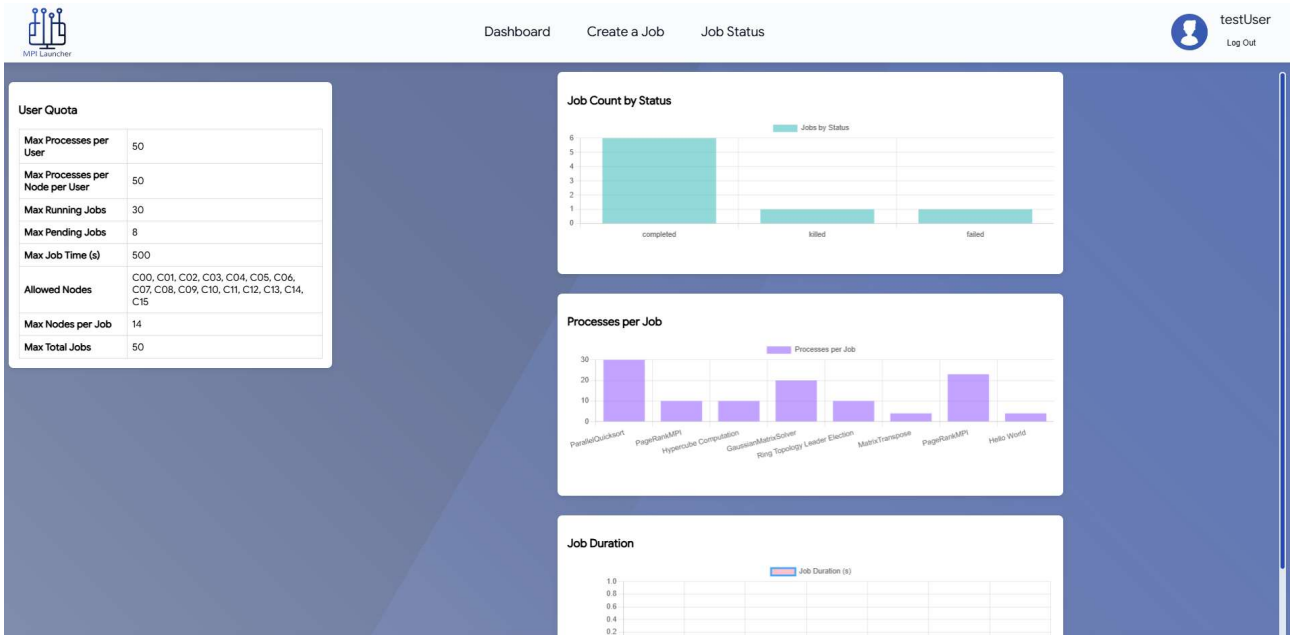

Figura 4.6. Pagina Dashboard

După cum a fost menționat anterior, contul administrator are acces la o pagină specială de unde poate vizualiza limitările fiecărui utilizator în utilizarea clusterului. De aici le poate modifica și influența disponibilitatea microserviciilor de a procesa joburi.

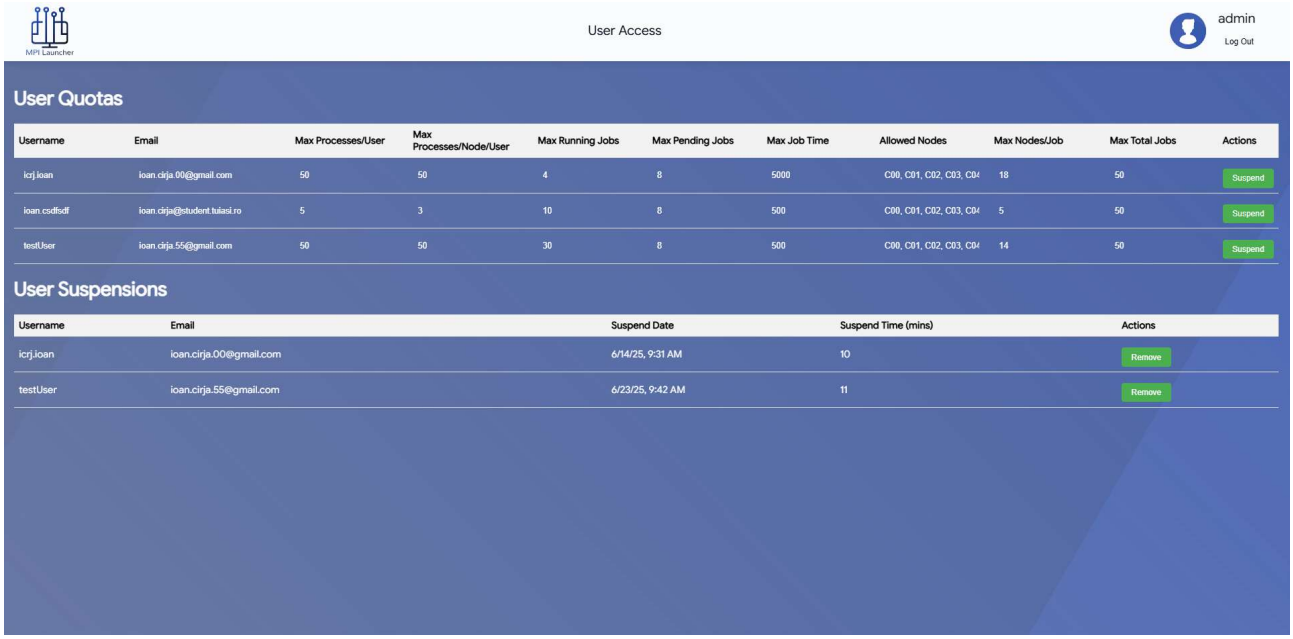

Figura 4.7. Pagina Administrator

Capitolul 5. Concluzii

Lucrarea de față prezintă proiectarea și implementarea unei aplicații web, realizată cu scopul de a oferi un nivel de accesibilitate ridicat în utilizarea OpenMPI pe un cluster alcătuit din multiple calculatoare interconectate între ele. Ideea de bază este reprezentată de crearea proceselor OpenMPI, aferente unei aplicații paralelizate, care apar în urma folosirii unui utilitar precum `mpirun` utilizând o configurație specificată de către client, într-un mod interactiv și grafic, în locul utilizării liniei de comandă. În urma pornirii unor astfel de sarcini, pe care, în cadrul lucrării, le-am numit joburi MPI, rezultatele vor fi disponibile utilizatorului în timp real în cadrul interfeței.

Ideea de utilizare a unei platforme web pentru a expune accesul la cluster distribuit este din ce în ce mai abordată.Acest lucru se datorează nivelului de abstractizare al complexității utilizării infrastructurii hardware pe care aceasta o oferă. Cu toate acestea, crearea unor aplicații care realizează acest lucru, din perspectiva MPI, nu este des întâlnită. Totuși, așa cum se observă în [7], [9], [8], [10], folosirea MPI din browser poate crește productivitatea studenților începători, care studiază subiecte legate de programarea paralelă și distribuită, ce folosește MPI. Pe lângă aceasta, aplicația poate veni în ajutorul oricărui client care dorește accesul rapid la o tehnologie găzduită pe o infrastructură hardware la distanță, expusă într-un mediu web.

Platforma web abstractizează lansarea unui job MPI folosind comanda `mpirun` prin punerea la dispoziție a unei interfețe grafice, de unde utilizatorul poate alcătui o configurație, încărca un executabil și specifica distribuția proceselor în cadrul clusterului. Aceasta urmărește să ușureze utilizarea noțiunilor de programare paralelă și distribuită pentru utilizatorii care nu necesită funcționalități avansate expuse de implementarea OpenMPI a standardului MPI.

Lansarea joburilor OpenMPI se realizează într-un mod controlat, pentru a evita supraîncărcarea clusterului cu procese și a nu reduce performanța semnificativ. Pentru acest lucru am implementat o logică care rulează în paralel și gestionează modul de trimitere a datelor către cluster. De asemenea, am implementat o serie de permisiuni de utilizare a aplicației, care impactează comunicarea dintre utilizatorul final și clusterul pus la dispoziție.

Ca și arhitectură, am ales să implementez un model client-server, bazat pe microservicii web pentru facilitarea logicii de afaceri și pe o interfață grafică pentru a expune funcționalitățile implementate de acestea. Am utilizat noțiuni ale stilului arhitectural REST, propus de Roy Fielding în 2000, la care am adăgat elemente de comunicare bidirecțională, prin utilizarea canelelor WebSocket. De asemenea, am utilizat o serie de librării care facilitează fire de execuție paralelă și sarcini de fundal pentru a gestiona elementele de asincronicitate apărute în lansarea unor procese de la distanță în cadrul unei infrastructuri hardware distribuite. Pentru persistența datelor, am ales o soluție non-relațională, distribuită către fiecare microserviciu.

Astfel, aplicația dezvoltată marchează obiectivele prezentate la începutul lucrării. Utilizatorii se pot autentifica și trimite joburi multiple către cluster. Microserviciile vor analiza datele utilizatorului, limitările asociate contului său, și încărcarea clusterului și vor trimite către acesta jobul în așa fel încât supraîncărcarea să nu aibă loc. Administratorul poate gestiona cotele clienților și efectua modificări, astfel împactând utilizarea clusterului.

Limitările principale ale platformei provin din perspectiva numărului de utilizatori și a scalării, întrucât există o dependență de infrastructura hardware de care dispune clusterul. În plus, din perspectiva funcționalităților expuse în dezvoltarea programelor MPI, aplicația nu poate facilita dezvoltarea de cod MPI sau depanarea, ci pornește de la un executabil deja compilat.

În concluzie, aplicația prezentată constituie o soluție ce oferă accesibilitate și automatizare în utilizarea OpenMPI în cadrul unui cluster. Prin introducerea acestui nivel de abstractizare, dezvoltatorii MPI pot accesa resurse computaționale de la distanță și dezvolta programe paralele fără a necesita configurări inițiale.

5.1. Direcții viitoare de dezvoltare

Aplicația poate fi dezvoltată în ceea ce privește funcționalitățile MPI pe care le oferă. Astfel, ca direcții viitoare s-a propus introducere unui mediu dezvoltare integrat, pentru a facilita dezvoltarea de cod MPI în cadrul aplicației, fără a fi necesară o compilare externă. Clientul va putea dezvolta aplicații C/C++ și apela librăria MPI direct din browser, iar compilarea și executarea jobului se va realiza pe cluster. De asemenea, vor fi adăugate mai multe flag-uri MPI pentru a facilita configurații mai complexe, și se poate introduce un prototip pentru modul de depanare în cadrul mediului de dezvoltare.

O altă perspectivă de dezvoltare este introducerea mai multor implementări ale standardului MPI în cadrul aplicației, pe care utilizator le poate alege, și scalarea clusterului. Numărul de joburi pe care utilizatorii le pot porni la un moment dat este direct proporțional cu capacitatea clusterului de procesare.

De asemenea, scalarea orizontală a aplicației, condiționată de dezvoltarea infrastructurii hardware a clusterului și găzduirea acesteia într-un mediu dedicat constituie modalități prin care aplicația poate răspunde unui public mai larg.

5.2. Lecții învățate pe parcursul dezvoltării proiectului de diplomă

Dificultatea principală întâlnită în dezvoltarea platformei web provine din caracterul asincron al gestiunii și monitorizării joburilor și al comunicării cu clusterul. Am învățat cum pot fi create sarcini asincrone și paralele în cadrul unei aplicații web, care urmărește multiple șabloane de comunicare. Am folosit noțiuni ale stilului arhitectural REST pentru crearea și gestionarea utilizatorilor și a joburilor MPI, dar și WebSockets pentru a dispune utilizatorului actualizări privind starea clusterului și a joburilor sale în timp real.

De asemenea, am aprofundat noțiuni privind modul de operare al daemonilor `orted` din cadrul OpenMPI, precum modul de creare a joburilor MPI, folosind diferiți parametri de configurare si cum pot fi acestia abstractizați într-un mediu web.

Dezvoltarea aplicației a necesitat transpunerea unei tehnologii strâns legate de infrastructura hardware, bazată pe medii HPC și linii de comandă, într-un mediu web, orientat pe interfață grafică. Am dezvoltat un strat de microservicii FastAPI care traduc cererile REST în comenzi `mpirun`, lansate prin SSH, iar rezultatele și potențialele erori sunt transmise înapoi clientului prin canale WebSocket.

Bibliografie

[1] (2025) Standardul MPI. [Online]. Available: https://www.mpi-forum.org/docs/

[2] D. Shadija, M. Rezai, and R. Hill, "Towards an understanding of microservices," in *2017 23rd International Conference on Automation and Computing (ICAC)*, 2017, pp. 1–6.

[3] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[4] M. F. Ali and R. Z. Khan, "Distributed computing: An overview," *International Journal of Advanced Networking and Applications*, vol. 7, no. 1, p. 2630, 2015.

[5] (2025) Documentație OpenMPI. [Online]. Available: https://www.open-mpi.org/

[6] 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.

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

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

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

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

[11] 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.

[12] (2025) Documentație Paramiko. [Online]. Available: https://www.paramiko.org/

[13] (2025) Documentație AsyncSSH. [Online]. Available: https://asyncssh.readthedocs.io/en/latest/

[14] (2025) Documentație Asyncio. [Online]. Available: https://docs.python.org/3/library/asyncio.html

[15] 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.