Folyamatok hibatoleráns futtatása számítógépfürtön

KATONA ZOLTÁN

Budapesti Műszaki és Gazdaságtudományi Egyetem, Szélessávú Hírközlés és Villamosságtan Tanszék
kz340@hszk.bme.hu

Kulcsszavak: nagy kapacitást igénylő programok, MPI, együttműködési hibák

A cikkben két hibatűrő rendszert mutatok be, melyet az egy-két processzoros személyi számítógépekből álló számítógépfürtökre dolgoztak ki. Jelen esetben hiba alatt az olyan véletlenül bekövetkező eseményeket értjük, melyek miatt egy, vagy több számítógép többé nem része a számítógépfürtnek. A kiváltó ok lehet többek között a merevlemez, memória, alaplap, vagy a processzor meghibásodása, áramszünet, de akár az operációs rendszer, vagy bármelyik létfontosságú szoftver lefagyása is. A hibatűrő rendszerek legfontosabb feladata, hogy a több hétig, hónapig futó nagy számításigényű alkalmazást ne kelljen újraindítani egy ilyen nem várt esemény miatt. Biztosítaniuk kell az alkalmazás zavartalan futását, amelyet a hibák detektálásával, illetve ezek kiküszöbölésével érhetnek el.

Bevezetés

A kutatás és a tudományok területén egyre nagyobb szükség van olyan számítógépes háttérre, amely támogatja a nagy számításigényű, nagy pontosságú alkalmazások *(HPC – High Performance Computing)* futtatását. Ezekhez a feltételekhez a legalkalmasabb környezetet az igen drága, azonban gyors és megbízható, több processzoros, nagy memóriával és háttértárral rendelkező szuperszámítógépek nyújtják. Többnyire az egyetemek, kutatóközpontok nem engedhetik meg maguknak, hogy saját célra ilyen eszközt vásároljanak, ezért sorba kell állniuk, hogy használhassák a világ valamelyik szuperszámítógép-központjánál rendelkezésre álló kapacitást. A helyzet azonban enyhült azóta, hogy Magyarországon a Nemzeti Információs Infrastruktúra Fejlesztési Iroda Szuperszámítógép Központjában [1] 2001-ben üzembehelyeztek egy mára, 196 processzorosra bővült Sun szuperszámítógépet.

Sajnos ennek ellenére is fennállnak a fent említett nehézségek, amelyeknek a kiküszöbölésére kidolgoztak egy megoldást, melyben olcsó, egy-két processzoros, kis számítási kapacitással rendelkező személyi számítógépeket kötnek össze egy hálózattal *(Workstation Cluster)*, hogy az együttes számítási teljesítményük elég nagy legyen ahhoz, hogy megközelítsék a szuperszámítógépekét. Ez a megoldás két problémát vet fel:

- Hogyan lehet elérni a számítógépek és a rajtuk futó folyamatok (processzek) összehangolt működését?
- A személyi számítógépek olcsóságukból eredően megbízhatatlanok lehetnek, vagyis a meghibásodásig tartó idő várhatóértéke sokkal kisebb, mint a szuperszámítógépeké *(MTBF – Mean Time Between Failures)*.

Az 1990-es évek elején az első problémakör megoldására hozta létre az MPI Forum több mint 40 szervezet részvételével az MPI szabványt [2, 3]. Az üzenettovábbító illesztő felület *(MPI – Message Passing Interface)* célja az, hogy a gyakorlatban is alkalmazható, hordozható, hatékony és rugalmas felületet biztosítson üzenettovábbítás céljából. Ennek az interfésznek a segítségével lehet megoldani több számítógépen futó folyamatoknak a hatékony kommunikációját. A szabvány által definiált MPI a viszonyréteget és a megjelenítési réteget foglalja el az ISO OSI hétrétegű modelljében [4, 5]. Olyan értéknövelt szolgáltatásokat nyújt, mint a folyamatok szinkronizálása, a feladat szétosztás, másrészt foglalkozik a továbbítandó információk szintaktikájával és szemantikájával, amivel az adatábrázolás különbözőségeiből eredő problémákat kiküszöböli a heterogén rendszerekben (SGI IRIX, DEC Alpha stb.).

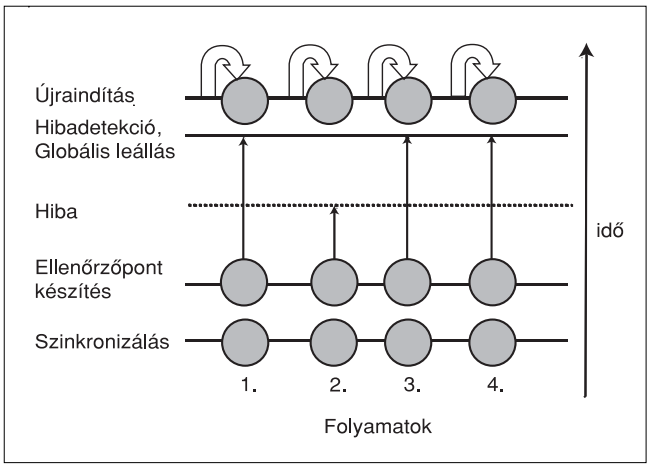
A második probléma megszüntetésére különböző hibadetektáló és elhárító technikák jelentek meg az évek során, melyeknek az előbb ismertetett MPI szabványon alapuló két eltérő gyakorlati megvalósítását szeretném bemutatni.

A hibadetektáló és elhárító technikák

A következő pontban leírt hibatűrő rendszerek működésének megértéséhez néhány alapfogalmat tisztázni kell. A cikkben MPI alkalmazásnak nevezzük a számítógépfürtön futó, nagy számítási kapacitást igénylő programot. Az MPI alkalmazás több folyamatból áll, melyek mindegyike ideális esetben egy külön processzoron fut. Ezek a folyamatok egymásnak üzeneteket – például kiindulási adatokat, eredményeket – küldve kommunikálnak, hogy az egész alkalmazás sikeresen befejezze a munkát.

A hibadetektáló és elhárító technikákat három fontos paraméter különbözteti meg: az átlátszóság *(transparency)*, az ellenőrzőpont koordináció *(checkpoint coor-*

dination) és az üzenetek naplózása *(message logging)* [6]. Ahhoz, hogy az átlátszóság teljesüljön, az üzenettovábbító alkalmazásnak képesnek kell lennie mind futási időben az automatikus hibadetektálásra és javításra, mind a hibajavítási folyamatba becsúszó hibáról értesítést adni a programozónak, felhasználónak. Az ellenőrzőpont-állomány *(checkpoint image)* nem más, mint egy, a folyamat futása során keletkező részeredményeket tároló állomány. Ha a számítógép, amelyen a folyamat eddig futott, hiba következtében kiesik a számítógépfürtből, akkor a hibatűrő rendszer ezt a folyamatot egy olyan gépre ütemezi, amely továbbra is a fürt tagja. Amennyiben nem készült ellenőrzőpont-állomány a hiba miatt kiesett folyamatról, akkor az újraütemezés során elölről kell kezdenie a számításokat, ellenkező esetben azonban a részeredmények segítségével a legutolsó közbülső állapotból folytathatja a feldolgozást.



1. ábra
Koordinált ellenőrzőpont-állomány készítés

Az ellenőrzőpont koordinációnak két fontos típusa van. A koordinált, illetve a nem koordinált változat. A koordinált esetben, ahogy az *1. ábra* is mutatja, minden folyamatot szinkronizálni kell, vagyis be kell szüntetniük a hálózati kommunikációt, hogy ne vesszen el információ az ellenőrzőpont-állományok készítésekor. Amennyiben egy folyamatot újra kell indítani, mert kiesett az a számítógép, amelyen eddig futott, akkor mindegyik folyamat újraindul a legfrissebb ellenőrzőpontról. Sajnos ezek a tulajdonságok okozzák, hogy ez a módszer nem skálázható, mivel nem lehet csak a kiesett folyamatokat újraütemezni, hanem mindegyiket újra kell indítani az utolsó ellenőrzőpontról.

Ez nagyban növeli a rendszer sérülékenységét, hiszen ha nem túl sűrűn készítünk ellenőrzőpont-állományokat, akkor értékes processzor idők veszhetnek el hiba esetén, akár csak egy gép kiesésekor is, mivel így mindegyik számítógép munkája elvész. Ha sűrűbben készítünk ellenőrzőpont-állományokat, akkor ez kevésbé hangsúlyos, eltekintve az ehhez szükséges többlet időtől.

A nem koordinált esetben egymástól függetlenül, szinkronizálás nélkül, eltérő időpontban készíthető mindegyik folyamatról ellenőrzőpont-állomány, így a rendszer skálázható, vagyis elegendő csak a kiesett folyamatokat újraütemezni. Nem koordinált esetben a folyamatok nem szüntetik be az ellenőrzőpont-állományok elkészítésekor a hálózati kommunikációt.

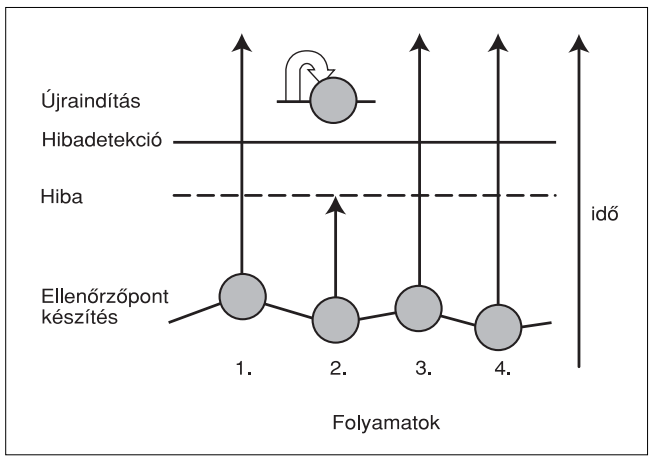
Ez azt jelenti, hogy a hálózaton levő üzeneteket naplózni kell, mivel az ellenőrzőpont-állományok nem hordoznak semmiféle információt ezekről. Vagyis, ha egy folyamat üzenetet küld egy másiknak – például egy kiindulási adatot – és a címzett kiesik, nem kapja meg az üzenetet, akkor az újraütemezett folyamat várni fogja az üzenetet, de a feladó abban a hitben él, hogy a címzett már megkapta. Ez végső soron az egész alkalmazás fennakadásához vezethet. A *2. ábra* a nem koordinált ellenőrzőpont-állomány készítésre mutat egy példát.

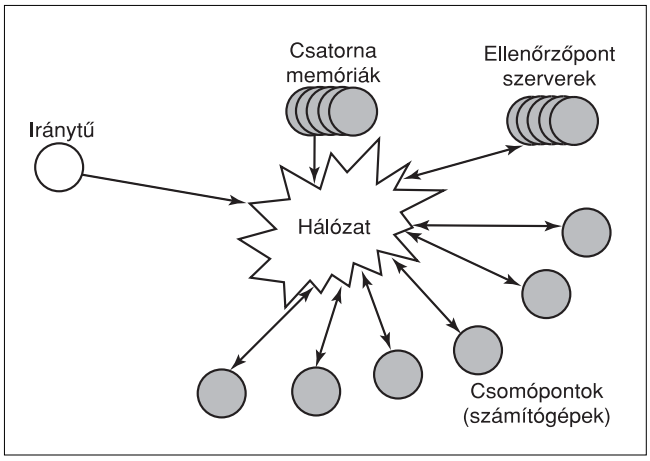
Ha a hibatűrő rendszer nem készít ellenőrzőpont-állományokat, akkor a rendszer csak a kommunikációs naplókra hagyatkozhat a kiesett folyamatok újraütemezésekor, vagyis a folyamatokat nem lehet részeredmények segítségével egy közbülső állapotból újraindítani. Az egész alkalmazást a számítás legelejétől újra kell indítani. Ebben az esetben a kommunikációs naplóknak az a haszna, hogy a folyamatoknak nem kell várniuk az üzenetekre, mert az a kiesés pillanatáig rendelkezésre áll.

Az üzenetek naplózása is többféle lehet. Létezik pesszimista és optimista naplózás. Pesszimista esetben megbízható adattároló eszközre mentik az üzeneteket, amelyeknek nagy az MTBF-je, ezért igen kicsi valószínűséggel vesznek el erről adatok. Optimista esetben nem megbízható adattárolóra mentik az üzeneteket.

Amennyiben egy számítógép tönkremegy, akkor a folyamatot más számítógépek naplóinak megfelelően indítják újra, viszont ha egynél több számítógép hibásodik meg, akkor az utolsó koherens ellenőrzőpontról történik a folyamat újraindítás, mivel ha rosszul van megtervezve a rendszer, akkor a kiesett gépek közötti kommunikáció is megszakad, amit csak úgy lehet kiküszöbölni, hogy a folyamatokat a koherens ellenőrzőpontokról indítjuk újra.

2. ábra
Nem koordinált ellenőrzőpont-állomány készítés





3. ábra Az MPICH-V rendszer felépítése

Hibatűrő megoldások, előnyeik és hátrányaik

Az alapok tisztázása után rátérek néhány gyakorlati példa részletezésére. Az első hibatűrő rendszer, amelyet bemutatok az MPICH-V [6], (Cluster&GRID group, Laboratoire de Recherche en Informatique, University of Paris South). Ez a hibatűrő rendszer azt feltételezi, hogy az MPI alkalmazás futása során keletkező hibák a számítógépek meghibásodása miatt keletkeznek. Ennek az elgondolásnak az architektúrája több elemből áll. Megbízható csatornamemóriák *(Channel Memory)*, megbízható ellenőrzőpont szerverek *(Checkpoint Server)* és egy irányító *(Dispatcher)* alkotják a csomópontokkal *(Node)* együtt a rendszert, ahogyan a *3. ábra* is mutatja.

A csatornamemóriák feladata, hogy naplózzák az MPI folyamatok közötti üzenetváltást. Az MPI folyamatok valójában nem egymással kommunikálnak, hanem egy-egy csatornamemóriával. A csomópontok csoportokba vannak szervezve, és mindegyikhez tartozik egy csatornamemória. Amennyiben egy csomópont üzenetet vár, akkor azt a saját csoportjához tartozó csatornamemóriától fogja megkapni, viszont ha üzenetet akar küldeni, akkor a címzett csoportjához tartozó csatornamemóriának kell elküldeni. A csatornamemóriák FIFO elven működnek, vagyis az először beérkező üzenet hagyja el először a memóriát.

Ezzel és a több csatornamemória felhasználásával, valamint a csomópontok csoportokba szervezésével szerették volna a tervezők elérni a koordinációs üzenetek csökkentését és a vevő számára az üzenetek teljes sorrendbe szervezését. Egy többszálú szerver végzi az esemény kezelését, vagyis a beérkező és a kimenő üzenetekkel kapcsolatos teendőket. Üzenetek nem csak a csomópontoktól érkezhetnek, hanem a csomópontokhoz csatolt ellenőrzőpont szerverektől, és az irányítótól is. Ezek többnyire vezérlő üzenetek. A többszálú szerver egy FIFO memóriába teszi az üzeneteket, ahonnan egy megbízható adattárolóra kerülnek, így abban az esetben, ha egy csomópont tönkremegy, akkor mintegy „újra lejátszható” a kommunikáció az újraindított MPI folyamattal. A legfrissebb ellenőrzőpont-állományok létrehozásának dátumánál régebbi üzeneteket törlik az adattárolóról.

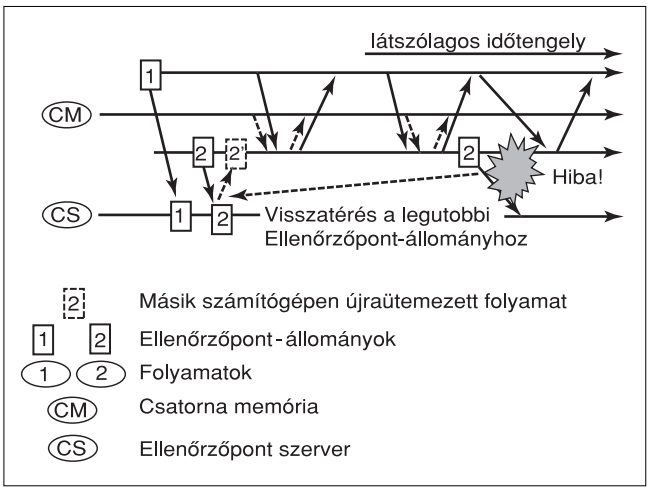
Az előző pontban tárgyaltak alapján a csatornamemóriák pesszimista típusú naplózást végeznek, mivel megbízhatóak. A megbízhatóság miatt a hardvernek szigorúbb követelményeket kell kielégítenie, így igen drága. Az ellenőrzőpont-szerverek tárolják az ellenőrzőpont-állományokat, amelyek a folyamatok egy korábbi állapotát írják le. Minden csomóponthoz egy fájl tartozik a szerveren.

Az ellenőrzőpont-állományok készítését kiváltó eseményeket nem kívülről – például az irányítótól – kapják a csomópontok, hanem lokálisan, adott időközönként érkeznek meg. Az algoritmus egy olyan (fork()) rendszerhívással kezdődik, amely az MPI folyamatról egy másolatot készít. A másolat minden hálózati kapcsolatot lezár, így minden kommunikációt megszakít, ezzel lehetővé válik az ellenőrzőpont-állomány elkészítése. Amikor elkészült a kép, a folyamat másolata befejezi a futását. Az ellenőrzőpont-állományt ezután a csomópont elküldi az ellenőrzőpont szervernek. A megoldás előnye, hogy az eredeti folyamatnak eközben nem kell megszakítania a futását. A csatornamemóriákhoz hasonlóan az ellenőrzőpont szervereknek is megbízhatónak kell lenniük, tehát ez a tulajdonság is hátrányok közé sorolható.

A következő rendszerelem az irányító. Az irányító többek között a parancsvégrehajtás inicializálását végzi, vagyis ellenőrzi, hogy a rendszerelemek készen vannak-e, csoportokba szervezi a számítást végző csomópontokat és csatornamemóriát rendel hozzájuk, továbbá figyeli a csomópontok állapotát, vagyis hogy érkezik-e a csomópontoktól „életjel”, vagy van-e időtúllépés. Emellett elindítja a megfelelő példányszámban a programokat az egyes számítógépeken, illetve ha egy MPI folyamat „halott”, akkor azt a fennmaradt csomópontok valamelyikén újraütemezi.

Ennek az összetett rendszernek a működését mutatja a *4. ábra*. Az ábrán a legrosszabb eset *(Worst Case)* látható, mivel a hálózaton levő ellenőrzőpont-állomány,

4. ábra Legrosszabb eset: üzenet és ellenőrzőpont-állomány elveszik



és az üzenet is elvész, hiszen az a számítógép, amelyiken a 2. MPI folyamat futott, tönkrement. Ezt az irányító veszi észre, mivel a számítógép nem küldött életjelet magáról. Ekkor az irányító a 2. MPI folyamatot egy másik csomópontra ütemezi úgy, hogy az „új" számítógép a 2. folyamat futtatásához szükséges ellenőrzőpont-állományt az ellenőrzőpont szervertől kapja meg. Az újraütemezett 2. folyamat (2') az ellenőrzőpont-állomány elkészítésének időpontjától az újabb kommunikációt a csatorna memóriával játssza le, mivel az 1. folyamat a köztük levő üzenetváltásnak ezen a részén már régen túl esett, vagyis a két folyamat emiatt, és a rendszer architektúrája miatt sem tud egymással közvetlenül kommunikálni.

Az ábrán a csatorna memória és a 2. folyamat, illetve a 2'. folyamat közötti kommunikációt jelző folyamatos, illetve szaggatott vonal szinte egymást fedik, de valójában időben nem egyszerre zajlanak az üzenetváltások, ezért látható az ábrán a látszólagos időtengely felirat.

A rendszer előnyei és hátrányai tehát a következők. Az irányító nem redundáns, emiatt végzetes hiba következhet be a kiesésekor. A csatornamemóriáknak és az ellenőrzőpont szervereknek megbízhatóaknak kell lenniük, ami tetemes összeggel megemeli a rendszer árát. A rendszer teljesítményét rontja, hogy a minden üzenetnek kétszer kell a hálózatra lépnie, mivel az MPI folyamatoknak a csatornamemóriákon keresztül kell kommunikálniuk. A hálózatterhelés főleg nagy méretű üzenetek esetén mutatkozik meg.

A rendszer előnyei közé sorolható az, hogy az összes MPI folyamat „halálát" túl tudja élni, mivel az ellenőrzőpont szerverek a folyamatok konzisztens ellenőrzőpont-állományainak halmazát tartalmazzák, továbbá a csatornamemóriákban a teljes rendszer kommunikációja el van mentve, ezzel lehetővé téve a rendszer gyorsabb helyreállítását. További előnyt jelent az, hogy az MPI folyamat leállása nélkül lehet ellenőrzőpont-állományt készíteni a folyamatról.

Az elmúlt években mások is foglalkoztak ezzel a témával, más szemszögből megközelítve a problémát. Az MPI/FT [7] (Mississippi State University, Department of Computer Science; MPI Software Technology, Inc.; NASA Jet Propulsion Laboratory, California Institute of Technology) módszer feltételezi, hogy a programozó által megírt MPI alkalmazás futása során keletkező hibák egy-egy csomópont meghibásodásából, továbbá véletlenszerű bithibákból eredhetnek. Tehát az előzőekben vizsgált rendszertől az MPI/FT ezzel is többre képes. Ezeket a bithibákat okozhatják a vezetékeken fellépő elektromágneses zavarok, áthallások. Feltételezi továbbá, hogy a processzor második szintű (L2) gyorsítótára mind a memória külső zavarok, mind az űrből érkező nagyenergiájú töltött részecskék ellen védve van, így a véletlen bithibák nem okozhatnak ezeken a helyeken gondot.

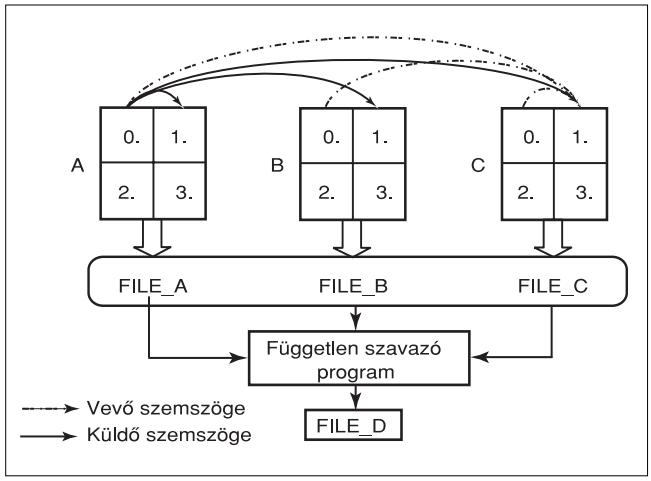
A hibadetektálásnak és javításnak több módszerét veti fel az MPI/FT. Önellenőrző szálak *(SCT – Self-Checking Thread)* használatát javasolja, amelyek különböző feladatokat töltenének be. Folyamatok globális adatstruktúráira szavaznának egyszerű többségi döntéssel, továbbá lokális adatokat több példányban tárolnának és időközönként szintén többségi szavazással eldöntenék, hogy melyikük tartalmaz helyes adatokat. Ezekre a szavazásokra főleg olyan helyeken van szükség, ahol gyakran előfordulhatnak az adatokban véletlenszerű változások, például bithibák. Ilyen környezet tipikusan a nagyszámú nagyenergiájú, illetve töltött részecskéket tartalmazó hely, például az űr. További feladatuk lehetne egy nem blokkoló kollektív függvény időnkénti meghívása, mellyel észlelni lehetne a kiesett MPI folyamatokat, mivel ezek nem hívják meg a függvényt, így az a hívó oldalon időtúllépéssel hibát fog jelezni. Feladatuk lenne még a folyamatok közötti kommunikáció és a belső dinamikus memória lefoglalás figyelése is.

A hibatűrő rendszer részét képezi a koordinátor *(Coordinator)* is, amely az előzőekben taglalt csatornamemóriához és irányítóhoz hasonlóan működik. Ez a koordinátor az SPMD *(Single Program, Multiple Data)* alkalmazásoknál egy különálló számítógép lehet, illetve a mester/szolga modellben a mester töltheti be az adott funkciót. A tudományos programok jelentős része a mester/szolga vagy az SPMD modellt követi. Az SPMD modell lényege, hogy minden processzor ugyanazt a programot hajtja végre, de a folyamatok futása minden processzoron más-más irányt vehet. Mivel ezek a modellek a legelterjedtebbek, ezért az MPI/FT is ezekkel tud a legjobban együttműködni.

A koordinátor feladata az MPI alkalmazás folyamatos ellenőrzése, a kiesett folyamat újraindítása egy ellenőrzőpontról, majd a napló alapján a kommunikáció újralejátszása a folyamattal, hogy a rendszer újra konzisztens állapotba kerüljön. A feladatai közé tartozik továbbá az is, hogy az üzenetek számára virtuális csatornaként működjön, mivel így minden kommunikációt naplózni tud. Periodikusan vezérlőüzeneteket kell küldenie az önellenőrző szálaknak, ezenkívül válaszolnia kell az általuk küldött üzenetekre.

A biztosabb végeredmény érdekében a párhuzamos nMR *(n-Modular Redundancy)* módot vezették be

5. ábra Párhuzamos nMR végrahajtás



a tervezők, amelynek a lényege, hogy minden MPI folyamatnak *n* példánya készül az MPI alkalmazás indításakor. Ezt szemlélteti az *5. ábra* [7].

Az ábrán az MPI alkalmazást 4 párhuzamos folyamattal indítjuk el, és mindegyiknek készül két másolata. Az ábrán jól látható, hogy mi történik üzenetküldéskor. Ha a nulladik folyamat az elsőnek üzenetet akar küldeni, akkor azt az első folyamat minden példánya megkapja, illetve ha az első üzenetet vár a nulladik folyamattól, akkor a nulladik folyamat összes példányától megkapja azt. Ekkor a vevő a vett üzeneteket összehasonlítva egyszerű többségi szavazással megállapíthatná, hogy melyik üzenet tartalmaz helyes adatokat. A folyamatok az eredményeket egy-egy fájlban tárolhatják és egy független szavazó program ezeket összehasonlíthatja.

Az MPI/FT hátrányai a központosított irányítás, a koordinátor használata. A koordinátor ment el minden kommunikációt, amely a folyamatok között lezajlik, ami azt jelenti, hogy a koordinátor egy létfontosságú elem (centralizált). A rendszer ezt az nMR mód segítségével szeretné kiküszöbölni, vagyis redundáns koordinátort vezet be. Ez rövid számolás után igen nagy hálózatterhelést jelent.

Tegyük fel, hogy egyetlen folyamat akar üzenetet küldeni egy másiknak. Mivel ezek a folyamatok is nMR módban futnak, ezért mindegyikből van a rendszerben *n* példány. A működési elv alapján így *n* darab folyamat fog *n* másik folyamatnak üzenetet küldeni, ami összesen eddig n^2 üzenetet jelent. Ehhez hozzá kell venni, hogy minden üzenetnek keresztül kell mennie a koordinátoron, vagyis minden üzenet kétszer kerül a hálózatra, tehát $2*n^2$ üzenetnél tartunk. Mivel a koordinátor is nMR módban fut, ez azt jelenti, hogy ugyanez az üzenetmennyiség megjelenik minden egyes koordinátor miatt a hálózaton, vagyis az eredetileg elküldeni szándékozott egy darab üzenetből $2*n^3$ üzenet keletkezett.

Hogy még inkább szemléltessem a probléma súlyát, figyelembe kell venni, hogy egyszerű többségi döntés végrehajtásához *n*-et páratlannak érdemes választani, hogy ne kerüljünk döntésképtelen helyzetbe. Ez azt jelenti, hogy *n*-nek legalább 3-nak kell lennie, vagyis a minimális hálózatterhelés esetén is 1 üzenet elküldése valójában 54 üzenetküldéssel jár. Ezek után már nem is érdemes abba belegondolni, ha az üzenet mérete nő, vagy ha nem csak két folyamat kommunikál, ahogy az előzőekben feltételeztem, hanem több.

Ezek a tények arra engednek következtetni, hogy az MPI/FT-t nem érdemes nagy számítógépfürtökben alkalmazni, hanem inkább kisebb, nagy megbízhatóságú, redundáns rendszerekben lehet hasznát venni, mint amilyenek egy űreszközön is előfordulhatnak. További lehetséges alkalmazási területe a megoldásnak az, hogy dedikált processzorokat alkalmazunk, amelyek pont-pont összeköttetéseken keresztül kommunikálnak egymással, hiszen ekkor a nagy hálózati terhelés megoszlik az összeköttetések között.

Összefoglalás

A cikkben áttekintettem a számítógépfürtökre kidolgozott hibatűrő rendszerek egy részét. Értelmeztem az alapvető fogalmakat, az ellenőrzőpont koordináció és a naplózás típusait, jelentőségüket. Bemutattam az MPICH-V és az MPI/FT hibatűrésre kidolgozott megoldások architektúráját, a hibadetektálási és javítási folyamatuk lényegét. Kifejtettem a rendszerek előnyeit, hátrányait, miszerint az MPICH-V drága, de megbízható, így nem redundáns rendszerelemeket alkalmaz, illetve túl tudja élni akár az összes MPI folyamat „halálát”.

Az MPI/FT ezzel ellentétben olcsó redundáns rendszerelemeket alkalmaz, a hibavalószínűséget a párhuzamos nMR móddal próbálja csökkenteni. Sajnos ez a megoldás a túlzottan nagy hálózatterheléssel jár, ezért nem igazán alkalmas arra, hogy nagy számítógépfürtön használjuk.

Irodalom

[1] Nemzeti Információs Infrastruktúra Fejlesztési (NIIF) Program Szuperszámítógép Központjának honlapja, http://www.iif.hu/szuper/

[2] TLTP High Performance Computing Courseware, High Performance Computing Consortium, http://www.cs.ncl.ac.uk/old/modules/2002-03/csc305/TLTP_HPC_Course/

[3] HP MPI User’s Guide, National Center for Supercomputing Applications, http://archive.ncsa.uiuc.edu/SCD/Hardware/CommonDoc/HP/MPI/1_intro.html

[4] C. J. Beckmann, D. D. McManus, G. Cybenko: ”Horizons in scientific and distributed Computing”, COMPUTING IN SCIENCE & ENGINEERING, January-February 1999, pp.23-30.

[5] ISO 7498, Information Processing Systems – Open System Interconnection – Basic Reference Model, International Standards Organization, Geneva, 1984.

[6] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, Th. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, A. Selikhov: ”MPICH-V: Toward a scalable fault tolerant MPI for Volatile nodes”, SC2002

[7] R. Batchu, Jothi P. Neelamegam, Z. Cui, M. Beddhu, A. Skjellum, Y. Dandass, M. Apte: ”MPI/FTTM: Architecture and Taxonomies for Fault-Tolerant, Message-Passing Middleware for Performance-Portable Paralell Computing”, DSM 2001, May 2001, pp.26-33.