Politechnika Śląska w Gliwicach

Obliczenia Równoległe II

Komunikacja między procesami oraz redukcja danych (MPI)

Autor:
Bartłomiej Buchała

Informatyka SSM, semestr II
Rok akademicki 2016/2017
Grupa OS1

28 listopada 2016

1 Wstęp

Wraz z rozwojem nauk ścisłych pojawiają się coraz bardziej skomplikowane problemy natury naukowej. Do ich rozwiązania niezbędne są komputery o dużej mocy obliczeniowej. Rozwój technologii pozwolił jednak na stworzenie maszyn o większej szybkości obliczeń. Zgodnie z prawem Moore'a, które zakłada, że liczba tranzystorów w procesorach rośnie wykładniczo, moc obliczeniowa jrdnostek centralnych wzrasta dwukrotnie co każde dwa lata. Przy stałym zwiększaniu taktowania procesora, napotkano jednak pewien problen – dla pewnego progu zużycie prądu (a przede wszystkim temperatura pracującego CPU) rosną eksponencjalnie w stosunku do częstotliwości taktowania. W okolicach 2005 roku, większość producentów procesorów zdecydowała się wykorzystać inne podejście – zastosować obliczenia równoległe. W tym celu, zamiast rozwijać coraz to szybsze (a konkretnie – wyżej taktowane) procesory monolityczne, kolejne jednostki centralne miały zostać wyposażone w wielokrotne procesory zintegrowane w jednym obwodzie – tak zwane **procesory wielordzeniowe**.

Dodanie dodatkowych rdzeniów nie rozwiązało jednak w magiczny sposób problemów z wydajnością w przypadku większości istniejących programów. Główną przyczyną był fakt, że spora część algorytmów przygotowana była z myślą o wykonaniu sekwencyjnym – czyli przeznaczonym do wykonaniu na jednym procesorze. W tym przypadku wykonywany kod nie był świadomy obecności innych jednostek obliczeniowych, co uniemożliwiało ich użycie przy wykonywaniu kolejnych rozkazów. Szybkość z jaką wykonywał się taki program była zazwyczaj zbliżona do tej wyliczonej w trakcie użycia jednego procesora. Do doprowadziło do powstania do programów równoległych.

Przez **programowanie równoległe** rozumiemy taką motodę tworzenia algorytmu, która pozwala jednoznacznie wskazać, które fragmenty obliczeń mają zostać wykonane w sposób równoległy na osobnych procesorach. W tym celu wyodrębniono 3 pojęcia:

Program współbieżny (ang. *concurrent*) występuje w przypadku, gdy procesy są wykonywane przez jeden procesor rzeczywisty metodą przeplotu.

Program równoległy (ang. *parallel*) to przypadek, kiedy każdy proces wykonywany jest przez osobną jednostkę obliczeniową, a procesory posiadają dostęp do wspólnej pamięci.

Program rozproszony (ang. *distributed*) występuje, gdy procesy wykonywane są przez odrębne, rozproszone procesory połączone kanałami komunikacyjnymi.

W pierwszym rozpatrywanym przypadku nie dochodzi do prawdziwego wykonania równoległego, ponieważ w dowolnym momencie czasu nie istnieją przynajmniej 2 procesy, które są wykonywane jednocześnie. Obliczeniami zajmuje się jeden procesor, a kolejne rozkazy procesów wykonywane są na zmianę – pomiędzy nimi zachodzi przełączanie kontekstu (zapamiętanie niezbędnych danych dotyczących stanu procesu). W dwóch pozostałych scenariuszach, należy rozwiązać dodatkowo jeden problem: komunikację między procesorami w określonych momentach obliczeń. Istnieją dwa so=posoby realizacji takiego przedsięwzięcia:

- Wykorzystanie **pamięci wspólnej** – zakłada ono istnienie pamięci operacyjnej, w której znajdują się dane potrzebne do obliczeń. Każdy procesor biorący udział w obliczeniach ma dostęp do zmiennych wspólnych (ang *shared variables*), na których może wykonywać określone operacje (np. operacje czytania, zapisu, lub bardziej zaawansowane jak porównanie-zamiana lub czytanie-modyfikacja-zapis).

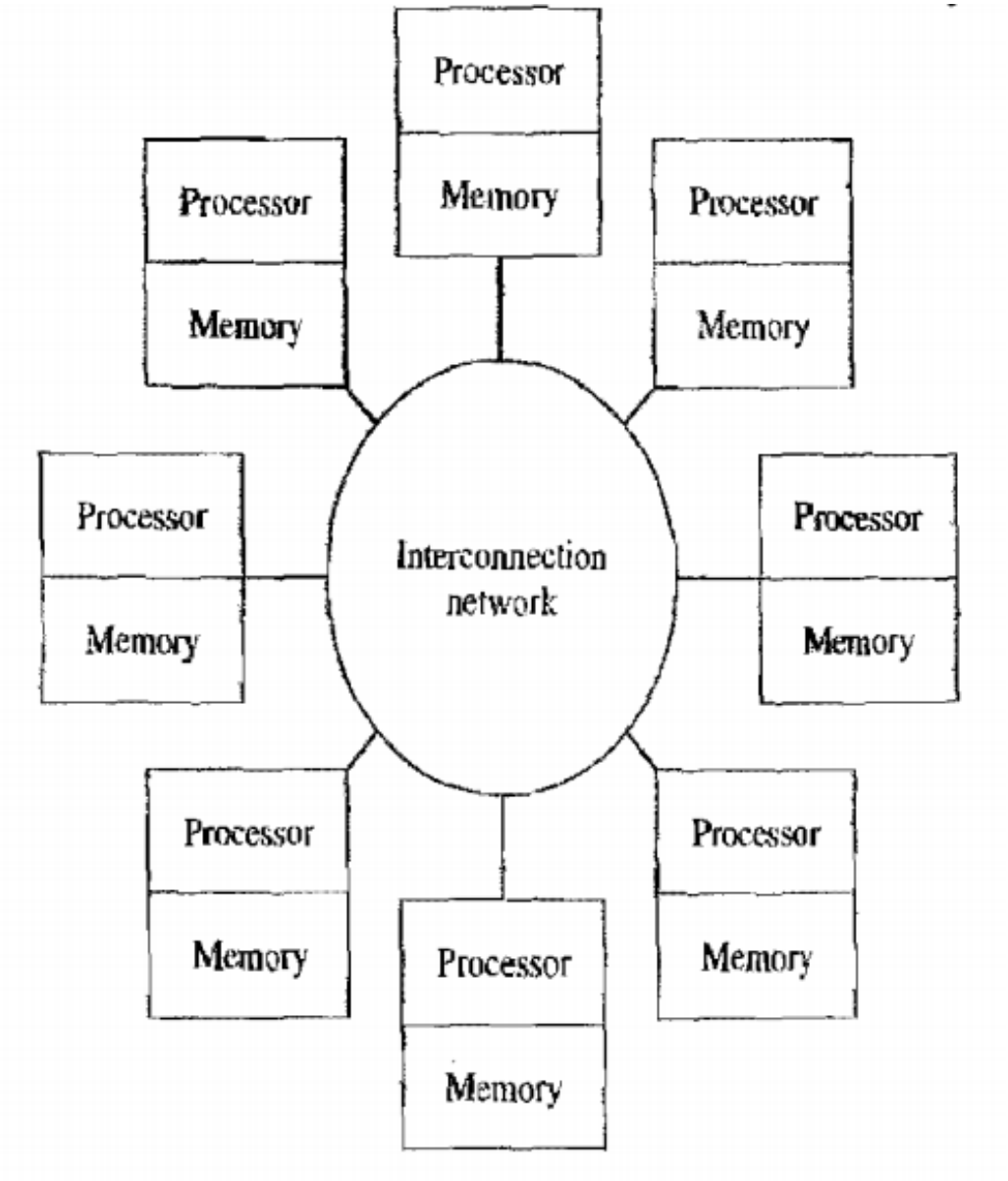
- Przesył wiadomości za pomocą **kanałów komunikacyjnych** – zakłada istenie specjalnych kanałów, poprzez które procesory wysyłają między sobą wirtualne wiadomości. Każdy kanał jest dwukierunkowy i łączy 2 procesory, natomiast ich zbiór stanowi sieć połączeń. Procesory w klastrze mogą był połączone w różne schaematy, np. listy cyklicznej czy macierzy. Obliczenia wykonywane są asynchronicznie, gdyż nie można dokładnie określić momentów, w których operacje wykonywane są współbieżnie, a także momentów wysyłu i odbioru wiadomości pomiędzy poszczególnymi CPU.

W latach 90-tych XX wieku powstały dwa standardy, które miały ułatwić tworzenie i pracę z kodem przeznaczonym do wykonania równoległego: OpenMP (ang. *Open Multi-Processing*), który charakteryzuje się wykorzystaniem pamięci wspólnej) oraz MPI (ang. *Message Passing Interface*), korzystający z kanałów komunikacyjnych. Dalsza część referatu zostanie poświęcona temu drugiemu.

2 Interfejs MPI

2.1 Model sieciowy

Komunikacja odbywa sie za pomocą przesyłania wiadomości (czyli między innymi w standardzie MPI)w tak zwanym modelu sieciowym. Składa się on z określonej liczby procesorów, przy czym każdy z nich posiada własną pamięć lokalną. Procesory posiadają dostęp jedynie do instrukcji i danych przechowywanych w swojej pamięci lokalnej – nie istnieje pamięć wspólna. Aby umożliwić wymianę informacji pomiędzy procesorami, tworzona jest sieć połączeń (ang *interconnection network*), która zbudowana jest z dwukierunkowych kanałów komunikacyjnych (łącz).



Rysunek 1: Model sieciowy. Źródło: [4], str 94

Wymiana informacji między procesorami jest realizowana poprzez kooperujące ze sobą procedury trasowania (ang. *routing*), które działają w każdym procesorze. Dzięki nim, każdy węzeł sieci (tutaj: procesor) posiada informację, z którymi węzłami może wymieniać informacje. Zbiór wszystkich procedur trasowania definiuje **topologię sieci połączeń**. Można ją opisać przy pomocy grafu, gdzie wierzchołkami (węzłami) są procedory, natomiast krawędzi to dwukierunkowe łącza.

Ocenę skuteczność/przydatność danej sieci podczas prowadzenia obliczeń równoległych można określić biorąc pod uwagę kilka parametrów:

- **Średnica sieci** (ang *diameter*) – maksymalna odległość zmierzoną za pomocą liczby krawędzi między dowolnymi dwoma wierzchołkami. Im mniejsza średnica, tym lepsza jest sieć – oznacza to, że informacje będą potrzebowały średnio mniej czasu na dotarcie do właściwego odbiorcy. Przypadek pesymistyczny zakłada, że wiadomość będzie musiała zostać przesłana przez liczbę krawędzi równej średnicy.

- **Szerokość połowienia sieci** (ang. *bisection width*) – minimalna liczba krawędzi, którą należy usunąć z obecnej sieci, aby móc ją podzielić na 2 równe podsieci.

- **Szerokość pasma** (ang. *bisection bandwidth*) – jest to iloczyn szerokości połowienia oraz szybkości przesyłu danych w pojedynczym kanale. Pozwala określić liczbę bitów, jaką można przesłać między połówkami w jednostce czasu. Im większa szerokość pasma, tym lepiej.

- **Maksymalny stopień wierzchołka** – maksymalna liczba krawędzi połączonych z danym wierzchołkiem (liczona globalnie dla całej sieci). Dla niewielkiego stopnia łatwiej zaprogramować procedury komunikacyjne ze względu na fakt, że używają one mniejszej liczby kanałów. Zakłada się, że sieć jest dobra jeżeli przy wzroście liczby p procesorów średnica sieci rośnie nie szybciej niż logarytmicznie w funkcji p, natomiast maksymalny stopień wierzchołka jest stałą liczbą o małej wartości.

- **Spójność krawędziowa** (ang. *edge connectivity*) – definiowana jaka minimalna liczba krawędzi, które muszą zostać wyłączone z sieci aby ta stała się niespójna (graf rozłoży się na 2 lub więcej osobnych podgrafów). Im większa spójność krawędziowa, tym odporniejsza jest sieć – istnieje mniejsze prawdopodobieństwo całkowitego unieruchomienia sieci w przypadku, gdy któryś procesor ulegnie uszkodzeniu. Większa spójność prowadzi też do zmniejszenia rywalizacji poszczególnych węzłów o łącze.

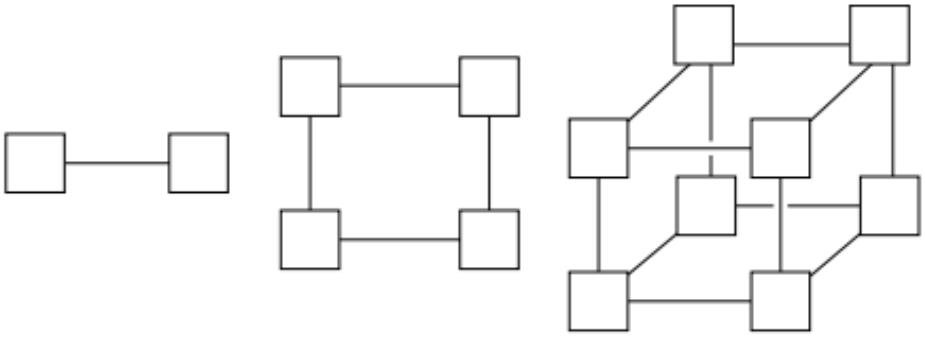
- **Koszt sieci** – zazwyczaj określana jako suma wszystkich kanałów w sieci.

Przykładowe topologie sieci połączeń:

- siatka

- torus (jedno-, wielowymiarowy)

- kostka (jedno-, wielowymiarowa)



Rysunek 2: Topologia kostki. Od lewej: jedno-, dwu- oraz trójwymiarowa. Źródło: [3], str 40

2.2 Zasada działania MPI

MPI jako interfejs przeznaczony do pracy z obliczeniami rozproszonymi oparty został o model sieciowy, posiadaja jednak kilka cech, które wyróżniają od standardowej implementacji tego wzorca. MPI można potraktować jako interfejs pomiędzy programem a systemem operacyjnym.

Tradycyjnie, każdy z procesów posiada własną pamięć lokalną, co narzuca konieczność komunikacji przez dwukierunkowe łącza kounikacyjne (w nomenklaturze MPI nazywane **komunikatorami**). Komunikacja polega na przesłaniu danych z pamięci procesu źródłowego do pamięci lokalnej procesu docelowego przy wykorzystaniu węzłów pośrednich. Domyślnie każdy nowoutworzony proces znajduje się w komunikatorze świat (MPI_COMM_WORLD). Takie rozwiązanie sprawia, że każdy proces może wymieniać dane z dowolnym pośród pozostałych, niezależnie od fizycznej struktury procesorów (jest ona przezroczysta dla procesów). Istnieje możliwość zdefiniowania własnych komunikatorów – co może okazać się przydatne w przypadku, gdy programista chce zawężyć zakres procesów, do których wysyłana jest wiadomość rozgłoszeniowa (ang. *broadcast*).

W trakcie tworzenia programu w oparciu o ten interfejs, warto podzielić część programu przeznaczoną do obliczeń rozproszonych na części, które mają zostać przydzielone do osobnych procesów. Na początku pracy, deklarowana jest ilość procesów, które mają zostać zaangażowane do pracy. Może się to odbywać na jeden z 2 sposobów:

- Statyczny – procesy tworzone są przed wykonaniem programu. Program (proces główny, tak zwany *root*) nie może zostać zakończony przed końcem pracy wszystkich pozostałych procesów.
- Dynamiczny – Potrzebne procesy są tworzone podczas pracy programu. Ta opcja jest dostępna wyłącznie dla wersji MPI-2 (zaprezentowanej w 1997 roku).

Każdy utworzony proces posiada własny unikalny identyfikator (id) w ramach komunikatora. W różnych komunikatorach ten sam proces może posiadać różne id. Identyfikatorem procesu głównego (*roota*) jest liczba 0.

W momencie tworzenia nowego procesu, tworzona jest kopia programu przeznaczonego tylko dla tego procesu. W praktyce oznacza to, że posiada dostęp do każdej zmiennej zadeklarowanej globalnie, jednak tylko w ramach lokalnej kopii. W przypadku, gdy proces potrzebuje danych znajdujących się w innym węźle, konieczna jest wymiana informacji w ramach komunikatora. Do rozdzielania pracy stosuje się standardowe operacje rozgałęzienia (między innymi instrukcje *if* czy *else* w językach C/C++) identyfikując id procesu. Jest to technika zwana SPMD (*Single Program Multiple Data* – pojedynczy program, wiele danych), będący subkategorią MIMD (*Multiple Instruction Multiple Data* - wiele instrukcji, wiele danych), znanej z taksonomii Flynna. Zazwyczaj utworzone kopie programów działają w sposób asynchroniczny, lecz może dojść do sytuacji, w której procesy te będą działały synchronicznie. Określony sposób działania może być uzależniony od funkcji, jakie zostaną użyte przez programistę.

2.3 Kompilacja i uruchamianie

Szczegóły związane z kompilacją i uruchamianiem programu napisanego przy użyciu biblioteki MPI są zależne od używanego systemu operacyjnego. Większość z nich do kompilacji używa komendy, której można użyć z poziomu linii poleceń/terminala:

```
$ mpicc -g -Wall -o <plik_źródłowy> <plik_wynikowy>.c
```

Zazwyczaj `mpicc` jest skryptem opakowującym (ang. *wrapper script*) dla kompilatora języka (dla powyższego przypadku, języka C). Skrypt opakowujący jest pisany w celu uruchomienia określonego programu. Skrypt ten upraszcza uruchomienie kompilatora poprzez jawne wskazanie, w którym miejscu znajdują się potrzebne pliki nagłówkowe oraz które biblioteki nalezy połączyć z plikiem obiektu.

Uruchamianie skompilowanego programu odbywa się przez następującą komendę:

```
mpirun -np <liczba_procesów> <nazwa_pliku_wynikowego> <parametry>
```

W wyżej wymienionej komendzie ¡liczba_procesów¿ jest liczbą całkowitą dodatnią i wskazuje, ile procesów ma być wykonanych równolegle. ¡nazwa_pliku_wynikowego¿ oraz ¡parametry¿ przekazywane są do procesów za pośrednictwem zmiennych `argc` i `argv` znajdujących się w nagłówku funkcji main (zgodnie z zasadami języka C).

2.4 Inicjalizacja i kończenie programu

Większość elementów składowych programu napisanego przy użyciu MPI jest instrukcjami natywnymi używanego języka (C, C++, Ada, Fortran). Aby umożliwić korzystanie z instrukcji nowej biblioteki, należy dodać następującą instrukcję (w języku C):

```
include "mpi.h"
```

Spowoduje to włączenie pliku nagłówkowego `mpi.h`. Znajdują się w nim prototypy funkcji MPI, makrodefinicje, definicje typów oraz inne definicje i deklaracje potrzebne do skompilowania programu MPI. Pierwszą instrukcją, jaka jest wykonywana przed rozdzieleniem pracy pomiędzy wątki jest `MPI_INIT` o następującej składni:

```
int MPI_Init(
        int* argc_p
        char*** argv_p)
```

Argumenty funkcji są wskaźnikami do argumentów funkcji main, kolejno `argc` i `argv`. W przypadku, gdy parametry wywołania nie istnieją lub nie są potrzebne dla instrukcji MPI, można do obu przekazać wartość NULL. Wartością zwracaną przez `MPI_Init` jest kod błedu, co jest standardem dla większości funkcji tej biblioteki. Jeżeli zwrócona wartość jest równa `MPI_SUCCESS`, oznacza to poprawne wykonanie inicjalizacji. Pozostałe kody oznaczają błędy jakie wystąpiły podczas pracy, a ich wartości uzależnione są od implementacji bilioteki. Informację o tym, czy w danym momencie programu mechanizm MPI został zainicjalizowany, możemy uzyskać za pomocą funkcji `MPI_Initialized`. Rzadziej używaną alternatywe dla `MPI_Init` stanowi `MPI_Init_thread`, który dodatkowo inicjalizuje środowisko wątków.

Analogicznie, ostatnią instrukcją, jaka powinna zostać wywołana w programie MPI, jest funkcja finalizacji:

```
int MPI_Finalize(void);
```

Jej wywołanie powoduje zwolnienie wszystkich zasobów komputera, które zostały wcześniej zaalokowane przez funkcję inicjalizacji, a następnie wykorzystywane w trakcie obliczeń równoległych. Podobnie jak wcześniejsza funkcja, MPI_Finalize zwraca kod błędu.

Nieobowiązkowymi, ale niemal równie ważnymi funkcjami są `MPI_Comm_size` oraz `MPI_Comm_rank` o następującej składni:

```
int MPI_Comm_size(
        MPI_Comm_comm,
        int* comm_s z_p);

int MPI_Comm_rank(
        MPI_Comm_comm,
        int* my_rank_p);
```

W obu przypadkach, pierwszym argumentem jest komunikator, w którym znajduje się proces go wywołujący. Komunikatory w bibliotece MPI są nieprzeźroczystym obiektem (tzn. o nieznanej wewnętrznej strukturze), w ramach której kolekcja procesów może wymieniać między sobą dane. Posiadają one własny typ, `MPI_Comm`. `MPI_Comm_size` jako swój drugi argument zwraca liczbę procesów znajdującą się w komuikatorze, natomiast `MPI_Comm_rank` informuje jaki identyfikator został przydzielony procesowi który wykonał tę funkcję w ramach komunikatora. Funkcje te ułatwiają rozdzielanie zadać pomiędzy procesy oraz kontrolę nad przepływem pracy algorytmu równoległego.

2.5 Typy danych w MPI

Interfejs MPI wykorzystuje własne typy danych w trakcie wymiany informacji. Zamiast informacji o ilości przesyłanych bajtów, w trakcie transfery wysyłana jest informacja o ilości przesyłanych elementów danego typu (argument `count`). Wartość ta może być równa zero, co jest tożsame z informacją, że część wiadomości zawierająca dane jest pusta. Podstawowe typy danych MPI, których można użyć w trakcie

komunikacji odpowiadają typom podstawowym języka programowania, z którego korzystamy i różnią się w zależności od implementacji.

Typ MPI	Odpowiednik w języku C
MPI_CHAR	char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG_INT	signed long long int
MPI_SIGNED_CHAR	signed char
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED	unsigned short int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_C_BOOL	_Bool
MPI_BYTE	8 bitów (8 cyfr binarnych)
MPI_PACKED	-

Rysunek 3: Najważniejsze typy MPI i ich odpowiedniki dla języka C Źródło: [2], str 26

3 Komunikacja między procesami w biliotece MPI

3.1 Komunikacja punkt-punkt

3.1.1 MPI_Send

Ten rodzaj komunikacji stanowi jeden z dwóch podstawowych odmian transferu danych w modelu sieciowym. Polega on na przesyłaniu wiadomości pomiędzy określona parą procesów. W bibliotece MPI służy do tego para funkcji `MPI_Send` oraz `MPI_Receive`. Składnia nagłówka pierwszej z nich prezentuje się następująco:

```
int MPI_Send(
        void* message_buff,
        int message_size,
        MPI_Datatype message_type/,
        int dest,
        int tag ,
        MPI_Comm communicator);
```

Gdzie:

- `message_buff` – pierwszy argument funkcji będący wskaźnikiem na bufor, w którym przetrzymywana jest wiadomość wysyłana przez proces. Jest to adres (w pamięci przeznaczonej na zmienne), pod którym zapisana jest wiadomość przed wysłaniem.
- `message_size` – określa, ile elementów ma zostać przesłanych w wiadomości
- `message_type` – typ wysyłanych danych. Stanowi jeden z wbudowanych typów danych dla MPI Rozdział 2.5 lub typ pochodny zdefiniowany przez użytkownika.
- `dest` – id procesu, do którego wiadomość ma zostać przesłana.
- `tag` – liczba nieujemna z zakresu 0 do 32767 (lub więcej, zależnie od implementacji). Informacja, mająca na celu ułatwienie rozróżniania procesów. Umożliwia to wysyłanie dodatkowej informacji do innych procesów.

- `communicator` – nazwa komunikatora, w ramach którego wysyłana jest wiadomość.

Wszystkie wymienione parametry są parametrami wejściowymi. Działanie `MPI_Send` może być uzależnione od implementacji. Rozróżnia się 2 sposoby komunikacji:

- Komunikacja synchroniczna – polega na wstrzymaniu obliczeń przez proces wysyłający do momentu, gdy proces odbierający zgłosi gotowość do przyjęcia wiadomości. Dopiero po zgłoszeniu, informacja zostaje wysłana i procesy kontynują pracę.
- Komunikacja buforowana – po wywołaniu funkcji `MPI_Send` wiadomość kopiowana jest do bufora, a proces wysyłający kontynuuje pracę. Proces odbierający może pobrać wiadomość z bufora w dowolnym momencie, o ile jest ona aktualnie dostępna. Ten sposób przyspiesza działanie programu w porównaniu do poprzednika, ale wymaga obecności buforowania w systemie obliczeniowym.

3.1.2 MPI_Receive

Funkcją służącą do odbierania wiadomości wysłanych przy pomocy `MPI_Send` jest `MPI_Receive` o nagłówku:

```
int MPI_Recv(
        void* message_buff,
        int message_size,
        MPI Datatype message_type,
        int source,
        int tag,
        MPI_Comm communicator,
        MPI_Status* status);
```

Gdzie:

- `message_buff` wskazuje na obszar pamięci, do którego zapisane zostaną otrzymane dane.
- `message_size` – liczba danych, z których ma się składać wczytana wiadomość
- `message_type` – typ wysyłanych danych. Stanowi jeden z wbudowanych typów danych dla MPI Rozdział 2.5 lub typ pochodny zdefiniowany przez użytkownika. Należy zwrócić szczególną uwagę, aby w obszarze pamięci, do którego będzie zapisywana wiadomość, istniała odpowiednia ilość wolnego miejsca.
- `source` – identyfikator procesu, od którego odbierana będzie wiadomość.
- `tag` – liczba nieujemna z zakresu 0 do 32767 (lub więcej, zależnie od implementacji). Informacja, mająca na celu ułatwienie rozróżniania procesów. Umożliwia to wysyłanie dodatkowej informacji do innych procesów.
- `communicator` – nazwa komunikatora, w ramach którego wysyłana jest wiadomość.
- `status` – wskaźnik do struktury przechowujacej informacje o odebranej wiadomości. Struktura taka zawiera m. in. dane o numerze (id) nadawcy, znaczniku oraz o kodzie błędu.

Większość parametrów ma charakter wejściowy, za wyjątkiem `message_buff` oraz `status`, które mają charakter wyjściowy. Funkcja MPI_Receive jest funkcją blokującą, co oznacza że proces w momencie jej wywołania jest blokowany do momentu odebrania wiadomości.

3.1.3 Dodatkowe funkcje i struktury

Dla zwiększenia elastyczności (oraz przyspieszenia wykonania programów równoległych) w bibliotece MPI zaimplementowano dodatkowe elementy:

- `MPI_ISend` – specyficzna odmiana funkcji `MPI_Send`. W przeciwieństwie do poprzedniczki nie blokuje procesu wysyłającego, niezależnie od obecności buforowania w systemie. Wiadomość jest wysyłana, a sterowanie natychmiast oddawane jest do procesu nadawcy, pozwalając na kontynuowanie obliczeń. Proces ten powinien po jakimś czasie wywołać funkcję `MPI_Wait` w celu sprawdzenia, czy wysyłanie zostało zakończone.

- `MPI_IReceive` – nieblokująca odmiana `MPI_Receive`. Jej wywołanie inicjuje odbieranie wiadomości, po czym sterowanie wraca do procesu wywołującego, który kontunuuje obliczenia. Podobnie jak wyżej, można wywołać funkcję `MPI_Wait` (lub alternatywnie `MPI_Test` sprawdzającą, czy wiadomość została odebrana.

- `MPI_ANY_SOURCE` – zezwala na przyjęcie dowolnego typu danych ze źródła.

- `MPI_ANY_TAG` – pozwala na przyjęcie dowolnego znacznika ze źródła.

- `MPI_Probe` – funkcja pozwalająca na sprawdzenie przychodzącej wiadomości bez jej właściwego odebrania. Umożliwia uniknięcie sytuacji, w której typ oraz identyfikator nadawcy są zgodne, jednak wiadomość jest za duża i trzeba zaalokować dodatkową pamięć przed jej odebraniem.

3.2 Komunikacja kolektywna oraz redukcja danych

Komunikacja typu punkt-punkt pozwala precyzyjnie wskazać, jak ma odbywać się transfer wiadomości między procesami, nie jest jednak pozbawiony wad. W przypadku, gdy istnieje potrzeba zaangażowania dużej ilości procesów (gdzie dla nowoczesnych superkomputerów wartość ta może figurować na poziomie dziesiątek lub setek tysięcy) zaprogramowanie każdej pary procesorów między którymi ma odbywać się komunikacja byłoby niezwykle żmudnym zajęciem. W tym celu istnieje **komunikacja kolektywna**, w ramach której wszystkie procesy wykonują tą samą funkcję komunikacyjną. Przykładem komunikacji kolektywnej jest operacja rozgłaszania ("jeden do wszystkich"). Podstawowe funkcje, które implementują ją w biliotece MPI to `MPI_Bcast` oraz `MPI_Reduce`.

3.2.1 MPI_Bcast

Nagłówek funkcji `MPI_Bcast` prezentuje się następująco:

```
int MPI_Bcast (
        void *buffer,
        int count,
        MPI_Datatype type,
        int root,
        MPI_Comm comm);
```

Gdzie:

- `buffer` – adres początkowy miejsca w pamięci, gdzie przetrzymywana jest wiadomość do wysłania.

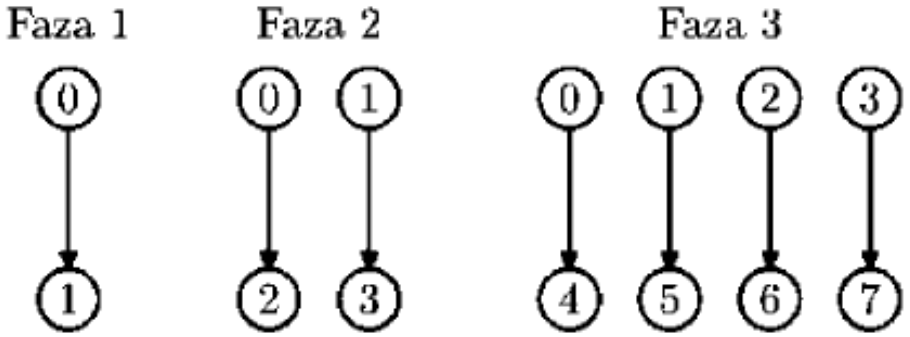
- `message_count` – określa, ile elementów ma zostać przesłanych w wiadomości

- `message_type` – typ wysyłanych danych. Stanowi jeden z wbudowanych typów danych dla MPI Rozdział 2.5 lub typ pochodny zdefiniowany przez użytkownika.

- `root` – identyfikator procesu, który dokonuje rozgłoszenia wiadomości.

- `comm` – komunikator w ramach którego wykonywane jest rozgłoszenie.

Użycie tej funkcji sprawia, że proces o id równemu arguemntowi `root` wysyła dane `buffer` do wszystkich pozostałych procesów w ramach komunikatora `comm`. Wszyskie procesy, które biorą udział w rozgłaszaniu muszą posiadać identyczne parametry. Charakter parametrów jest zależny od wykonawcy: dla procesu `root` parametry mają charakter wejściowy, a we wszystkich innych – wyjściowy. Ponieważ rodzaj komunikacji znacząco różni się od punkt-punkt, funkcja ta jest niekompatybilna z funkcjami `MPI_Send` oraz `MPI_Receive`. Do odebrania informacji wysłanych przez rozgłaszanie, konieczne jest zastosowanie operacji redukcji. W przypadku, gdy któryś z procesów uczestniczących w rozgłaszaniu nie wykona wywołania tej funkcji, istnieje ryzyko zablokowania pracy w ramach komunikatora.



Rysunek 4: Schemat rozgłaszania dla 8 procesów. Źródło: [1], str 180

3.2.2 MPI_Reduce

Funkcja redukcji również należy do komunikacji kolektywnej i stanowi przeciwieństwo rozgłaszania. Jest to operacja typu "wszystkie do jednego" (ang. *all-to-one*. Nagłówek tej funkcji ma następującą postać:

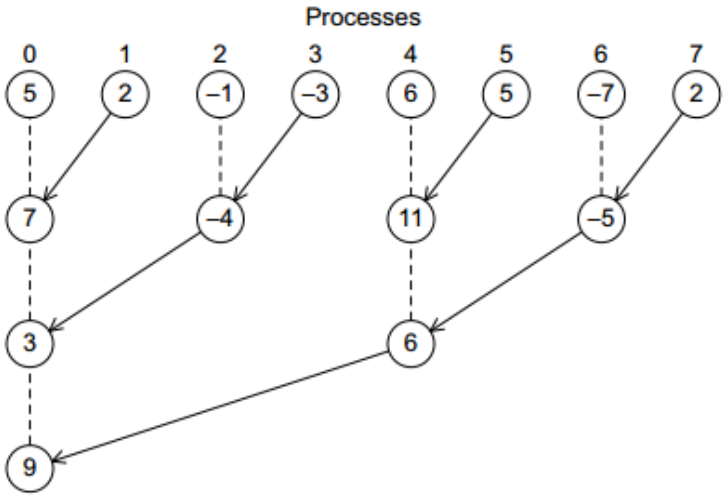
```
int MPI_Reduce(
        void* collected_data,
        void* result_data,
        int count,
        MPI_Datatype type,
        MPI_Op operator,
        int dest,
        MPI_Comm comm);
```

Gdzie:

- `collected_data` – adres pierwszej danej, na której zostanie wykonana operacja redukcji we wszystkich procesach w ramach komunikatora.
- `result_data` – adres pamięci, pod jaki zostanie zapisany wynik redukcji.
- `count` – liczba danych, na których zostani wykonana operacja redukcji.
- `type` – typ odbieranych i liczonych danych. Stanowi jeden z wbudowanych typów danych dla MPI Rozdział 2.5 lub typ pochodny zdefiniowany przez użytkownika.
- `operator` – wskazuje, jaki rodzaj operacji ma zostać wykonany na odebranych danych.
- `dest` – identyfikator wskazujący, który proces ma odebrać i przetworzyć dane.
- `comm` – komunikator, w ramach którego wykonana zostanie operacja redukcji

Operacja redukcji polega na zredukowaniu danych lokalnych przechowywanych w procesach do jednej danej, charakteryzującej lokalnie zbiór danych. Po jej wykonaniu, wynik zostaje zapisany w procesie korzeniu. Niemal wszystkie parametry w funkcji `MPI_Reduce` mają charakter wejściowy zarówo dla korzenia odbierającego dane oraz procesów wysyłających. Wyjątkiem jest parametr przechowujący informację o wyniku, mający charakter wyjściowy.

W tabeli 6 przedstawiono dostępne operacje, które można wstawić w miejsce argumentu `operator`. Istnieje możliwość zadefiniowania własnej operacji redukcji. Większość standardowych operacji redukowana jest do pojedynczje wartości. Wyjątek od reguły stanowią `MPI_MAXLOC` oraz `MPI_MINLOC`, gdzie zarówno dane wyjściowe jak i wyjściowe stanowią parę wielkości. Parę wejściową tworzą dana podlegająca redukcji, a także identyfikator procesu, w którym ta dana się znajduje. Analogicznie, parę wyjściową stanowi dana wynikowa (maksimum lub minimum pośród danych lokalnych wszystkich węzłów) oraz numer jej procesu.



Rysunek 5: Przykładowy schemat redukcji sumy dla 8 procesów. Źródło: [3], str 102

Nazwa	Znaczenie	Typ argumentów
MPI_MAX	maksimum	całkowity oraz rzeczywisty
MPI_MIN	minimum	całkowity oraz rzeczywisty
MPI_SUM	suma	całkowity oraz rzeczywisty
MPI_PROD	iloczyn	całkowity oraz rzeczywisty
MPI_LAND	logiczne and	całkowity
MPI_LOR	logiczne or	całkowity
MPI_LXOR	logiczne exclusive or	całkowity
MPI_BAND	bitowe and	całkowity oraz byte
MPI_BOR	bitowe or	całkowity oraz byte
MPI_BXOR	bitowe exclusive or	całkowity oraz byte
MPI_MAXLOC	maksmum i jego lokalizacja	pary wielkości
MPI_MINLOC	minimum i jego lokalizacja	pary wielkości

Rysunek 6: Dostępne operacje dla funkcji MPI_Reduce Źródło: [2], str 176

3.2.3 Dodatkowe funkcje i struktury

Podobnie jak przypadku komunikacji typu punkt-punkt, do dyspozycji programisty zostały oddane inne funkcje umożliwiające komunikację kolektywną.

`MPI_Allreduce` stanowi alternatywną wersję `MPI_Reduce`. Posiada prawie identyczną listę argumentów, a jedyną różnicą jest brak argumentu numeru procesu docelowego. Wynika z założenia działania funkcji – po dokonaniu redukcji, wynik działania wysyłany jest do wszystkich procesów w ramach komunikatora.

`MPI_Scatter` rozdziela dane zapisane w ciągłej struturze danych (np. tablicy) oraz przesyła je z pojedynczego procesu źródłowego do pozostałych procesów w ramach komunikatora. Proces źródłowy (korzeń) dzieli wektor danych na p (liczba procesów w komunikatorze biorących udział w obliczeniach) elementów i rozsyła je do pozostałych procesów (włącznie ze sobą samym). W wyniku takiej operacji, każdy proces otrzyma fragment danych do obliczeń. `MPI_Scatter` wymaga od wszystkich procesów wcześniejszego zainicjalizowania dwóch tablic: jednej przechowującej adres wysyłanych dancyh oraz drugiej przechowującej adres pobranych danych

`MPI_Gather` jest funkcją o działaniu odwrotnym do `MPI_Scatter`. Jej wywołanie spowoduje zebranie danych ze wszystkich (w ramach komunikatora) przez procesor zwany korzeniem. Dane umieszczane są pamięci podręcznej korzenia wskazanej jednym z argumentów. Wszystkie pobrane porcje danych powinny być tej samej wielkości, o co dba wcześniej wywoływan funkcja `MPI_Scatter`.

`MPI_Comm_split` jest funkcją wywoływaną przez wszystkie procesy w ramach komunikatora. Efektem wywołania jest podział starego komunikatora na nowy według określonego klucza. Zadaniem funkcji jest ułatwienie grupowania procesów.

4 Podsumowanie

Wczesne wprowadzenie ujednoliconego standardu, jakim jest MPI przysporzyło mu popularności na tle rosnącego zapotrzebowania i rozwoju na obliczenia równoległe. Obszerna dokumentacja oraz mnogość implementacji zarówno pod kątem języków programowania jak i architektury procesorów sprawiła, że MPI stał się często stosowanym rozwiązaniem w systemach rozproszonych. Dodatkowymi zaletami przemiawiającymi za biblioteką jest hermetyczny interfejs programistyczny, sprawiający że program uruchomiony na różnych maszynach działa identycznie. Biblioteka MPI udostępnia kilka odmiennych sposobów komunikacji, pozostawiając osobie piszącej kod prawo wyboru, którą wersję zaimplementować. W mniejszych systemach (np. komputer z pojedynczym procesorem wielordzeniowym) zazwyczaj wygodniejszym sposobem jest komunikacja typu punk-punkt przy pomocy funkcji MPI_Receive i MPI_Send — zarządzanie na poziomie pojedynczego kanału komunikacyjnego ułatwia modyfikację algorytmu. W przypadku dużej liczby procesów, wygodniejszym sposobem jest komunikacja kolektywna.

Pomimo dużej elastyczności udostepnianej przez interfejs MPI, należy pamiętać, aby w miarę możliwości komunikacji między procesami. Z każdą wymianą wiadomości związany jest narzut czasowy, zazwyczaj znacznie większy od operacji podstawowych. W przypadku gdy inicjalizowana jest wymiana informacji ze wszystkimi procesami przy użyciu funkcji `MPI_Send`, proces źródłowy musi wykonać p - 1 instrukcji, gdzie p oznacza całkowitą liczbę procesów (należy pamiętać, że w przypadku dużych systemów może to oznaczać tysiące czasochłonnych operacji). Złożoność obliczeniowa takiego przedsięwzięcia jest rzędu $\mathcal{O}(p)$ (przy pominięciu n liczby danych i operacji na nich wykonywanych). W praktyce to oznacza, że wraz ze wzrostem liczby dostępnych procesów p program będzie przyspieszał do pewnej wartości granicznej P, po której znacznie zwalniać (narzut czasowy związany z komunikacja zacznie przeważać nad przyspieszeniem związanym ze zrównolegleniem programu). Dużo lepszym sposobem jest wykorzystanie komunikacji przez rozgłaszanie, w którym udział biorą wszystkie procesy przez cały czas trwania wymiany wiadomości. Nie eliminuje on całkowicie problemu związanego z rosnącym narzutem czasowym przy zwiększonej liczbie procesorów, ale sprawia że jest on mniejszy (rzędu $\mathcal{O}(logp)$). Alternatywną opcją służącą zmniejszającą narzut czasowy komunikacji jest zastosowanie obliczeń nadmiarowych (zwanych też redundantnymi). Są to operacje powtarzające się w wielu procesach, a dające ten sam wynik – czasami opłaca się wielokrotnie policzyć tę samą wartość w różnych procesach zamiast wykonać obliczenia w pojedynczym korzeniu, a następnie rozesłać do reszty (zwłaszcza, gdy są to nieskomplikowane obliczenia).

Literatura

[1] Z. J. Czech, *Wprowadzenie do obliczeń równoległych*, Wydawnictwo PWN, Warszawa 2013, wyd. 2

[2] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard, Version 3.0*, High Performance Computing Center Stuttgart (HLRS), Stuttgart 2012

[3] P. Pacheco, *An Introduction to Parallel Programming*, Morgan Kaufmann, San Francisco 2001

[4] M. J. Quinn, *Parallel Programming in C with MPI and OpenMP*, McGraw-Hill, Nowy Jork 2003