Podstawy Programowania w MPI

Prelegent: Jakub Gałecki

ICM, 21.09.2023

Sylwetka prowadzącego:

- Członek Zespołu Oprogramowania i Wsparcia Użytkowników ICM
- Autor biblioteki MES, ekstensywnie wykorzystującej MPI
- Jeden z maintainerów stosu aplikacji ICM

Materiały:

- Standard MPI
- Dokumentacja OpenMPI
- William Gropp, Ewing Lusk, and Anthony Skjellum. 2014. Using MPI: Portable Parallel Programming with the Message-Passing Interface. The MIT Press.

Agenda

Kontekst – rodzaje równoległości

Czym jest MPI?

Podstawowe definicje

Anatomia wiadomości

Komunikacja point-to-point

Komunikacja grupowa

Komunikacja nieblokująca

MPI “od kuchni”

Sesja praktyczna

Wstęp

Rodzaje równoległości

Rodzaj równoległości	Jak go osiągnąć?	Limit
Instrukcje SIMD	Kompilator, funkcje wbudowane	Szerokość jednostki wektorowej
Instruction-Level Parallelism	Kompilator	Liczba jednostek wykonania
SMT (hyper-threading)	Kod, OS, IPC	Liczba rdzeni logicznych (2)
Wielowątkowość	Kod, OS, IPC	Liczba rdzeni fizycznych
Wiele procesorów (ccNUMA)	-	Liczba socketów
Wiele maszyn	**Komunikacja sieciowa (MPI)**	**Dostępne zasoby**

SIMD – Single Instruction, Multiple Data

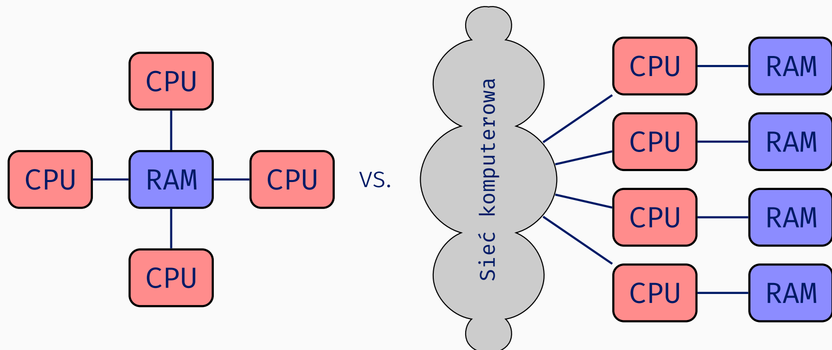
SMT – Simultaneous Multi-Threading

OS – Operating System

IPC – Inter-process communication

ccNUMA – cache-coherent Non-Uniform Memory Access

Pamięć współdzielona vs. rozproszona



Pamięć współdzielona vs. rozproszona

$\{\text{wątek}\} \subseteq \{\text{proces}\} \subset \text{system}$

Pamięć współdzielona:

- Wątki współdzielą przestrzeń adresów – brak kosztu współdzielenia danych (poza inherentnym zjawiskami hardware'owymi)
- Pułapki synchronizacji
- Wiele modeli/bibliotek: OpenMP, TBB, pthreads, ...
- Ograniczona skalowalność

Pamięć rozproszona:

- Procesy mają rozdzielne przestrzenie adresów – każda wymiana danych wymaga komunikacji
- W HPC jedna biblioteka: MPI
- Skalowalność ograniczona jedynie rozmiarem klastra

Message **P**assing **I**nterface

Standard ustalany przez MPI-Forum

Natywny interfejs dla Fortrana, C, ~~C++~~

Obecnie wersja 4.0 (2021):

`https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf`

Konkurujące implementacje:

- MPICH – `https://www.mpich.org/`
- OpenMPI – `http://www.open-mpi.org/`

Fundament stosu HPC

Model programowania MPI

Model procesowy SPMD – Single Program Multiple Data:

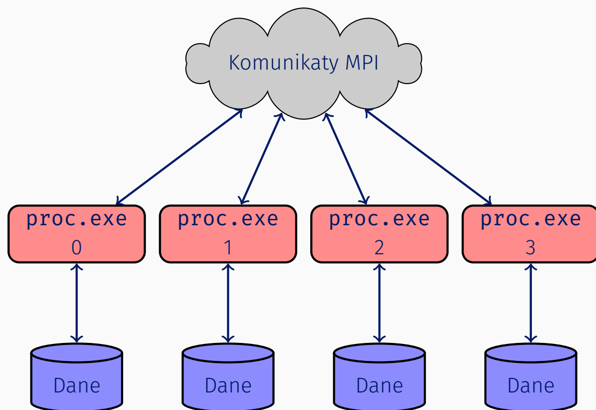
- Uruchamiamy launcher, podając mu nazwę właściwego programu, liczbę procesów n, inne opcje…
- Launcher uruchamia n podprocesów wykonujących podany program
- Podprocesy wykonują pracę, wymieniając komunikaty, za których “doręczenie” odpowiedzialny jest proces nadrzędny

Model MPMD również wspierany, lecz mniej popularny.

Założenia:

- Każdy proces ma swoją prywatną przestrzeń adresową, która może (lecz nie musi) pokrywać się z pamięcią fizyczną innych procesów
- Każda wymiana informacji jest jawna

Model SPMD



Podstawowe pojęcia

Komunikator – abstrakcja opisująca zbiór procesów mogących wymieniać między sobą komunikaty oraz kontekst komunikacji

Rząd – unikatowy numer procesu w ramach danego komunikatora. Numeracja rozpoczyna się od 0 i zawiera kolejne liczby całkowite.

Domyślnie tworzone są 2 komunikatory:

- `MPI_COMM_SELF` – komunikator zawierający jedynie lokalny proces
- `MPI_COMM_WORLD` – komunikator zawierający wszystkie procesy

Możemy tworzyć nowe komunikatory, np. w celu wydzielenia osobnych kontekstów komunikacji

Inicjalizacja/finalizacja

Deklaracje MPI zawarte są w nagłówku `mpi.h`

Aby zainicjalizować MPI musimy najpierw wywołać funkcję:

```
int MPI_Init(int* argc, char*** argv);
```

Aby zakończyć sesję MPI wołamy:

```
int MPI_Finalize();
```

Przed inicjalizacją oraz po finalizacji nie wolno nam wołać żadnych funkcji MPI (z nielicznymi wyjątkami).

Określanie rzędu i rozmiaru komunikatora

Rozmiar komunikatora określamy w następujący sposób:

```
int size;
int err_code = MPI_Comm_size(MPI_COMM_WORLD, &size);
```

Rząd procesu w ramach komunikatora określamy w następujący sposób:

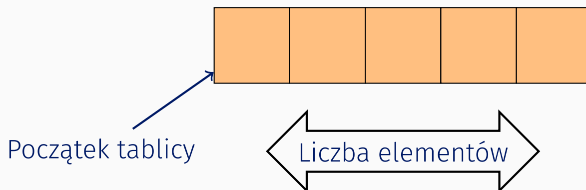
```
int my_rank;
int err_code = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

Komunikacja point-to-point

Komunikat

Komunikat zawiera:

- Wskaźnik do początku tablicy nadawanej/odbieranej
- Liczbę elementów znajdujących się w tablicy
- Typ elementów – `MPI_Datatype`



Typy danych

Typ danych jest w MPI określany dynamicznym uchwytem typu `MPI_Datatype`

Pozwala to na tworzenie własnych typów danych

Wbudowane typy całkowite i zmiennoprzecinkowe są domyślnie zdefiniowane, np.

- `char` – `MPI_CHAR`
- `int` – `MPI_INT`
- `double` – `MPI_DOUBLE`

Komunikat

Komunikat opisuje:

- Komunikator, w ramach którego wykonywana jest komunikacja
- Nadawca (rząd)
- Odbiorca (rząd)
- Etykieta (tag)

Aby komunikat został odebrany, jego opisy w procesie nadającym i przyjmującym muszą być **identyczne**

Wyjątki (wildcard) – `MPI_ANY_SOURCE`, `MPI_ANY_TAG`

Komunikacja point-to-point

Nadawca:

```
int MPI_Send(const void* buf, int size, MPI_Datatype type,
             int dest, int tag, MPI_Comm comm);
```

```
const double buf[3] = {3.14, 42., 2.71};
const int dest = 1, tag = 0;
int err = MPI_Send(buf, 3, MPI_DOUBLE, dest, tag,
                   MPI_COMM_WORLD);
```

Odbiorca:

```
int MPI_Recv(void* buf, int size, MPI_Datatype type, int src,
             int tag, MPI_Comm comm, MPI_Status* status);
```

```
double buf[3];
const int src = 0, tag = 0;
int err = MPI_Recv(buf, 3, MPI_DOUBLE, src, tag,
                   MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

MPI_Status, MPI_Probe

Ostatni argument typu MPI_Status pozwala na przekazanie dodatkowych informacji, np. ile elementów zostało faktycznie wysłanych.

Podobną rolę pełni funkcja MPI_Probe, pozwalająca na inspekcję przychodzącej komunikacji przed przystąpieniem do właściwego jej odebrania:

```
MPI_Probe(int source, int tag, MPI_Comm comm,
          MPI_Status* status);
```

Co dokładnie robi MPI z moją wiadomością?

Fundamentalnie 2 tryby wysyłania:

- Synchroniczne – dane wysyłane są bezpośrednio z podanego bufora
- Buforowane – dane są najpierw kopiowane do tymczasowego bufora

Tryb	Funkcja MPI	Zachowanie
Synchroniczny	`MPI_Ssend`	Pełna synchronizacja
Buforowany	`MPI_Bsend`	Buforowanie (wymaga konfiguracji bufora)
Standardowy	`MPI_Send`	Implementacja decyduje
Ready send	`MPI_Rsend`	Musi nastąpić po rozpoczęciu odbierania

Uwaga: Wszystkie ww. tryby są **blokujące**, tzn. wykonanie funkcji wysyłającej zakończy się dopiero gdy podany bufor z wiadomością nie będzie już potrzebny.

Komunikacja grupowa

Komunikacja grupowa

Komunikacja grupowa (kolektywna) – komunikacja, w której biorą udział wszystkie procesy w danym komunikatorze

Opisuje powtarzalny schemat wymiany danych w grupie procesów

Następujące typy:

- Bariera
- Broadcast
- Gather + AllGather
- Scatter
- All-to-all
- Reduce + AllReduce
- Scan

Prefiks `All` sugeruje, że po operacji następuje broadcast

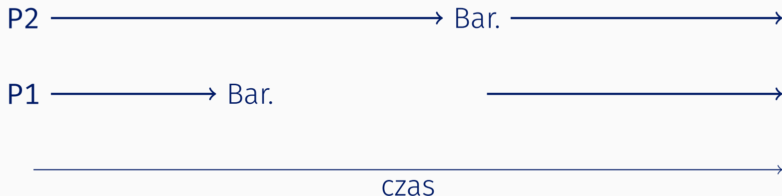
Bariera

Blokuje wykonanie programu dopóki wszystkie procesy w komunikatorze jej nie osiągną

Preferujemy synchronizację poprzez komunikaty

Służy m.in. do debugowania, profilowania, synchronizacji I/O

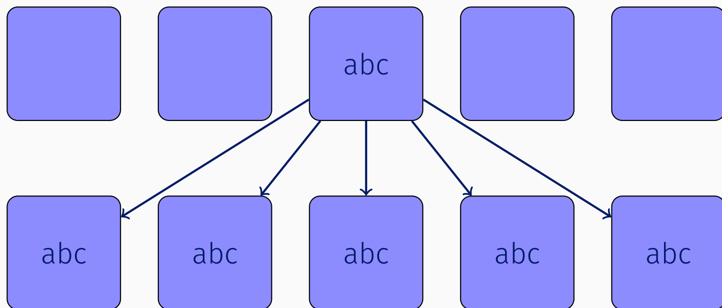
```
int MPI_Barrier(MPI_Comm comm);
```



Broadcast

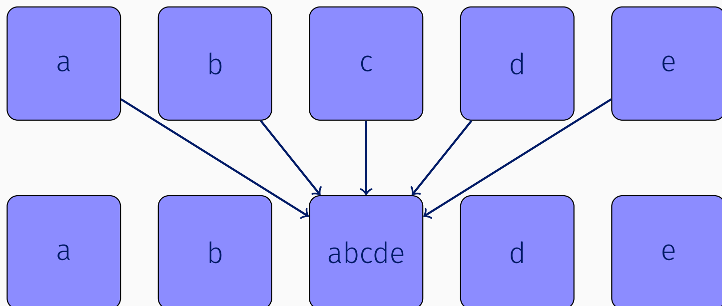
Broadcast – nadajemy dane z jednego procesu do wszystkich pozostałych

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



Gather – zbieramy dane z wielu procesów do jednego procesu

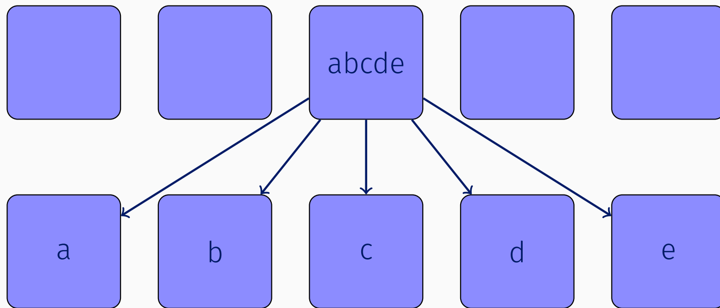
```
int MPI_Gather(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
               void* recvbuf, int recvcount, MPI_Datatype recvtype,
               int root, MPI_Comm comm);
```



Scatter

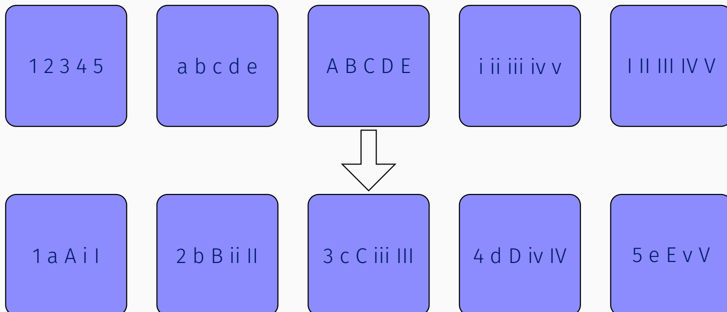
Scatter – Rozdzielamy dane z jednego procesu na wiele procesów

```
int MPI_Scatter(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
                void* recvbuf, int recvcount, MPI_Datatype recvtype,
                int root, MPI_Comm comm);
```



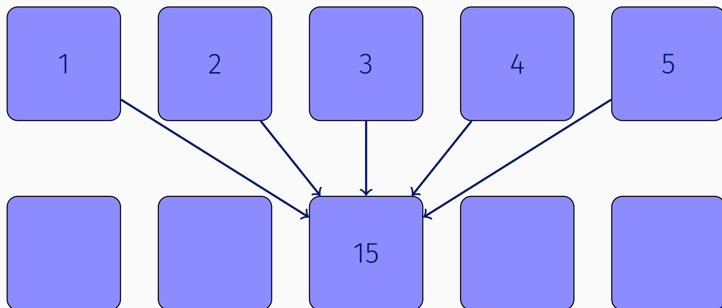
All-to-all – Wszyscy do wszystkich, transpozycja

```
int MPI_Alltoall(const void* sendbuf, int sendcount,
                 MPI_Datatype sendtype, void* recvbuf, int recvcount,
                 MPI_Datatype recvtype, MPI_Comm comm);
```



Reduce – Globalna redukcja (suma, max, etc.)

```
int MPI_Reduce(const void* sendbuf, void* recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op, int root,
               MPI_Comm comm);
```



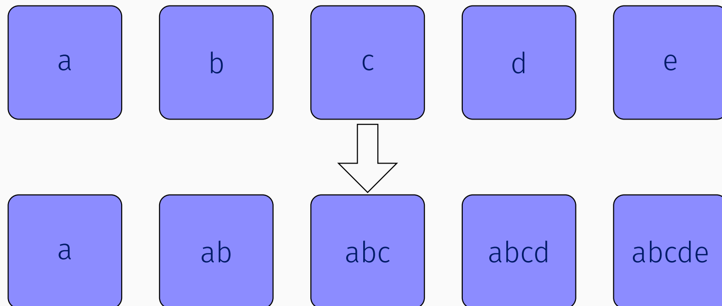
Typy redukcji

Możliwość zdefiniowania własnej redukcji, domyślnie mamy dostęp do:

Nazwa	Znaczenie
MPI_MAX	maksimum
MPI_MIN	minimum
MPI_SUM	suma
MPI_PROD	iloczyn
MPI_LAND	koniunkcja logiczna
MPI_BAND	koniunkcja bitowa
MPI_LOR	
MPI_BOR	
MPI_LXOR	
MPI_BXOR	
MPI_MAXLOC	max. wartość i lokalizacja
MPI_MINLOC	min. wartość i lokalizacja

Scan – Skan (suma prefiksowa) względem wartości w procesach

```
int MPI_Scan(const void *sendbuf, void *recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);
```



Komunikacja nieblokująca

Komunikacja nieblokująca

Komunikacja jest zlecana, kontrola zwracana jest natychmiast do punktu wywołania, komunikacja odbywa się w tle

Testowanie – sprawdzanie, czy dany komunikat się zakończył

Czekanie – blokowanie do czasu zakończenia komunikacji

Zarządzanie pamięcią na komunikaty w pełni po stronie użytkownika – wydajność, ale także potencjał błędów

Komunikacja nieblokująca jest **fundamentalna** dla wydajności programów – communication and computation overlap

Komunikacja nieblokująca

Wysyłanie:

```
int MPI_Isend(const void* buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm, MPI_Request* request);
```

Odbieranie:

```
int MPI_Irecv(void* buf, int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm, MPI_Request* request)
```

Dodatkowo wersje w różnych trybach, analogicznie jak dla komunikacji nieblokującej: `MPI_Ibsend`, `MPI_Issend`, etc.

Komunikacja zbiorowa także może odbywać się w sposób nieblokujący: `MPI_Ibcast`, `MPI_Iscatter`, etc.

Uwaga: bufory możemy zwalniać dopiero po upewnieniu się, że komunikacja się zakończyła!

MPI_Request

`MPI_Request` – abstrakcja reprezentująca uchwyt do żądanej komunikacji, jej instancje są zwracane przez funkcje do komunikacji asynchronicznej

Testowanie:

```
int MPI_Test(MPI_Request* request, int* flag, MPI_Status* status);
```

Czekanie:

```
int MPI_Wait(MPI_Request* request, MPI_Status* status);
```

Anulowanie*:

```
int MPI_Cancel(MPI_Request* request);
```

Utworzone `MPI_Request` musi zostać w jakiś sposób obsłużone, inaczej dojdzie do wycieku zasobów

MPI_Request

Dodatkowo dostępna jest funkcjonalność obsługująca grupy żądań

Testowanie:

```
int MPI_Testall(int count, MPI_Request array_of_requests[],
                int* flag, MPI_Status array_of_statuses[]);

int MPI_Testany(int count, MPI_Request array_of_requests[],
                int* index, int* flag, MPI_Status* status);

int MPI_Testsome(int incount, MPI_Request array_of_requests[],
                 int* outcount, int array_of_indices[],
                 MPI_Status array_of_statuses[]);
```

Czekanie:

```
int MPI_Waitall(int count, MPI_Request array_of_requests[],
                MPI_Status* array_of_statuses);

int MPI_Waitany(int count, MPI_Request array_of_requests[],
                int* index, MPI_Status* status);

int MPI_Waitsome(int incount, MPI_Request array_of_requests[],
                 int* outcount, int array_of_indices[],
                 MPI_Status array_of_statuses[]);
```

Kompilacja i uruchamianie programów MPI

Kompilacja

Implementacje dostarczają wrappery na kompilator, odpowiedzialne m. in. za ustawienie odpowiednich flag i linkowanie bibliotek dynamicznych:

- `mpifort` / `mpif77` / `mpif90`
- `mpicc`
- `mpicxx` / `mpic++`*

Wykorzystywany kompilator możemy podmienić dynamicznie ustawiając zmienną środowiskową, np. `OMPI_MPI<lang>`

Wsparcie ze strony CMake'a: `find_package(MPI)` z modułu `FindMPI`

Uruchamianie programów MPI

Launcher: `mpiexec` / `mpirun`

```
mpiexec -n 4 moj_super_program
```

Dostępne wiele opcji związanych z ustawianiem mapowania procesów na fizyczne zasoby sprzętowe

Na klastrach obliczeniowych (w tym maszynach ICM) korzystamy z systemów kolejkowych

Natywna integracja ze Slurmem

```
srun -n 4 moj_super_program
```

MPI a wielowątkowość

W ramach procesu MPI możemy dodatkowo uruchamiać dodatkowe wątki

4 tryby pracy:

- Pojedynczy – proces ma jeden wątek
- Funneled – MPI będzie wołany tylko z wątku, który zainicjalizował MPI
- Szeregowy – MPI będzie wołany jednocześnie tylko z jednego wątku
- Równoległy – MPI może być wołany z wielu wątków jednocześnie

```
int MPI_Init_thread(int* argc, char*** argv,
                    int required, int* provided);
```

Popularne modele/biblioteki programowania wielowątkowego, takie jak OpenMP czy TBB poprawnie współpracują ze Slurmem i MPI w zakresie ustawiania domyślnej liczby agentów

Podsumowanie

Programowanie w modelu pamięci rozproszonej jest najbardziej skalowalnym podejściem do równoległości

MPI służy do wymiany danych pomiędzy procesami, które potencjalnie mogą znajdować się na innych maszynach

MPI oferuje wiele abstrakcji, takich jak komunikacja zbiorowa i nieblokująca, które pozwalają na tworzenie wydajnych i poprawnych aplikacji

Aplikacje MPI uruchamiamy przeważnie przy pomocy systemu kolejkowego

MPI nie wyklucza (a wręcz wspiera) inne modele równoległości

Tematy nieporuszone na szkoleniu

Wirtualne topologie komunikatorów

Rozproszone I/O

Komunikacja jednostronna

Tworzenie własnych typów danych i operacji

Szerszy rys historyczny

Dziękuję za uwagę!

W razie dodatkowych pytań zapraszam do kontaktu pod adresem
`jgalecki@icm.edu.pl`

Sesja praktyczna

Zadanie 1 – Hello MPI

Napisz program, który wydrukuje z każdego procesu:

- Swój rząd
- Rozmiar komunikatora

Czy kolejność drukowanych komunikatów jest deterministyczna? Dlaczego?

Jak osiągnąć deterministyczną kolejność?

Zadanie 2 – Całka metodą trapezów

Mając dany uporządkowany zbiór punktów $X = \{x_0, ..., x_N\}$ oraz odpowiadający im zbiór $Y = \{y_i\}$, możemy przybliżyć całkę funkcji $y = f(x)$ ze wzoru

$$\int_{x_0}^{x_N} f(x)\,\mathrm{d}x \approx \frac{1}{2}\sum_{i=0}^{N-1}(x_{i+1} - x_i)\cdot(y_i + y_{i+1})$$

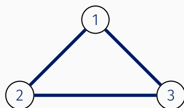
Daną masz funkcję f oraz zbiór X, podzielony równomiernie między procesy MPI. Oblicz przybliżoną wartość całki f po zakresie określonym przez X. Wydrukuj ją z każdego procesu.

Jakiej komunikacji użyjesz, aby zoptymalizować czas wykonania programu?

Zadanie 3 – partycjonowanie kratownicy 2D

Masz daną kratownicę – zbiór prętów, których końce (węzły) mogą być ze sobą dowolnie połączone. Każdy pręt kratownicy opisany jest 2 liczbami – numerami węzłów na obu końcach. Topologia kratownicy opisana jest dwuwymiarową tablicą T o wymiarach $2 \times N_e$, gdzie N_e to liczba pretów. i-ta kolumna tablicy zawiera opis i-tego pręta. Dodatkowo dane są tablice X i Y zawierająca współrzędne węzłów. Twoim zadaniem jest podział i rozesłanie kratownicy znajdującej się początkowo w całości w pamięci procesu 0.

Uwaga: nie rozsyłaj całych tablic X i Y, a jedynie wymagane elementy – konieczna jest przenumerowanie topologii poszczególnych partycji.



$$t = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{bmatrix} \qquad x = \begin{bmatrix} 0 & -1 & 1 \end{bmatrix} \qquad y = \begin{bmatrix} 0 & -1 & -1 \end{bmatrix}$$